GBATEK

Gameboy Advance Technical Info - Extracted from no$gba version 1.4

GBA Reference

CPU Reference

Technical Data

CPU Modes

ARM Mode	ARM7TDMI 32bit RISC CPU, 16.78MHz, 32bit opcodes (GBA)
THUMB Mode	ARM7TDMI 32bit RISC CPU, 16.78MHz, 16bit opcodes (GBA)
CGB Mode	Z80/8080-style 8bit CPU, 4.2MHz or 8.4MHz (CGB compatibility)
DMG Mode	Z80/8080-style 8bit CPU, 4.2MHz (monochrome gameboy compatib.)

Internal Memory

BIOS ROM	16 KBytes
Work RAM	288 KBytes (32K in-chip + 256K on-board)
VRAM	96 KBytes
OAM	1 KByte (128 OBJs 3x16bit, 32 OBJ-Rotation/Scalings 4x16bit)
Palette RAM	1 KByte (256 BG colors, 256 OBJ colors)

Video

Display	240x160 pixels (2.9 inch TFT color LCD display)
BG layers	4 background layers
BG types	Tile/map based, or Bitmap based
BG colors	256 colors, or 16 colors/16 palettes, or 32768 colors
OBJ colors	256 colors, or 16 colors/16 palettes
Effects	Rotation/Scaling, alpha blending, fade-in/out, mosaic, window
OBJ size	12 types (in range 8x8 up to 64x64 dots)
OBJs/Screen	max. 128 OBJs of any size (up to 64x64 dots each)
OBJs/Line	max. 128 OBJs of 8x8 dots size (under best circumstances)
Priorities	OBJ/OBJ: 0-127, OBJ/BG: 0-3, BG/BG: 0-3
Effects	Rotation/Scaling, alpha blending, fade-in/out, mosaic, window

Sound

```
Analogue     4 channel CGB compatible
Digital      2 DMA sound channels
Output       Built-in speaker, or stereo headphones
```

Controls

```
Gamepad      4 Direction Keys, 6 Buttons
```

Communication Ports

```
Serial Port  Various transfer modes, 4-Player Link, Single Game Pak play
```

External Memory

```
GBA Game Pak max. 32MB ROM or flash ROM + max 64K SRAM
CGB Game Pak max. 32KB ROM + 8KB SRAM (more memory requires banking)
```

Power Supply

```
Battery      Life-time approx. 15 hours
External     3.3V DC (works with somewhat 2.7V-3.3V, or maybe a bit more)
```

The separate CPU modes cannot be operated simultaneously. Switching is allowed between ARM and THUMB modes only (that are the two GBA modes).
This manual does not describe CGB and DMG modes, both are completely different than GBA modes, and both cannot be accessed from inside of GBA modes anyways.

Memory Map

General Internal Memory

```
0000:0000-0000:3FFF   BIOS - System ROM    (16 KBytes)
0000:4000-01FF:FFFF   Not used
0200:0000-0203:FFFF   WRAM - On-board Work RAM  (256 KBytes) 2 Wait
0204:0000-02FF:FFFF   Not used
0300:0000-0300:7FFF   WRAM - In-chip Work RAM   (32 KBytes)
0300:8000-03FF:FFFF   Not used
0400:0000-0400:03FE   I/O Registers
0400:0400-04FF:FFFF   Not used
```

Internal Display Memory

```
0500:0000-0500:03FF   BG/OBJ Palette RAM     (1 Kbyte)
0500:0400-05FF:FFFF   Not used
0600:0000-0617:FFFF   VRAM - Video RAM     (96 KBytes)
0618:0000-06FF:FFFF   Not used
0700:0000-0700:03FF   OAM - OBJ Attributes   (1 Kbyte)
0700:0400-07FF:FFFF   Not used
```

External Memory (Game Pak)

```
0800:0000-09FF:FFFF   Game Pak ROM/FlashROM (max 32MB) - Wait State 0
0A00:0000-0BFF:FFFF   Game Pak ROM/FlashROM (max 32MB) - Wait State 1
0C00:0000-0DFF:FFFF   Game Pak ROM/FlashROM (max 32MB) - Wait State 2
0E00:0000-0E00:FFFF   Game Pak SRAM    (max 64 KBytes) - 8bit Bus width
0E01:0000-0FFF:FFFF   Not used
```

Unused Memory Area

```
1000:0000-FFFF:FFFF   Not used (upper 4bits of address bus unused)
```

Default WRAM Usage

By default, the 256 bytes at 0300:7F00h-0300:7FFFh in Work RAM are reserved for Interrupt vector, Interrupt Stack, and BIOS Call Stack. The remaining WRAM is free for whatever use (including User Stack, which is initially located at 0300:7F00h).

Address Bus Width and CPU Read/Write Access Widths
Shows the Bus-Width, supported read and write widths, and the clock cycles for 8/16/32bit accesses.

Region	Bus	Read	Write	Cycles
BIOS ROM	32	8/16/32	-	1/1/1
Work RAM 32K	32	8/16/32	8/16/32	1/1/1
I/O	32	8/16/32	8/16/32	1/1/1
OAM	32	8/16/32	16/32	1/1/1 *
Work RAM 256K	16	8/16/32	8/16/32	3/3/6 **
Palette RAM	16	8/16/32	16/32	1/1/2 *
VRAM	16	8/16/32	16/32	1/1/2 *
GamePak ROM	16	8/16/32	-	5/5/8 **/***
GamePak Flash	16	8/16/32	16/32	5/5/8 **/***
GamePak SRAM	8	8	8	5 **

Timing Notes:

```
*   Plus 1 cycle if GBA accesses video memory at the same time.
**  Default waitstate settings, see System Control chapter.
*** Separate timings for sequential, and non-sequential accesses.
One cycle equals approx. 59.59ns (ie. 16.78MHz clock).
```

All memory (except GamePak SRAM) can be accessed by 16bit and 32bit DMA.

GamePak Memory
Only DMA3 (and the CPU of course) may access GamePak ROM. GamePak SRAM can be accessed by the CPU only - restricted to bytewise 8bit transfers. SRAM is supposed as external WRAM expansion - not for battery-buffered data storage - for that purpose it'd be more recommended to use a Flash ROM chip somewhere located in the ROM area.
For details about configuration of GamePak Waitstates, read respective chapter.
SRAM should be accessed only through library ???

VRAM, OAM, and Palette RAM Access
These memory regions can be accessed during H-Blank or V-Blank only (unless display is disabled by Forced Blank bit in DISPCNT register).
There is an additional restriction for OAM memory: Accesses during H-Blank are allowed only if 'H-Blank Interval Free' in DISPCNT is set (which'd reduce number of display-able OBJs though).
The CPU appears to be able to access VRAM/OAM/Palette at any time, a waitstate (one clock cycle) being inserted automatically in case that the display controller was accessing memory simultaneously. (Ie. unlike as in old 8bit gameboy, the data will not get lost.)

CPU Mode Performance
Note that the GamePak ROM bus is limited to 16bits, thus executing ARM instructions (32bit opcodes) from inside of GamePak ROM would result in a not so good performance. So, it'd be more recommended to use THUMB instruction (16bit opcodes) which'd allow each opcode to be read at once.
(ARM instructions can be used at best performance by copying code from GamePak ROM into internal Work RAM)

Data Format
Even though the ARM CPU itself would allow to select between Little-Endian and Big-Endian format by using an external circuit, in the GBA no such circuit exists, and the data format is always Little-Endian. That is, when accessing 16bit or 32bit data in memory, the least significant bits are stored in the first byte (smallest address), and the most significant bits in the last byte. (Ie. same as for 80x86 and Z80 CPUs.)

I/O Map

Forward

The base address for GBA I/O ports is 04000000h - all address below are actually meant to be located at 04000NNNh in memory rather than at NNNh.

LCD I/O Registers

```
000h       R/W  DISPCNT   LCD Control
002h       R/W  -         Undocumented - Green Swap
004h       R/W  DISPSTAT  General LCD Status (STAT,LYC)
006h       R    VCOUNT    Vertical Counter (LY)
008h       R/W  BG0CNT    BG0 Control
00Ah       R/W  BG1CNT    BG1 Control
00Ch       R/W  BG2CNT    BG2 Control
00Eh       R/W  BG3CNT    BG3 Control
010h       W    BG0HOFS   BG0 X-Offset
012h       W    BG0VOFS   BG0 Y-Offset
014h       W    BG1HOFS   BG1 X-Offset
016h       W    BG1VOFS   BG1 Y-Offset
018h       W    BG2HOFS   BG2 X-Offset
01Ah       W    BG2VOFS   BG2 Y-Offset
01Ch       W    BG3HOFS   BG3 X-Offset
01Eh       W    BG3VOFS   BG3 Y-Offset
020h       W    BG2PA     BG2 Rotation/Scaling Parameter A (dx)
022h       W    BG2PB     BG2 Rotation/Scaling Parameter B (dmx)
024h       W    BG2PC     BG2 Rotation/Scaling Parameter C (dy)
026h       W    BG2PD     BG2 Rotation/Scaling Parameter D (dmy)
028h-02Ah  W    BG2X      BG2 Reference Point X-Coordinate
02Ch-02Eh  W    BG2Y      BG2 Reference Point Y-Coordinate
030h       W    BG3PA     BG3 Rotation/Scaling Parameter A (dx)
032h       W    BG3PB     BG3 Rotation/Scaling Parameter B (dmx)
034h       W    BG3PC     BG3 Rotation/Scaling Parameter C (dy)
036h       W    BG3PD     BG3 Rotation/Scaling Parameter D (dmy)
038h-03Ah  W    BG3X      BG3 Reference Point X-Coordinate
03Ch-03Eh  W    BG3Y      BG3 Reference Point Y-Coordinate
040h       W    WIN0H     Window 0 Horizontal Dimensions
042h       W    WIN1H     Window 1 Horizontal Dimensions
044h       W    WIN0V     Window 0 Vertical Dimensions
046h       W    WIN1V     Window 1 Vertical Dimensions
048h       R/W  WININ     Control Inside of Window(s)
04Ah       R/W  WINOUT    Control Outside of Windows & Inside of OBJ Window
04Ch       W    MOSAIC    Mosaic Size
04Eh       -    -         Not used
050h       R/W  BLDCNT    Color Special Effects Selection   (formerly BLDMOD)
052h       W    BLDALPHA  Alpha Blending Coefficients        (formerly COLEV)
054h       W    BLDY      Brightness (Fade-In/Out) Coefficient(formerly COLY)
056h-05Eh  -    -         Not used
```

Sound Registers

```
060h     R/W  SOUND1CNT_L Channel 1 Sweep register       (SG10_L)(NR10)
062h     R/W  SOUND1CNT_H Channel 1 Duty/Length/Envelope (SG10_H)(NR11, NR12)
064h     R/W  SOUND1CNT_X Channel 1 Frequency/Control    (SG11)  (NR13, NR14)
066h     -    -           Not used                        -
068h     R/W  SOUND2CNT_L Channel 2 Duty/Length/Envelope (SG20)  (NR21, NR22)
06Ah     -    -           Not used                        -
06Ch     R/W  SOUND2CNT_H Channel 2 Frequency/Control    (SG21)  (NR23, NR24)
06Eh     -    -           Not used                        -
070h     R/W  SOUND3CNT_L Channel 3 Stop/Wave RAM select (SG30_L)(NR30)
072h     R/W  SOUND3CNT_H Channel 3 Length/Volume        (SG30_H)(NR31, NR32)
074h     R/W  SOUND3CNT_X Channel 3 Frequency/Control    (SG31)  (NR33, NR34)
076h     -    -           Not used                        -
078h     R/W  SOUND4CNT_L Channel 4 Length/Envelope      (SG40)  (NR41, NR42)
07Ah     -    -           Not used                        -
07Ch     R/W  SOUND4CNT_H Channel 4 Frequency/Control    (SG41)  (NR43, NR44)
07Eh     -    -           Not used                        -
080h     R/W  SOUNDCNT_L  Control Stereo/Volume/Enable (SGCNT0_L)(NR50, NR51)
082h     R/W  SOUNDCNT_H  Control Mixing/DMA Control   (SGCNT0_H)
084h     R/W  SOUNDCNT_X  Control Sound on/off         (SGCNT1)  (NR52)
086h     -    -           Not used
088h     BIOS SOUNDBIAS   Sound PWM Control            (SG_BIAS)
08Ah-08Eh  -    -         Not used
```

```
090h-09Eh  R/W  WAVE_RAM  Channel 3 Wave Pattern RAM (2 banks!!) (SGWR)
0A0h-0A2h  W    FIFO_A    Channel A FIFO, Data 0-3 (SGFIFOA)
0A4h-0A6h  W    FIFO_B    Channel B FIFO, Data 0-3 (SGFIFOB)
0A8h-0AEh  -    -         Not used
```

DMA Transfer Channels

```
0B0h-0B2h  W    DMA0SAD   DMA 0 Source Address
0B4h-0B6h  W    DMA0DAD   DMA 0 Destination Address
0B8h       W    DMA0CNT_L DMA 0 Word Count
0BAh       R/W  DMA0CNT_H DMA 0 Control
0BCh-0BEh  W    DMA1SAD   DMA 1 Source Address
0C0h-0C2h  W    DMA1DAD   DMA 1 Destination Address
0C4h       W    DMA1CNT_L DMA 1 Word Count
0C6h       R/W  DMA1CNT_H DMA 1 Control
0C8h-0CAh  W    DMA2SAD   DMA 2 Source Address
0CCh-0CEh  W    DMA2DAD   DMA 2 Destination Address
0D0h       W    DMA2CNT_L DMA 2 Word Count
0D2h       R/W  DMA2CNT_H DMA 2 Control
0D4h-0D6h  W    DMA3SAD   DMA 3 Source Address
0D8h-0DAh  W    DMA3DAD   DMA 3 Destination Address
0DCh       W    DMA3CNT_L DMA 3 Word Count
0DEh       R/W  DMA3CNT_H DMA 3 Control
0E0h-0FEh  -    -         Not used
```

Timer Registers

```
100h       R/W  TM0CNT_L  Timer 0 Counter/Reload (formerly TM0D)
102h       R/W  TM0CNT_H  Timer 0 Control        (formerly TM0CNT)
104h       R/W  TM1CNT_L  Timer 1 Counter/Reload (formerly TM1D)
106h       R/W  TM1CNT_H  Timer 1 Control        (formerly TM1CNT)
108h       R/W  TM2CNT_L  Timer 2 Counter/Reload (formerly TM2D)
10Ah       R/W  TM2CNT_H  Timer 2 Control        (formerly TM2CNT)
10Ch       R/W  TM3CNT_L  Timer 3 Counter/Reload (formerly TM3D)
10Eh       R/W  TM3CNT_H  Timer 3 Control        (formerly TM3CNT)
110h-11Eh  -    -         Not used
```

Serial Communication (1)

```
120h-122h  R/W  SIODATA32 SIO Data (Normal-32bit Mode) (shared with below!)
120h       R/W  SIOMULTI0 SIO Data 0 (Parent)    (Multi-Player Mode) (SCD0)
122h       R/W  SIOMULTI1 SIO Data 1 (1st Child) (Multi-Player Mode) (SCD1)
124h       R/W  SIOMULTI2 SIO Data 2 (2nd Child) (Multi-Player Mode) (SCD2)
126h       R/W  SIOMULTI3 SIO Data 3 (3rd Child) (Multi-Player Mode) (SCD3)
128h       R/W  SIOCNT    SIO Control Register                    (SCCNT_L)
12Ah       R/W  SIOMLT_SEND SIO Data (Local of Multi-Player) (shared below)
12Ah       R/W  SIODATA8  SIO Data (Normal-8bit and UART Mode)    (SCCNT_H)
12Ch-12Eh  -    -         Not used
```

Keypad Input

```
130h       R    KEYINPUT  Key Status            (formerly P1)
132h       R/W  KEYCNT    Key Interrupt Control (formerly P1CNT)
```

Serial Communication (2)

```
134h       R/W  RCNT      SIO Mode Select/General Purpose Data (formerly R)
136h       -    IR        Ancient - Infrared Register (Prototypes only)
138h-13Eh  -    -         Not used
140h       R/W  JOYCNT    SIO JOY Bus Control         (formerly HS_CTRL)
142h-14Eh  -    -         Not used
150h-152h  R/W  JOY_RECV  SIO JOY Bus Receive Data    (formerly JOYRE)
154h-156h  R/W  JOY_TRANS SIO JOY Bus Transmit Data   (formerly JOYTR)
158h       R/?  JOYSTAT   SIO JOY Bus Receive Status  (formerly JSTAT)
15Ah-1FEh  -    -         Not used
```

Interrupt, Waitstate, and Power-Down Control

```
200h       R/W  IE        Interrupt Enable Register
202h       R/W  IF        Interrupt Request Flags / IRQ Acknowledge
204h       R/W  WAITCNT   Game Pak Waitstate Control (formerly WSCNT)
206h       -    -         Not used
```

```
208h        R/W  IME       Interrupt Master Enable Register
20Ah-2FFh   -    -         Not used
300h        R/W  HALTCNT   Undocumented - Power Down Control
302h-40Fh   -    -         Not used
410h        ?    ?         Undocumented - Purpose Unknown ??? 0FFh
411h-7FFh   -    -         Not used
800h-802h   R/W  ?         Undocumented - Internal Memory Control (R/W)
804h-FFFFh  -    -         Not used
```

All further addresses at 4XXXXXXh are unused and do not contain mirrors of the I/O area, with the only exception that 800h-802h is repeated each 64K (ie. mirrored at 10800h, 20800h, etc.)

LCD Video Controller

Registers
LCD I/O Display Control
LCD I/O Interrupts and Status
LCD I/O BG Control
LCD I/O BG Scrolling
LCD I/O BG Rotation/Scaling
LCD I/O Window Feature
LCD I/O Mosaic Function
LCD I/O Color Special Effects

VRAM
LCD VRAM Overview
LCD VRAM Character Data
LCD VRAM BG Screen Data Format (BG Map)
LCD VRAM Bitmap BG Modes

Sprites
LCD OBJ - Overview
LCD OBJ - OAM Attributes
LCD OBJ - OAM Rotation/Scaling Parameters
LCD OBJ - VRAM Character (Tile) Mapping

Other
LCD Color Palettes
LCD Dimensions and Timings

LCD I/O Display Control

000h - DISPCNT - LCD Control (Read/Write)

```
Bit   Expl.
0-2   BG Mode                 (0-5=Video Mode 0-5, 6-7=Prohibited)
3     Reserved for BIOS       (CGB Mode - cannot be changed after startup)
4     Display Frame Select    (0-1=Frame 0-1) (for BG Modes 4,5 only)
5     H-Blank Interval Free   (1=Allow access to OAM during H-Blank)
6     OBJ Character VRAM Mapping (0=Two dimensional, 1=One dimensional)
7     Forced Blank            (1=Allow access to VRAM,Palette,OAM)
8     Screen Display BG0  (0=Off, 1=On)
9     Screen Display BG1  (0=Off, 1=On)
10    Screen Display BG2  (0=Off, 1=On)
11    Screen Display BG3  (0=Off, 1=On)
```

```
12    Screen Display OBJ  (0=Off, 1=On)
13    Window 0 Display Flag   (0=Off, 1=On)
14    Window 1 Display Flag   (0=Off, 1=On)
15    OBJ Window Display Flag (0=Off, 1=On)
```

The table summarizes the facilities of the separate BG modes (video modes).

Mode	Rot/Scal	Layers	Size	Tiles	Colors	Features
0	No	0123	256x256..512x515	1024	16/16..256/1	SFMABP
1	Mixed	012-	(BG0,BG1 as above Mode 0, BG2 as below Mode 2)			
2	Yes	--23	128x128..1024x1024	256	256/1	S-MABP
3	Yes	--?-	240x160	1	32768	--MABP
4	Yes	--??	240x160	2	256/1	--MABP
5	Yes	--??	160x128	2	32768	--MABP

Features: S)crolling, F)lip, M)osaic, A)lphaBlending, B)rightness, P)riority.

BG Modes 0-2 are Tile/Map-based. BG Modes 3-5 are Bitmap-based, in these modes 1 or 2 Frames (ie. bitmaps, or 'full screen tiles') exists, if two frames exist, either one can be displayed, and the other one can be redrawn in background.

Blanking Bits
Setting Forced Blank (Bit 7) causes the video controller to display white lines, and all VRAM, Palette RAM, and OAM may be accessed.
"When the internal HV synchronous counter cancels a forced blank during a display period, the display begins from the beginning, following the display of two vertical lines." What ???
Setting H-Blank Interval Free (Bit 5) allows to access OAM during H-Blank time - using this feature reduces the number of sprites that can be displayed per line.

Display Enable Bits
By default, BG0-3 and OBJ Display Flags (Bit 8-12) are used to enable/disable BGs and OBJ. When enabeling Window 0 and/or 1 (Bit 13-14), color special effects may be used, and BG0-3 and OBJ are controlled by the window(s).

Frame Selection
In BG Modes 4 and 5 (Bitmap modes), either one of the two bitmaps/frames may be displayed (Bit 4), allowing the user to update the other (invisible) frame in background. In BG Mode 3, only one frame exists.
In BG Modes 0-2 (Tile/Map based modes), a similiar effect may be gained by altering the base address(es) of BG Map and/or BG Character data.

002h - Undocumented - Green Swap (R/W)
Normally, red green blue intensities for a group of two pixels is output as BGRbgr (uppercase for left pixel at even xloc, lowercase for right pixel at odd xloc). When the Green Swap bit is set, each pixel group is output as BgRbGr (ie. green intensity of each two pixels exchanged).

```
Bit   Expl.
0     Green Swap  (0=Normal, 1=Swap)
1-15  Not used
```

This feature appears to be applied to the final picture (ie. after mixing the separate BG and OBJ layers). Eventually intended for other display types (with other pin-outs). With normal GBA hardware it is just producing an interesting dirt effect.

LCD I/O Interrupts and Status

004h - DISPSTAT - General LCD Status (Read/Write)

Display status and Interrupt control. The H-Blank conditions are generated once per scanline, including for the 'hidden' scanlines during V-Blank.

```
Bit   Expl.
0     V-Blank flag    (Read only) (1=VBlank)
1     H-Blank flag    (Read only) (1=HBlank)
2     V-Counter flag  (Read only) (1=Match)
3     V-Blank IRQ Enable          (1=Enable)
4     H-Blank IRQ Enable          (1=Enable)
5     V-Counter IRQ Enable        (1=Enable)
6-7   Not used
8-15  V-Count Setting             (0-227)
```

The V-Count-Setting value is much the same as LYC of older gameboys, when its value is identical to the content of the VCOUNT register then the V-Counter flag is set (Bit 2), and (if enabled in Bit 5) an interrupt is requested.

006h - VCOUNT - Vertical Counter (Read only)
Indicates the currently drawn scanline, values in range from 160-227 indicate 'hidden' scanlines within VBlank area.

```
Bit   Expl.
0-7   Current scanline (0-227)
8-15  Not Used
```

Note: This is much the same than the 'LY' register of older gameboys.

LCD I/O BG Control

008h - BG0CNT - BG0 Control (R/W) (BG Modes 0,1 only)
00Ah - BG1CNT - BG1 Control (R/W) (BG Modes 0,1 only)
00Ch - BG2CNT - BG2 Control (R/W) (BG Modes 0,1,2 only)
00Eh - BG3CNT - BG3 Control (R/W) (BG Modes 0,2 only)

```
Bit   Expl.
0-1   BG Priority           (0-3, 0=Highest)
2-3   Character Base Block  (0-3, in units of 16 KBytes) (=BG Tile Data)
4-5   Not used (must be zero)
6     Mosaic                (0=Disable, 1=Enable)
7     Colors/Palettes       (0=16/16, 1=256/1)
8-12  Screen Base Block     (0-31, in units of 2 KBytes) (=BG Map Data)
13    Display Area Overflow (0=Transparent, 1=Wraparound; BG2CNT/BG3CNT only)
14-15 Screen Size (0-3)
```

Internal Screen Size (dots) and size of BG Map (bytes):

```
Value  Text Mode      Rotation/Scaling Mode
0      256x256 (2K)   128x128   (256 bytes)
1      512x256 (4K)   256x256   (1K)
2      256x512 (4K)   512x512   (4K)
3      512x512 (8K)   1024x1024 (16K)
```

In case that some or all BGs are set to same priority then BG0 is having the highest, and BG3 the lowest priotity.

In 'Text Modes', the screen size is organized as follows: The screen consists of one or more 256x256 pixel (32x32 tiles) areas. When Size=0: only 1 area (SC0), when Size=1 or Size=2: two areas (SC0,SC1 either horizontally or vertically arranged next to each other), when Size=3: four areas (SC0,SC1 in upper row, SC2,SC3 in lower row). Whereas SC0 is defined by the normal BG Map base address (Bit 8-12 of BG#CNT), SC1 uses same address +2K, SC2 address +4K, SC3 address +6K. When the screen is scrolled it'll always wraparound.

In 'Rotation/Scaling Modes', the screen size is organized as follows, only one area (SC0) of variable size 128x128..1024x1024 pixels (16x16..128x128 tiles) exists (SC0). When the screen is rotated/scaled (or scrolled?) so that the LCD viewport reaches outside of the background/screen area, then BG may be either displayed as transparent or wraparound (Bit 13 or BG#CNT).

LCD I/O BG Scrolling

010h - BG0HOFS - BG0 X-Offset (W)
012h - BG0VOFS - BG0 Y-Offset (W)

```
Bit   Expl.
0-8   Offset (0-511)
9-15  Not used
```

Specifies the coordinate of the upperleft first visible dot of BG0 background layer, ie. used to scroll the BG0 area.

014h - BG1HOFS - BG1 X-Offset (W)
016h - BG1VOFS - BG1 Y-Offset (W)
Same as above BG0HOFS and BG0VOFS for BG1 respectively.

018h - BG2HOFS - BG2 X-Offset (W)
01Ah - BG2VOFS - BG2 Y-Offset (W)
Same as above BG0HOFS and BG0VOFS for BG2 respectively.

01Ch - BG3HOFS - BG3 X-Offset (W)
01Eh - BG3VOFS - BG3 Y-Offset (W)
Same as above BG0HOFS and BG0VOFS for BG3 respectively.

The above BG scrolling registers are exclusively used in Text modes, ie. for all layers in BG Mode 0, and for the first two layers in BG mode 1.
In other BG modes (Rotation/Scaling and Bitmap modes) above registers are ignored. Instead, the screen may be scrolled by modifying the BG Rotation/Scaling Reference Point registers.

LCD I/O BG Rotation/Scaling

028h - BG2X_L - BG2 Reference Point X-Coordinate, lower 16 bit (W)
02Ah - BG2X_H - BG2 Reference Point X-Coordinate, upper 12 bit (W)
02Ch - BG2Y_L - BG2 Reference Point Y-Coordinate, lower 16 bit (W)
02Eh - BG2Y_H - BG2 Reference Point Y-Coordinate, upper 12 bit (W)
These registers are replacing the BG scrolling registers which are used for Text mode, ie. the X/Y coordinates specify the source position from inside of the BG Map/Bitmap of the pixel to be displayed at upper left of the GBA display. The normal BG scrolling registers are ignored in Rotation/Scaling and Bitmap modes.

```
Bit   Expl.
0-7   Fractional portion (8 bits)
8-26  Integer portion    (19 bits)
27    Sign               (1 bit)
28-31 Not used
```

Because values are shifted left by eight, fractional portions may be specified in steps of 1/256 pixels (this would be relevant only if the screen is actually rotated or scaled). Normal signed 32bit values may be written to above registers (the most significant bits will be ignored and the value will be cut-down to 28bits, but this is no actual problem because signed values have set all MSBs to the same value).

Internal Reference Point Registers
The above reference points are automatically copied to internal registers during each vblank, specifying the origin for the first scanline. The internal registers are then incremented by dmx and dmy after each scanline.
Caution: Writing to a reference point register by software outside of the Vblank period does immediately copy the new value to the corresponding internal register, that means: in the current frame, the new value specifies the origin of the <current> scanline (instead of the topmost scanline).

020h - BG2PA - BG2 Rotation/Scaling Parameter A (alias dx) (W)
022h - BG2PB - BG2 Rotation/Scaling Parameter B (alias dmx) (W)
024h - BG2PC - BG2 Rotation/Scaling Parameter C (alias dy) (W)
026h - BG2PD - BG2 Rotation/Scaling Parameter D (alias dmy) (W)

```
Bit   Expl.
0-7   Fractional portion (8 bits)
8-14  Integer portion    (7 bits)
15    Sign               (1 bit)
```

See below for details.

03Xh - BG3X_L/H, BG3Y_L/H, BG3PA-D - BG3 Rotation/Scaling Parameters
Same as above BG2 Reference Point, and Rotation/Scaling Parameters, for BG3 respectively.

dx (PA) and dy (PC)
When transforming a horizontal line, dx and dy specify the resulting gradient and magnification for that line. For example:
Horizontal line, length=100, dx=1, and dy=1. The resulting line would be drawn at 45 degrees, f(y)=1/1*x. Note that this would involve that line is magnified, the new length is SQR(100^2+100^2)=141.42. Yup, exactly - that's the old a^2 + b^2 = c^2 formula.

dmx (PB) and dmy (PD)
These values define the resulting gradient and magnification for transformation of vertical lines. However, when rotating a square area (which is surrounded by horizontal and vertical lines), then the desired result should be usually a rotated <square> area (ie. not a parallelogram, for example).
Thus, dmx and dmy must be defined in direct relationship to dx and dy, taking the example above, we'd have to set dmx=-1, and dmy=1, f(x)=-1/1*y.

Area Overflow
In result of rotation/scaling it may often happen that areas outside of the actual BG area become moved into the LCD viewport. Depending of the Area Overflow bit (BG2CNT and BG3CNT, Bit 13) these areas may be either displayed (by wrapping the BG area), or may be displayed transparent.
This works only in BG modes 1 and 2. The area overflow is ignored in Bitmap modes (BG modes 3-5), the outside of the Bitmaps is always transparent.

--- more details and confusing or helpful formulas ---

The following parameters are required for Rotation/Scaling

```
Rotation Center X and Y Coordinates (x0,y0)
Rotation Angle                      (alpha)
Magnification X and Y Values        (xMag,yMag)
```

The display is rotated by 'alpha' degrees around the center.
The displayed picture is magnified by 'xMag' along x-Axis (Y=y0) and 'yMag' along y-Axis (X=x0).

Calculating Rotation/Scaling Parameters A-D

```
A = Cos (alpha) / xMag    ;distance moved in direction x, same line
```

```
B = Sin (alpha) / xMag    ;distance moved in direction x, next line
C = Sin (alpha) / yMag    ;distance moved in direction y, same line
D = Cos (alpha) / yMag    ;distance moved in direction y, next line
```

Calculating the position of a rotated/scaled dot
Using the following expressions,

```
x0,y0    Rotation Center
x1,y1    Old Position of a pixel (before rotation/scaling)
x2,y2    New position of above pixel (after rotation scaling)
A,B,C,D  BG2PA-BG2PD Parameters (as calculated above)
```

the following formula can be used to calculate x2,y2:

```
x2 = A(x1-x0) + B(y1-y0) + x0
y2 = C(x1-x0) + D(y1-y0) + y0
```

LCD I/O Window Feature

The Window Feature may be used to split the screen into four regions. The BG0-3,OBJ layers and Color Special Effects can be separately enabled or disabled in each of these regions.

The DISPCNT Register
DISPCNT Bits 13-15 are used to enable Window 0, Window 1, and/or OBJ Window regions, if any of these regions is enabled then the "Outside of Windows" region is automatically enabled, too.
DISPCNT Bits 8-12 are kept used as master enable bits for the BG0-3,OBJ layers, a layer is displayed only if both DISPCNT and WININ/OUT enable bits are set.

040h - WIN0H - Window 0 Horizontal Dimensions (W)
042h - WIN1H - Window 1 Horizontal Dimensions (W)

```
Bit   Expl.
0-7   X2, Rightmost coordinate of window, plus 1
8-15  X1, Leftmost coordinate of window
```

044h - WIN0V - Window 0 Vertical Dimensions (W)
046h - WIN1V - Window 1 Vertical Dimensions (W)

```
Bit   Expl.
0-7   Y2, Bottom-most coordinate of window, plus 1
8-15  Y1, Top-most coordinate of window
```

048h - WININ - Control of Inside of Window(s) (R/W)

```
Bit   Expl.
0-3   Window 0 BG0-BG3 Enable Bits     (0=No Display, 1=Display)
4     Window 0 OBJ Enable Bit          (0=No Display, 1=Display)
5     Window 0 Color Special Effect    (0=Disable, 1=Enable)
6-7   Not used
8-11  Window 1 BG0-BG3 Enable Bits     (0=No Display, 1=Display)
12    Window 1 OBJ Enable Bit          (0=No Display, 1=Display)
13    Window 1 Color Special Effect    (0=Disable, 1=Enable)
14-15 Not used
```

04Ah - WINOUT - Control of Outside of Windows & Inside of OBJ Window (R/W)

```
Bit   Expl.
0-3   Outside BG0-BG3 Enable Bits      (0=No Display, 1=Display)
4     Outside OBJ Enable Bit           (0=No Display, 1=Display)
```

```
5     Outside Color Special Effect     (0=Disable, 1=Enable)
6-7   Not used
8-11  OBJ Window BG0-BG3 Enable Bits   (0=No Display, 1=Display)
12    OBJ Window OBJ Enable Bit        (0=No Display, 1=Display)
13    OBJ Window Color Special Effect  (0=Disable, 1=Enable)
14-15 Not used
```

The OBJ Window
The dimension of the OBJ Window is specified by OBJs which are having the "OBJ Mode" attribute being set to "OBJ Window". Any non-transparent dots of any such OBJs are marked as OBJ Window area. The OBJ itself is not displayed.
The color, palette, and display priority of these OBJs are ignored. Both DISPCNT Bits 12 and 15 must be set when defining OBJ Window region(s).

Window Priority
In case that more than one window is enabled, and that these windows do overlap, Window 0 is having highest priority, Window 1 medium, and Obj Window lowest priority. Outside of Window is having zero priority, it is used for all dots which are not inside of any window region.

LCD I/O Mosaic Function

04Ch - MOSAIC - Mosaic Size (W)
The Mosaic function can be separately enabled/disabled for BG0-BG3 by BG0CNT-BG3CNT Registers, as well as for each OBJ0-127 by OBJ attributes in OAM memory. Also, setting all of the bits below to zero effectively disables the mosaic function.

```
Bit   Expl.
0-3   BG Mosaic H-Size  (minus 1)
4-7   BG Mosaic V-Size  (minus 1)
8-11  OBJ Mosaic H-Size (minus 1)
12-15 OBJ Mosaic V-Size (minus 1)
```

Example: When setting H-Size to 5, then pixels 0-5 of each display row are colorized as pixel 0, pixels 6-11 as pixel 6, pixels 12-17 as pixel 12, and so on.

Normally, a 'mosaic-pixel' is colorized by the color of the upperleft covered pixel. In many cases it might be more desireful to use the color of the pixel in the center of the covered area - this effect may be gained by scrolling the background (or by adjusting the OBJ position, as far as upper/left rows/columns of OBJ are transparent).

LCD I/O Color Special Effects

Two types of Special Effects are supported: Alpha Blending (Semi-Transparency) allows to combine colors of two selected surfaces. Brightness Increase/Decrease adjust the brightness of the selected surface.

050h - BLDCNT (formerly BLDMOD) - Color Special Effects Selection (R/W)

```
Bit   Expl.
0     BG0 1st Target Pixel (Background 0)
1     BG1 1st Target Pixel (Background 1)
2     BG2 1st Target Pixel (Background 2)
3     BG3 1st Target Pixel (Background 3)
4     OBJ 1st Target Pixel (Top-most OBJ pixel)
5     BD  1st Target Pixel (Backdrop)
6-7   Color Special Effect (0-3, see below)
       0 = None                (Special effects disabled)
```

```
        1 = Alpha Blending      (1st+2nd Target mixed)
        2 = Brightness Increase (1st Target becomes whiter)
        3 = Brightness Decrease (1st Target becomes blacker)
8     BG0 2nd Target Pixel (Background 0)
9     BG1 2nd Target Pixel (Background 1)
10    BG2 2nd Target Pixel (Background 2)
11    BG3 2nd Target Pixel (Background 3)
12    OBJ 2nd Target Pixel (Top-most OBJ pixel)
13    BD  2nd Target Pixel (Backdrop)
14-15 Not used
```

Selects the 1st Target layer(s) for special effects. For Alpha Blenging/Semi-Transparency, it does also select the 2nd Target layer(s), which should have next lower display priority as the 1st Target. However, any combinations are possible, including that all layers may be selected as both 1st+2nd target, in that case the top-most pixel will be used as 1st target, and the next lower pixel as 2nd target.

052h - BLDALPHA (formerly COLEV) - Alpha Blending Coefficients (W)
Used for Color Special Effects Mode 1, and for Semi-Transparent OBJs.

```
Bit   Expl.
0-4   EVA Coefficient (1st Target) (0..16 = 0/16..16/16, 17..31=16/16)
5-7   Not used
8-12  EVB Coefficient (2nd Target) (0..16 = 0/16..16/16, 17..31=16/16)
13-15 Not used
```

For this effect, the top-most non-transparent pixel must be selected as 1st Target, and the next-lower non-transparent pixel must be selected as 2nd Target, if so - and only if so, then color intensities of 1st and 2nd Target are mixed together by using the parameters in BLDALPHA register, for each pixel each R, G, B intensities are calculated separately:

```
I = MIN ( 31, I1st*EVA + I2nd*EVB )
```

Otherwise - for example, if only one target exists, or if a non-transparent non-2nd-target pixel is moved between the two targets, or if 2nd target has higher display priority than 1st target - then only the-most pixel is displayed (at normal intensity, regardless of BLDALPHA).

054h - BLDY (formerly COLY) - Brightness (Fade-In/Out) Coefficient (W)
Used for Color Special Effects Modes 2 and 3.

```
Bit   Expl.
0-4   EVY Coefficient (Brightness) (0..16 = 0/16..16/16, 17..31=16/16)
5-15  Not used
```

For each pixel each R, G, B intensities are calculated separately:

```
I = I1st + (31-I1st)*EVY   ;For Brightness Increase
I = I1st - (I1st)*EVY      ;For Brightness Decrease
```

The color intensities of any selected 1st target surface(s) are increased or decreased by using the parameter in BLDY register.

Semi-Transparent OBJs
OBJs that are defined as 'Semi-Transparent' in OAM memory are always selected as 1st Target (regardless of BLDCNT Bit 4), and are always using Alpha Blending mode (regardless of BLDCNT Bit 6-7).
The BLDCNT register may be used to perform Brightness effects on the OBJ (and/or other BG/BD layers). However, if a semi-transparent OBJ pixel does overlap a 2nd target pixel, then semi-transparency becomes priority, and the brightness effect will not take place (neither on 1st, nor 2nd target).

The OBJ Layer
Before special effects are applied, the display controller computes the OBJ priority ordering, and isolates the top-most OBJ pixel. In result, only the top-most OBJ pixel is recursed at the time when processing special effects. Ie. alpha blending and semi-transparency can be used for OBJ-to-BG or BG-to-OBJ , but not for OBJ-to-OBJ.

LCD VRAM Overview

The GBA contains 96 Kbytes VRAM built-in, located at address 06000000-06017FFF, depending on the BG Mode used as follows:

BG Mode 0,1,2 (Tile/Map based Modes)

```
06000000-0600FFFF  64 KBytes shared for BG Map and Tiles
06010000-06017FFF  32 KBytes OBJ Tiles
```

The shared 64K area will be split into BG Map area (max. 32K) and BG Tiles area (min 32K), the respective addresses for Map and Tile areas are set up by BG0CNT-BG3CNT registers. The Map address may be specified in units of 2K (steps of 800h), the Tile address in units of 16K (steps of 4000h).

BG Mode 3 (Bitmap based Mode for still images)

```
06000000-06013FFF  80 KBytes Frame 0 buffer (only 75K actually used)
06014000-06017FFF  16 KBytes OBJ Tiles
```

BG Mode 4,5 (Bitmap based Modes)

```
06000000-06009FFF  40 KBytes Frame 0 buffer (only 37.5K used in Mode 4)
0600A000-06013FFF  40 KBytes Frame 1 buffer (only 37.5K used in Mode 4)
06014000-06017FFF  16 KBytes OBJ Tiles
```

Note
Additionally to the above VRAM, the GBA also contains 1 KByte Palette RAM (at 05000000h) and 1 KByte OAM (at 07000000h) which are both used by the display controller as well.

LCD VRAM Character Data

Each character (tile) consists of 8x8 dots (64 dots in total). The color depth may be either 4bit or 8bit (see BG0CNT-BG3CNT).

4bit depth (16 colors, 16 palettes)
Each tile occupies 32 bytes of memory, the first 4 bytes for the topmost row of the tile, and so on. Each byte representing two dots, the lower 4 bits define the color for the left (!) dot, the upper 4 bits the color for the right dot.

8bit depth (256 colors, 1 palette)
Each tile occupies 64 bytes of memory, the first 8 bytes for the topmost row of the tile, and so on. Each byte selects the palette entry for each dot.

LCD VRAM BG Screen Data Format (BG Map)

The display background consists of 8x8 dot tiles, the arrangement of these tiles is specified by the BG Screen Data (BG Map). The separate entries in this map are as follows:

Text BG Screen (2 bytes per entry)
Specifies the tile number and attributes. Note that BG tile numbers are always specified in steps of 1 (unlike OBJ tile numbers which are using steps of two in 256 color/1 palette mode).

```
Bit   Expl.
0-9   Tile Number     (0-1023) (a bit less in 256 color mode, because
                         there'd be otherwise no room for the bg map)
10    Horizontal Flip (0=Normal, 1=Mirrored)
11    Vertical Flip   (0=Normal, 1=Mirrored)
12-15 Palette Number  (0-15)    (Not used in 256 color/1 palette mode)
```

A Text BG Map always consists of 32x32 entries (256x256 pixels), 400h entries = 800h bytes. However, depending on the BG Size, one, two, or four of these Maps may be used together, allowing to create backgrounds of 256x256, 512x256, 256x512, or 512x512 pixels, if so, the first map (SC0) is located at base+0, the next map (SC1) at base+800h, and so on.

Rotation/Scaling BG Screen (1 byte per entry)
In this mode, only 256 tiles can be used. There are no x/y-flip attributes, the color depth is always 256 colors/1 palette.

```
Bit   Expl.
0-7   Tile Number     (0-255)
```

The dimensions of Rotation/Scaling BG Maps depend on the BG size. For size 0-3 that are: 16x16 tiles (128x128 pixels), 32x32 tiles (256x256 pixels), 64x64 tiles (512x512 pixels), or 128x128 tiles (1024x1024 pixels).

The size and VRAM base address of the separate BG maps for BG0-3 are set up by BG0CNT-BG3CNT registers.

LCD VRAM Bitmap BG Modes

In BG Modes 3-5 the background is defined in form of a bitmap (unlike as for Tile/Map based BG modes). Bitmaps are implemented as BG2, with Rotation/Scaling support. As bitmap modes are occupying 80KBytes of BG memory, only 16KBytes of VRAM can be used for OBJ tiles.

BG Mode 3 - 240x160 pixels, 32768 colors
Two bytes are associated to each pixel, directly defining one of the 32768 colors (without using palette data, and thus not supporting a 'transparent' BG color).

```
Bit   Expl.
0-4   Red Intensity   (0-31)
5-9   Green Intensity (0-31)
10-14 Blue Intensity  (0-31)
15    Not used
```

The first 480 bytes define the topmost line, the next 480 the next line, and so on. The background occupies 75 KBytes (06000000-06012BFF), most of the 80 Kbytes BG area, not allowing to redraw an invisble second frame in background, so this mode is mostly recommended for still images only.

BG Mode 4 - 240x160 pixels, 256 colors (out of 32768 colors)
One byte is associated to each pixel, selecting one of the 256 palette entries. Color 0 (backdrop) is transparent, and OBJs may be displayed behind the bitmap.
The first 240 bytes define the topmost line, the next 240 the next line, and so on. The background occupies 37.5 KBytes, allowing two frames to be used (06000000-060095FF for Frame 0, and 0600A000-060135FF for Frame 1).

BG Mode 5 - 160x128 pixels, 32768 colors
Colors are defined as for Mode 3 (see above), but horizontal and vertical size are cut down to 160x128 pixels only - smaller than the physical dimensions of the LCD screen.
The background occupies exactly 40 KBytes, so that BG VRAM may be split into two frames (06000000-06009FFF for Frame 0, and 0600A000-06013FFF for Frame 1).

In BG modes 4,5, one Frame may be displayed (selected by DISPCNT Bit 4), the other Frame is invisible and may be redrawn in background.

LCD OBJ - Overview

General
Objects (OBJs) are moveable sprites. Up to 128 OBJs (of any size, up to 64x64 dots each) can be displayed per screen, and under best circumstances up to 128 OBJs (of small 8x8 dots size) can be displayed per horizontal display line.

Maximum Number of Sprites per Line
The total available OBJ rendering cycles per line are

```
1210   (=304*4-6)    If "H-Blank Interval Free" bit in DISPCNT register is 0
954    (=240*4-6)    If "H-Blank Interval Free" bit in DISPCNT register is 1
```

The required rendering cycles are (depending on horizontal OBJ size)

```
Cycles per Screen Pixels  OBJ Type              OBJ Type Screen Pixel Range
8 cycles per 8 pixels     Normal OBJs           8..64 pixels
26 cycles per 8 pixels    Rotation/Scaling OBJs 8..64 pixels   (area clipped)
26 cycles per 8 pixels    Rotation/Scaling OBJs 16..128 pixels (double size)
```

Caution:
The maximum number of OBJs per line is also affected by undisplayed (offscreen) OBJs which are having higher priority than displayed OBJs.
To avoid this, move displayed OBJs to the begin of OAM memory (ie. OBJ0 has highest priority, OBJ127 lowest).
Otherwise (in case that the program logic expects OBJs at fixed positions in OAM) at least take care to set the OBJ size of undisplayed OBJs to 8x8 with Rotation/Scaling disabled (this reduces the overload).
Does the above also apply for VERTICALLY OFFSCREEN (or VERTICALLY not on CURRENT LINE) sprites ???

VRAM - Character Data
OBJs are always combined of one or more 8x8 pixel Tiles (much like BG Tiles in BG Modes 0-2).
However, OBJ Tiles are stored in a separate area in VRAM: 06100000-0617FFFF (32 KBytes) in BG Mode 0-2, or 06140000-0617FFFF (16 KBytes) in BG Mode 3-5.
Depending on the size of the above area (16K or 32K), and on the OBJ color depth (4bit or 8bit), 256-1024 8x8 dots OBJ Tiles can be defined.

OAM - Object Attribute Memory
This memory area contains Attributes which specify position, size, color depth, etc. appearance for each of the 128 OBJs. Additionally, it contains 32 OBJ Rotation/Scaling Parameter groups. OAM is located at 0700:0000-0700:03FF (sized 1 KByte).

LCD OBJ - OAM Attributes

OBJ Attributes
There are 128 entries in OAM for each OBJ0-OBJ127. Each entry consists of 6 bytes (three 16bit Attributes). Attributes for OBJ0 are located at 0700:0000, for OBJ1 at 0700:0008, OBJ2 at 0700:0010, and so on.

As you can see, there are blank spaces at 0700:0006, 0700:000E, 0700:0016, etc. - these 16bit values are used for OBJ Rotation/Scaling (as described in the next chapter) - they are not directly related to the separate OBJs.

OBJ Attribute 0 (R/W)

```
Bit   Expl.
0-7   Y-Coordinate           (0-255)
8     Rotation/Scaling Flag  (0=Off, 1=On)
When Rotation/Scaling used (Attribute 0, bit 8 set):
  9     Double-Size Flag     (0=Normal, 1=Double)
When Rotation/Scaling not used (Attribute 0, bit 8 cleared):
  9     OBJ Disable          (0=Normal, 1=Not displayed)
10-11 OBJ Mode  (0=Normal, 1=Semi-Transparent, 2=OBJ Window, 3=Prohibited)
12    OBJ Mosaic             (0=Off, 1=On)
13    Colors/Palettes        (0=16/16, 1=256/1)
14-15 OBJ Shape              (0=Square,1=Horizontal,2=Vertical,3=Prohibited)
```

Caution: A very large OBJ (of 128 pixels vertically, ie. a 64 pixels OBJ in a Double Size area) located at Y>128 will be treated as at Y>-128, the OBJ is then displayed parts offscreen at the TOP of the display, it is then NOT displayed at the bottom.

OBJ Attribute 1 (R/W)

```
Bit   Expl.
0-8   X-Coordinate           (0-511)
When Rotation/Scaling used (Attribute 0, bit 8 set):
  9-13  Rotation/Scaling Parameter Selection (0-31)
        (Selects one of the 32 Rotation/Scaling Parameters that
        can be defined in OAM, for details read next chapter.)
When Rotation/Scaling not used (Attribute 0, bit 8 cleared):
  9-11  Not used
  12    Horizontal Flip      (0=Normal, 1=Mirrored)
  13    Vertical Flip        (0=Normal, 1=Mirrored)
14-15 OBJ Size               (0..3, depends on OBJ Shape, see Attr 0)
        Size  Square   Horizontal  Vertical
        0     8x8      16x8        8x16
        1     16x16    32x8        8x32
        2     32x32    32x16       16x32
        3     64x64    64x32       32x64
```

OBJ Attribute 2 (R/W)

```
Bit   Expl.
0-9   Character Name          (0-1023=Tile Number)
10-11 Priority relative to BG (0-3; 0=Highest)
12-15 Palette Number   (0-15) (Not used in 256 color/1 palette mode)
```

Notes:

OBJ Mode
The OBJ Mode may be Normal, Semi-Transparent, or OBJ Window.
Semi-Transparent means that the OBJ is used as 'Alpha Blending 1st Target' (regardless of BLDCNT register, for details see chapter about Color Special Effects).
OBJ Window means that the OBJ is not displayed, instead, dots with non-zero color are used as mask for the OBJ Window, see DISPCNT and WINOUT for details.

OBJ Tile Number
There are two situations which may divide the amount of available tiles by two (by four if both situations apply):

1. When using the 256 Colors/1 Palette mode, only each second tile may be used, the lower bit of the tile

number should be zero (in 2-dimensional mapping mode, the bit is completely ignored).

2. When using BG Mode 3-5 (Bitmap Modes), only tile numbers 512-1023 may be used. That is because lower 16K of OBJ memory are used for BG. Attempts to use tiles 0-511 are ignored (not displayed).

Priority
In case that the 'Priority relative to BG' is the same than the priority of one of the background layers, then the OBJ becomes higher priority and is displayed on top of that BG layer.
Caution: Take care not to mess up BG Priority and OBJ priority. For example, the following would cause garbage to be displayed:

```
OBJ No. 0 with Priority relative to BG=1   ;hi OBJ prio, lo BG prio
OBJ No. 1 with Priority relative to BG=0   ;lo OBJ prio, hi BG prio
```

That is, OBJ0 is always having priority above OBJ1-127, so assigning a lower BG Priority to OBJ0 than for OBJ1-127 would be a bad idea.

LCD OBJ - OAM Rotation/Scaling Parameters

As described in the previous chapter, there are blank spaces between each of the 128 OBJ Attribute Fields in OAM memory. These 128 16bit gaps are used to store OBJ Rotation/Scaling Parameters.

Location of Rotation/Scaling Parameters in OAM
Four 16bit parameters (PA,PB,PC,PD) are required to define a complete group of Rotation/Scaling data. These are spread across OAM as such:

```
1st Group - PA=0700:0006, PB=0700:000E, PC=0700:0016, PD=0700:001E
2nd Group - PA=0700:0026, PB=0700:002E, PC=0700:0036, PD=0700:003E
etc.
```

By using all blank space (128 x 16bit), up to 32 of these groups (4 x 16bit each) can be defined in OAM.

OBJ Rotation/Scaling PA,PB,PC,PD Parameters (R/W)
Each OBJ that uses Rotation/Scaling may select between any of the above 32 parameter groups. For details, refer to the previous chapter about OBJ Attributes.
The meaning of the separate PA,PB,PC,PD values is identical as for BG, for details read the chapter about BG Rotation/Scaling.

OBJ Reference Point & Rotation Center
The OBJ Reference Point is the upper left of the OBJ, ie. OBJ X/Y coordinates: X+0, Y+0.
The OBJ Rotation Center is always (or should be usually?) in the middle of the object, ie. for a 8x32 pixel OBJ, this would be at the OBJ X/Y coordinates: X+4, and Y+16.

OBJ Double-Size Bit (for OBJs that use Rotation/Scaling)
When Double-Size is zero: The sprite is rotated, and then display inside of the normal-sized (not rotated) rectangular area - the edges of the rotated sprite will become invisible if they reach outside of that area.
When Double-Size is set: The sprite is rotated, and then display inside of the double-sized (not rotated) rectangular area - this ensures that the edges of the rotated sprite remain visible even if they would reach outside of the normal-sized area. (Except that, for example, rotating a 8x32 pixel sprite by 90 degrees would still cut off parts of the sprite as the double-size area isn't large enough.)

LCD OBJ - VRAM Character (Tile) Mapping

Each OBJ tile consists of 8x8 dots, however, bigger OBJs can be displayed by combining several 8x8 tiles.

The horizontal and vertical size for each OBJ may be separately defined in OAM, possible H/V sizes are 8,16,32,64 dots - allowing 'square' OBJs to be used (such like 8x8, 16x16, etc) as well as 'rectangular' OBJs (such like 8x32, 64x16, etc.)

When displaying an OBJ that contains of more than one 8x8 tile, one of the following two mapping modes can be used. In either case, the tile number of the upperleft tile must be specified in OAM memory.

Two Dimensional Character Mapping (DISPCNT Bit 6 cleared)
This mapping mode assumes that the 1024 OBJ tiles are arranged as a matrix of 32x32 tiles / 256x256 pixels (In 256 color mode: 16x32 tiles / 128x256 pixels). Ie. the upper row of this matrix contains tiles 00h-1Fh, the next row tiles 20h-3Fh, and so on.
For example, when displaying a 16x16 pixel OBJ, with tile number set to 04h; The upper row of the OBJ will consist of tile 04h and 05h, the next row of 24h and 25h. (In 256 color mode: 04h and 06h, 24h and 26h.)

One Dimensional Character Mapping (DISPCNT Bit 6 set)
In this mode, tiles are mapped each after each other from 00h-3FFh.
Using the same example as above, the upper row of the OBJ will consist of tile 04h and 05h, the next row of tile 06h and 07h. (In 256 color mode: 04h and 06h, 08h and 0Ah.)

LCD Color Palettes

Color Palette RAM
BG and OBJ palettes are using separate memory regions:

```
05000000-050001FF - BG Palette RAM (512 bytes, 256 colors)
05000200-050003FF - OBJ Palette RAM (512 bytes, 256 colors)
```

Each BG and OBJ palette RAM may be either split into 16 palettes with 16 colors each, or may be used as a single palette with 256 colors.
Note that some OBJs may access palette RAM in 16 color mode, while other OBJs may use 256 color mode at the same time. Same for BG0-BG3 layers.

Transparent Colors
Color 0 of all BG and OBJ palettes is transparent. Even though palettes are described as 16 (256) color palettes, only 15 (255) colors are actually visible.

Backdrop Color
Color 0 of BG Palette 0 is used as backdrop color. This color is displayed if an area of the screen is not covered by any non-transparent BG or OBJ dots.

Color Definitions
Each color occupies two bytes (same as for 32768 color BG modes):

```
Bit   Expl.
0-4   Red Intensity   (0-31)
5-9   Green Intensity (0-31)
10-14 Blue Intensity  (0-31)
15    Not used
```

Intensities
Under normal circumstances (light source/viewing angle), the intensities 0-14 are practically all black, and only intensities 15-31 are resulting in visible medium..bright colors.

Note: The intensity problem appears in the 8bit CGB "compatibilty" mode either. The original CGB

display produced the opposite effect: Intensities 0-14 resulted in dark..medium colors, and intensities 15-31 resulted in bright colors. Any "medium" colors of CGB games will appear invisible/black on GBA hardware, and only very bright colors will be visible.

LCD Dimensions and Timings

Horizontal Dimensions
The drawing time for each dot is 4 CPU cycles.

```
Visible     240 dots,  57.221 us,    960 cycles - 78% of h-time
H-Blanking   68 dots,  16.212 us,    272 cycles - 22% of h-time
Total       308 dots,  73.433 us,   1232 cycles - ca. 13.620 kHz
```

VRAM and Palette RAM may be accessed during H-Blanking. OAM can accessed only if "H-Blank Interval Free" bit in DISPCNT register is set.

Vertical Dimensions

```
Visible (*) 160 lines, 11.749 ms, 197120 cycles - 70% of v-time
V-Blanking   68 lines,  4.994 ms,  83776 cycles - 30% of v-time
Total       228 lines, 16.743 ms, 280896 cycles - ca. 59.737 Hz
```

All VRAM, OAM, and Palette RAM may be accessed during V-Blanking.
Note that no H-Blank interrups are generated within V-Blank period.

System Clock
The system clock is 16.78MHz (16*1024*1024 Hz), one cycle is thus approx. 59.59ns.

(*) Even though vertical screen size is 160 lines, the upper 8 lines are not <really> visible, these lines are covered by a shadow when holding the GBA orientated towards a light source, the lines are effectively black - and should not be used to display important information.

Sound Controller

The GBA supplies four 'analogue' sound channels for Tone and Noise (mostly compatible to CGB sound), as well as two 'digital' sound channels (which can be used to replay 8bit DMA sample data).

Sound Channel 1 - Tone & Sweep
Sound Channel 2 - Tone
Sound Channel 3 - Wave Output
Sound Channel 4 - Noise
Sound Channel A and B - DMA Sound

Sound Control Registers
Comparision of CGB and GBA Sound

The GBA includes only a single (mono) speaker built-in, each channel may be output to either left and/or right channels by using the external line-out connector (for stereo headphones, etc).

Sound Channel 1 - Tone & Sweep

060h - SOUND1CNT_L (formerly SG10_L) (NR10) - Channel 1 Sweep register (R/W)

```
Bit        Expl.
0-2   R/W  Number of sweep shift      (n=0-7)
3     R/W  Sweep Frequency Direction  (0=Increase, 1=Decrease)
4-6   R/W  Sweep Time; units of 7.8ms (0-7, min=7.8ms, max=54.7ms)
7-15  -    Not used
```

Sweep is disabled by setting Sweep Time to zero, if so, the direction bit should be set.
The change of frequency (NR13,NR14) at each shift is calculated by the following formula where X(0) is initial freq & X(t-1) is last freq:

```
X(t) = X(t-1) +/- X(t-1)/2^n
```

062h - SOUND1CNT_L (SG10_H) (NR11, NR12) - Channel 1 Duty/Len/Envelope (R/W)

```
Bit        Expl.
0-5   W    Sound length; units of (64-n)/256s  (0-63)
6-7   R/W  Wave Pattern Duty                   (0-3, see below)
8-10  R/W  Envelope Step-Time; units of n/64s  (1-7, 0=No Envelope)
11    R/W  Envelope Direction                  (0=Decrease, 1=Increase)
12-15 R/W  Initial Volume of envelope          (1-15, 0=No Sound)
```

Wave Duty:

```
0: 12.5% ( -_______-_______-_______ )
1: 25%   ( --______--______--______ )
2: 50%   ( ----____----____----____ ) (normal)
3: 75%   ( ------__------__------__ )
```

The Length value is used only if Bit 6 in NR14 is set.

064h - SOUND1CNT_X (SG11) (NR13, NR14) - Channel 1 Frequency/Control (R/W)

```
Bit        Expl.
0-10  W    Frequency; 131072/(2048-n)Hz  (0-2047)
11-13 -    Not used
14    R/W  Length Flag  (1=Stop output when length in NR11 expires)
15    W    Initial      (1=Restart Sound)
```

Sound Channel 2 - Tone

This sound channel works exactly as channel 1, except that it doesn't have a Tone Envelope/Sweep Register.

066h - Not used
068h - SOUND2CNT_L (SG20) (NR21, NR22) - Channel 2 Duty/Length/Envelope (R/W)
06Ah - Not used
06Ch - SOUND2CNT_H (SG21) (NR23, NR24) - Channel 2 Frequency/Control (R/W)
06Eh - Not used
For details, refer to channel 1 description.

Sound Channel 3 - Wave Output

This channel can be used to output digital sound, the length of the sample buffer (Wave RAM) can be either 32 or 64 digits (4bit samples). This sound channel can be also used to output normal tones when initializing the Wave RAM by a square wave. This channel doesn't have a volume envelope register.

070h - SOUND3CNT_L (SG30_L) (NR30) - Channel 3 Stop/Wave RAM select (R/W)

```
Bit        Expl.
0-4   -    Not used
5     R/W  Wave RAM Bank Number (0-1, see below)
6     R/W  Wave RAM Dimension   (0=One bank/32 digits, 1=Two banks/64 digits)
7     R/W  Sound Channel 3 Off  (0=Stop, 1=Playback)
8-15  -    Not used
```

The currently selected Bank Number (Bit 5) will be played back, while reading/writing to/from wave RAM will address the other (not selected) bank. When dimension is set to two banks, output will start by replaying the currently selected bank.

072h - SOUND3CNT_H (SG30_H) (NR31, NR32) - Channel 3 Length/Volume (R/W)

```
Bit        Expl.
0-7   W    Sound length; units of (256-n)/256s  (0-255)
8-12  -    Not used.
13-14 R/W  Sound Volume  (0=Mute/Zero, 1=100%, 2=50%, 3=25%)
15    R/W  Force Volume  (0=Use above, 1=Force 75% regardless of above)
```

The Length value is used only if Bit 6 in NR34 is set.

074h - SOUND3CNT_X (SG31) (NR33, NR34) - Channel 3 Frequency/Control (R/W)

```
Bit        Expl.
0-10  W    Frequency; 131072/(2048-n) Hz   (0-2047)
11-13 -    Not used
14    R/W  Length Flag  (1=Stop output when length in NR31 expires)
15    W    Initial      (1=Restart Sound)
```

The above frequency is meant to be the sample rate per digit in wave RAM. The repeat rate for 32 digit wave RAM would be thus above frequency divided by 32. (Divided by 64 for 64 digit wave RAM).

090h - WAVE_RAM0_L (SGWR0_L) - Channel 3 Wave Pattern RAM (W/R)
092h - WAVE_RAM0_H (SGWR0_H) - Channel 3 Wave Pattern RAM (W/R)
094h - WAVE_RAM1_L (SGWR1_L) - Channel 3 Wave Pattern RAM (W/R)
096h - WAVE_RAM1_H (SGWR1_H) - Channel 3 Wave Pattern RAM (W/R)
098h - WAVE_RAM2_L (SGWR2_L) - Channel 3 Wave Pattern RAM (W/R)
09Ah - WAVE_RAM2_H (SGWR2_H) - Channel 3 Wave Pattern RAM (W/R)
09Ch - WAVE_RAM3_L (SGWR3_L) - Channel 3 Wave Pattern RAM (W/R)
09Eh - WAVE_RAM3_H (SGWR3_H) - Channel 3 Wave Pattern RAM (W/R)
This area contains 16 bytes (32 x 4bits) Wave Pattern data which is output by channel 3. Data is played back ordered as follows: MSBs of 1st byte, followed by LSBs of 1st byte, followed by MSBs of 2nd byte, and so on - this results in a confusing ordering when filling Wave RAM in units of 16bit data - ie. samples would be then located in Bits 4-7, 0-3, 12-15, 8-11.

In the GBA, two Wave Patterns exists (each 32 x 4bits), either one may be played (as selected in NR30 register), the other bank may be accessed by the users. After all 32 samples have been played, output of the same bank (or other bank, as specified in NR30) will be automatically restarted.

Internally, Wave RAM is a giant shift-register, there is no pointer which is addressing the currently played digit. Instead, the entire 128 bits are shifted, and the 4 least significant bits are output.
Thus, when reading from Wave RAM, data might have changed its postition. And, when writing to Wave RAM all data should be updated (it'd be no good idea to assume that old data is still located at the same position where it has been written to previously).

Sound Channel 4 - Noise

This channel is used to output white noise. This is done by randomly switching the amplitude between

high and low at a given frequency. Depending on the frequency the noise will appear 'harder' or 'softer'.

It is also possible to influence the function of the random generator, so the that the output becomes more regular, resulting in a limited ability to output Tone instead of Noise.

076h - Not used

078h - SOUND4CNT_L (SG40) (NR41, NR42) - Channel 4 Length/Envelope (R/W)

Bit		Expl.	
0-5	W	Sound length; units of (64-n)/256s	(0-63)
6-7	-	Not used	
8-10	R/W	Envelope Step-Time; units of n/64s	(1-7, 0=No Envelope)
11	R/W	Envelope Direction	(0=Decrease, 1=Increase)
12-15	R/W	Initial Volume of envelope	(1-15, 0=No Sound)

The Length value is used only if Bit 6 in NR44 is set.

07Ah - Not used

07Ch - SOUND4CNT_H (SG41) (NR43, NR44) - Channel 4 Frequency/Control (R/W)
The amplitude is randomly switched between high and low at the given frequency. A higher frequency will make the noise to appear 'softer'.
When Bit 3 is set, the output will become more regular, and some frequencies will sound more like Tone than Noise.

Bit		Expl.	
0-2	R/W	Dividing Ratio of Frequencies (r)	
3	R/W	Counter Step/Width (0=15 bits, 1=7 bits)	
4-7	R/W	Shift Clock Frequency (s)	
8-13	-	Not used	
14	R/W	Length Flag	(1=Stop output when length in NR41 expires)
15	W	Initial	(1=Restart Sound)

Frequency = 524288 Hz / r / 2^(s+1) ;For r=0 assume r=0.5 instead

07Eh - Not used

Sound Channel A and B - DMA Sound

The GBA contains two DMA sound channels (A and B), each allowing to replay digital sound (signed 8bit data, ie. -128..+127). Data can be transferred from INTERNAL memory (not sure if EXTERNAL memory works also ???) to FIFO by using DMA channel 1 or 2, the sample rate is generated by using one of the Timers.

0A0h - FIFO_A_L (SGFIFOA_L) - Sound A FIFO, Data 0 and Data 1 (W)
0A2h - FIFO_A_H (SGFIFOA_H) - Sound A FIFO, Data 2 and Data 3 (W)
These two registers may receive 32bit (4 bytes) of audio data (Data 0-3, Data 0 being located in least significant byte which is replayed first).
Internally, the capacity of the FIFO is 8 x 32bit (32 bytes), allowing to buffer a small amount of samples. As the name says (First In First Out), oldest data is replayed first.

0A4h - FIFO_B_L (SGFIFOB_L) - Sound B FIFO, Data 0 and Data 1 (W)
0A6h - FIFO_B_H (SGFIFOB_H) - Sound B FIFO, Data 2 and Data 3 (W)
Same as above, for Sound B.

Initializing DMA-Sound Playback
- Select Timer 0 or 1 in SGCNT0_H control register.

- Clear the FIFO.
- Manually write a sample byte to the FIFO.
- Initialize transfer mode for DMA 1 or 2.
- Initialize DMA Sound settings in sound control register.
- Start the timer.

DMA-Sound Playback Procedure
The pseudo-procedure below is automatically repeated.

```
If Timer overflows then
  Move 8bit data from FIFO to sound circuit.
  If FIFO contains only 4 x 32bits (16 bytes) then
    Request more data per DMA
    Receive 4 x 32bit (16 bytes) per DMA
  Endif
Endif
```

This playback mechanism will be repeated forever, regardless of the actual length of the sample buffer.

Synchronizing Sample Buffers
The buffer-end may be determined by counting sound Timer IRQs (each sample byte), or sound DMA IRQs (each 16th sample byte). Both methods would require a lot of CPU time (IRQ processing), and both would fail if interrupts are disabled for a longer period.
Better solutions would be to synchronize the sample rate/buffer length with V-blanks, or to use a second timer (in count up/slave mode) which produces an IRQ after the desired number of samples.

The Sample Rate
The GBA hardware does internally re-sample all sound output to 32.768kHz (default SOUNDBIAS setting). It'd thus do not make much sense to use higher DMA/Timer rates. Best re-sampling accuracy can be gained by using DMA/Timer rates of 32.768kHz, 16.384kHz, or 8.192kHz (ie. fragments of the physical output rate).

Sound Control Registers

080h - SOUNDCNT_L (SGCNT0_L) (NR50, NR51) - Channel L/R Volume/Enable (R/W)

```
Bit   Expl.
0-2   Sound 1-4 Master volume RIGHT (0-7)
3     Not used
4-6   Sound 1-4 Master Volume LEFT (0-7)
7     Not used
8-11  Sound 1-4 Enable Flags RIGHT (each Bit 8-11, 0=Disable, 1=Enable)
12-15 Sound 1-4 Enable Flags LEFT (each Bit 12-15, 0=Disable, 1=Enable)
```

082h - SOUNDCNT_H (SGCNT0_H) (GBA only) - DMA Sound Control/Mixing (R/W)

```
Bit   Expl.
0-1   Sound # 1-4 Volume   (0=25%, 1=50%, 2=100%, 3=Prohibited)
2     DMA Sound A Volume   (0=50%, 1=100%)
3     DMA Sound B Volume   (0=50%, 1=100%)
4-7   Not used
8     DMA Sound A Enable RIGHT (0=Disable, 1=Enable)
9     DMA Sound A Enable LEFT  (0=Disable, 1=Enable)
10    DMA Sound A Timer Select (0=Timer 0, 1=Timer 1)
11    DMA Sound A Reset FIFO   (1=Reset)
12    DMA Sound B Enable RIGHT (0=Disable, 1=Enable)
13    DMA Sound B Enable LEFT  (0=Disable, 1=Enable)
14    DMA Sound B Timer Select (0=Timer 0, 1=Timer 1)
15    DMA Sound B Reset FIFO   (1=Reset)
```

084h - SOUNDCNT_X (SGCNT1) (NR52) - Sound on/off (R/W)
When not using sound output, write 00h to this register to save power consumption. While Bit 7 is cleared, all other sound registers cannot be accessed, and their content must be re-initialized when re-enabling sound.

```
Bit   Expl.
0     Sound 1 ON flag (Read Only)
1     Sound 2 ON flag (Read Only)
2     Sound 3 ON flag (Read Only)
3     Sound 4 ON flag (Read Only)
4-6   Not used
7     All sound on/off  (0: stop all sound circuits) (Read/Write)
8-15  Not used
```

Bits 0-3 are automatically set when starting sound output, and are automatically cleared when a sound ends. (Ie. when the length expires, as far as length is enabled. The bits are NOT reset when an volume envelope ends.)

086h - Not used

088h - SOUNDBIAS (SG_BIAS) - Sound PWM Control (R/W, see below)
This register controls the final sound output. The default setting is 0200h, it is normally not required to change this value.

```
Bit    Expl.
0-9    Bias Level     (Default=200h, converting signed samples into unsigned)
10-13  Not used
14-15  Amplitude Resolution/Sampling Cycle (Default=0, see below)
```

Amplitude Resolution/Sampling Cycle (0-3):

```
0  9bit / 32.768kHz   (Default, best for DMA channels A,B)
1  8bit / 65.536kHz
2  7bit / 131.072kHz
3  6bit / 262.144kHz  (Best for FM channels 1-4)
```

For more information on this register, read the descriptions below.

08Ah - Not used
08Ch - Not used
08Eh - Not used

Mixing of the separate channels into 10bit
The current output levels of all six channels are added together by hardware, resulting in a signed value, typically in range -512..+511. The bias level (typically 200h = 512 decimal) is added to the result to convert it into an unsigned 10bit value, range 0..+1023. Values smaller than 0 or greater than 1023 appear to be clipped.

Resampling to 32.768kHz / 9bit (default)
The FM channels 1-4 are internally generated at 262.144kHz, and DMA sound A-B could be theoretically generated at timer rates up to 16.78MHz. However, the final sound output is resampled to a rate of 32.768kHz, at 9bit depth (the above 10bit value, divided by two). If necessary, rates higher than 32.768kHz can be selected in the SOUNDBIAS register, that would result in a depth smaller than 9bit though.

PWM (Pulse Width Modulation) Output 16.78MHz / 1bit
Okay, now comes the actual output. The GBA can output only two voltages (low and high), these 'bits' are output at system clock speed (16.78MHz). If using the default 32.768kHz sampling rate, then 512 bits are output per sample (512*32K=16M). Each sample value (9bit range, N=0..511), would be then output as N low bits, followed by 512-N high bits. The resulting 'noise' is smoothed down by capacitors, by the speaker, and by human hearing, so that it will effectively sound like clean D/A converted 9bit voltages at

32kHz sampling rate.

Changing the BIAS Level
Normally use 200h for clean sound output. A value of 000h might make sense during periods when no sound is output (causing the PWM circuit to output low-bits only, which is eventually reducing the power consumption, and/or preventing 32KHz noise). Note: Using the SoundBias function (SWI 25) allows to change the level by slowly incrementing or decrementing it (without hard scratch noise).

Comparision of CGB and GBA Sound

The GBA sound controller is mostly the same than that of older monochrome gameboy and CGB. The following changes have been done:

New Sound Channels
Two new sound channels have been added that may be used to replay 8bit digital sound. Sample rate and sample data must be supplied by using a Timer and a DMA channel.

New Control Registers
The SGCNT0_H register controls the new DMA channels - as well as mixing with the four old channels. The SOUNDBIAS register controls the final sound output.

Sound Channel 3 Changes
The length of the Wave RAM is doubled by dividing it into two banks of 32 digits each, either one or both banks may be replayed (one after each other), for details check NR30 Bit 5-6. Optionally, the sound may be output at 75% volume, for details check NR32 Bit 7.

Changed Control Registers
NR50 is not supporting Vin signals (that's been an external sound input from cartridge).

Changed I/O Addresses
The GBAs sound register are located at 0400:0060-0400:00AE instead of at FF10-FF3F as in CGB and monochrome gameboy. However, note that there have been new blank spaces inserted between some of the separate registers - therfore it is NOT possible to port CGB software to GBA just by changing the sound base address.

Accessing I/O Registers
In some cases two of the old 8bit registers are packed into a 16bit register and may be accessed as such.

Timers

The GBA includes four incrementing 16bit timers.
Timer 0 and 1 can be used to supply the sample rate for DMA sound channel A and/or B.

100h - TM0CNT_L (formerly TM0D) - Timer 0 Counter/Reload (R/W)
104h - TM1CNT_L (formerly TM1D) - Timer 1 Counter/Reload (R/W)
108h - TM2CNT_L (formerly TM2D) - Timer 2 Counter/Reload (R/W)
10Ch - TM3CNT_L (formerly TM3D) - Timer 3 Counter/Reload (R/W)
Writing to these registers intializes the <reload> value (but does not directly affect the current counter value). Reading returns the current <counter> value (or the recent/frozen counter value if the timer has been stopped).
The reload value is copied into the counter only upon following two situations: Automatically upon timer

overflows, or when the timer start bit becomes changed from 0 to 1.
Note: When simultaneously changing the start bit from 0 to 1, and setting the reload value at the same time (by a single 32bit I/O operation), then the newly written reload value is recognized as new counter value.

102h - TM0CNT_H (formerly TM0CNT) - Timer 0 Control (R/W)
106h - TM1CNT_H (formerly TM1CNT) - Timer 1 Control (R/W)
10Ah - TM2CNT_H (formerly TM2CNT) - Timer 2 Control (R/W)
10Eh - TM3CNT_H (formerly TM3CNT) - Timer 3 Control (R/W)

```
Bit   Expl.
0-1   Prescaler Selection (0=F/1, 1=F/64, 2=F/256, 3=F/1024)
2     Count-up Timing   (0=Normal, 1=See below)
3-5   Not used
6     Timer IRQ Enable  (0=Disable, 1=IRQ on Timer overflow)
7     Timer Start/Stop  (0=Stop, 1=Operate)
8-15  Not used
```

When Count-up Timing is enabled, the prescaler value is ignored, instead the time is incremented each time when the previous counter overflows. This function cannot be used for Timer 0 (as it is the first timer).
F = System Clock (16.78MHz).

DMA Transfers

Overview
The GBA includes four DMA channels, the highest priority is assigned to DMA0, followed by DMA1, DMA2, and DMA3. DMA Channels with lower priority are paused until channels with higher priority have completed.
The CPU is paused when DMA transfers are active, however, the CPU is operating during the periods when Sound/Blanking DMA transfers are paused.

Special features of the separate DMA channels
DMA0 - highest priority, best for timing critcal transfers (eg. HBlank DMA).
DMA1 and DMA2 - can be used to feed digital sample data to the Sound FIFOs.
DMA3 - can be used to write to Game Pak ROM/FlashROM (but not GamePak SRAM).
Beside for that, each DMA 0-3 may be used for whatever general purposes.

0B0h,0B2h - DMA0SAD - DMA 0 Source Address (W) (internal memory)
0BCh,0BEh - DMA1SAD - DMA 1 Source Address (W) (any memory)
0C8h,0CAh - DMA2SAD - DMA 2 Source Address (W) (any memory)
0D4h,0D6h - DMA3SAD - DMA 3 Source Address (W) (any memory)
The most significant address bits are ignored, only the least significant 27 or 28 bits are used (max 07FFFFFFh internal memory, or max 0FFFFFFFh any memory - except SRAM ???!).

0B4h,0B6h - DMA0DAD - DMA 0 Destination Address (W) (internal memory)
0C0h,0C2h - DMA1DAD - DMA 1 Destination Address (W) (internal memory)
0CCh,0CEh - DMA2DAD - DMA 2 Destination Address (W) (internal memory)
0D8h,0DAh - DMA3DAD - DMA 3 Destination Address (W) (any memory)
The most significant address bits are ignored, only the least significant 27 or 28 bits are used (max. 07FFFFFFh internal memory or 0FFFFFFFh any memory - except SRAM ???!).

0B8h - DMA0CNT_L - DMA 0 Word Count (W) (14 bit, 1..4000h)
0C4h - DMA1CNT_L - DMA 1 Word Count (W) (14 bit, 1..4000h)
0D0h - DMA2CNT_L - DMA 2 Word Count (W) (14 bit, 1..4000h)
0DCh - DMA3CNT_L - DMA 3 Word Count (W) (16 bit, 1..10000h)

Specifies the number of data units to be transferred, each unit is 16bit or 32bit depending on the transfer type, a value of zero is treated as max length (ie. 4000h, or 10000h for DMA3).

0BAh - DMA0CNT_H - DMA 0 Control (R/W)
0C6h - DMA1CNT_H - DMA 1 Control (R/W)
0D2h - DMA2CNT_H - DMA 2 Control (R/W)
0DEh - DMA3CNT_H - DMA 3 Control (R/W)

```
Bit   Expl.
0-4   Not used
5-6   Dest Addr Control  (0=Increment,1=Decrement,2=Fixed,3=Increment/Reload)
7-8   Source Adr Control (0=Increment,1=Decrement,2=Fixed,3=Prohibited)
9     DMA Repeat                   (0=Off, 1=On) (Must be zero if Bit 11 set)
10    DMA Transfer Type            (0=16bit, 1=32bit)
11    Game Pak DRQ  - DMA3 only -  (0=Normal, 1=DRQ <from> Game Pak, DMA3)
12-13 DMA Start Timing  (0=Immediately, 1=VBlank, 2=HBlank, 3=Special)
        The 'Special' setting (Start Timing=3) depends on the DMA channel:
        DMA0=Prohibited, DMA1/DMA2=Sound FIFO, DMA3=Video Capture
14    IRQ upon end of Word Count   (0=Disable, 1=Enable)
15    DMA Enable                   (0=Off, 1=On)
```

After changing the Enable bit from 0 to 1, wait 2 clock cycles before accessing any DMA related registers.

When accesing OAM (7000000h) or OBJ VRAM (6010000h) by HBlank Timing, then the "H-Blank Interval Free" bit in DISPCNT register must be set.

Source and Destination Address and Word Count Registers
The SAD, DAD, and CNT_L registers are holding the initial start addresses, and initial length. The hardware does NOT change the content of these registers during or after the transfer.
The actual transfer takes place by using internal pointer/counter registers. The initial values are copied into internal regs under the following circumstances:
Upon DMA Enable (Bit 15) changing from 0 to 1: Reloads SAD, DAD, CNT_L.
Upon Repeat: Reloads CNT_L, and optionally DAD (Increment+Reload).

DMA Repeat bit
If the Repeat bit is cleared: The Enable bit is automatically cleared after the specified number of data units has been transferred.
If the Repeat bit is set: The Enable bit remains set after the transfer, and the transfer will be restarted each time when the Start condition (eg. HBlank, Fifo) becomes true. The specified number of data units is transferred <each> time when the transfer is (re-)started. The transfer will be repeated forever, until it gets stopped by software.

Sound DMA (FIFO Timing Mode) (DMA1 and DMA2 only)
In this mode, the DMA Repeat bit must be set, and the destination address must be FIFO_A (040000A0h) or FIFO_B (040000A4h).
Upon DMA request from sound controller, 4 units of 32bits (16 bytes) are transferred (both Word Count register and DMA Transfer Type bit are ignored). The destination address will not be incremented in FIFO mode.
Keep in mind that DMA channels of higher priority may offhold sound DMA. For example, when using a 64 kHz sample rate, 16 bytes of sound DMA data are requested each 0.25ms (4 kHz), at this time another 16 bytes are still in the FIFO so that there's still 0.25ms time to satisfy the DMA request. Thus DMAs with higher priority should not be operated for longer than 0.25ms. (This problem does not arise for HBlank transfers as HBlank time is limited to 16.212us.)

Game Pak DMA
Only DMA 4 may be used to transfer data to/from Game Pak ROM or Flash ROM - it cannot access Game Pak SRAM though (as SRAM data bus is limited to 8bit units). In normal mode, DMA is requested

as long until Word Count becomes zero. When setting the 'Game Pack DRQ' bit, then the cartridge must contain an external circuit which outputs a /DREQ signal. Note that there is only one pin for /DREQ and /IREQ, thus the cartridge may not supply /IREQs while using DRQ mode.

Video Capture Mode (DMA3 only)
Intended to copy a bitmap from memory (or from external hardware/camera) to VRAM. When using this transfer mode, set the repeat bit, and write the number of data units (per scanline) to the word count register. Capture works similiar like HBlank DMA, however, the transfer is started when VCOUNT=2, it is then repeated each scanline, and it gets stopped when VCOUNT=162.

Transfer End
The DMA Enable flag (Bit 15) is automatically cleared upon completion of the transfer. The user may also clear this bit manually in order to stop the transfer (obviously this is possible for Sound/Blanking DMAs only, in all other cases the CPU is stopped until the transfer completes by itself).

Transfer rate/timing ???
DMA lockup when stopping while starting ???
Capture delayed, Capture Enable=AutoCleared ???

Communication Ports

The GBAs Serial Port may be used in various different communication modes. Normal mode may exchange data between two GBAs (or to transfer data from master GBA to several slave GBAs in one-way direction).
Multi-player mode may exchange data between up to four GBAs. UART mode works much like a RS232 interface. JOY Bus mode uses a standarized Nintendo protocol. And General Purpose mode allows to mis-use the 'serial' port as bi-directional 4bit parallel port.

SIO Normal Mode
SIO Multi-Player Mode
SIO UART Mode
SIO JOY BUS Mode
SIO General-Purpose Mode

Infrared Communication Adapters
Even though early GBA prototypes have been indended to support IR communication, this feature has been removed.
However, Nintendo is apparently considering to provide an external IR adapter (to be connected to the SIO connector, being accessed in General Purpose mode).
Also, it'd be theoretically possible to include IR ports built-in in game cartridges (as done for some older 8bit/monochrome Hudson games).

SIO Normal Mode

This mode is used to communicate between two units.
Transfer rates of 256KBit/s or 2MBit/s can be selected, however, the fast 2MBit/s is intended ONLY for special hardware expansions that are DIRECTLY connected to the GBA link port (ie. without a cable being located between the GBA and expansion hardware). In normal cases, always use 256KBit/s transfer rate which provides stable results.
Transfer lengths of 8bit or 32bit may be used, the 8bit mode is the same as for older DMG/CGB gameboys, however, the voltages for "GBA cartridges in GBAs" are different as for "GMG/CGB

cartridges in DMG/CGB/GBAs", ie. it is not possible to communicate between DMG/CGB games and GBA games.

134h - RCNT (R) - Mode Selction, in Normal/Multiplayer/UART modes (R/W)

```
Bit   Expl.
0-14  Not used
15    Must be zero (0) for Normal/Multiplayer/UART modes
```

128h - SIOCNT (SCCNT_L) - SIO Control, usage in NORMAL Mode (R/W)

```
Bit   Expl.
0     Shift Clock (SC)         (0=External, 1=Internal)
1     Internal Shift Clock     (0=256KHz, 1=2MHz)
2     SI State (opponents SO)  (0=Low, 1=High/None) --- (Read Only)
3     SO during inactivity     (0=Low, 1=High)
4-6   Not used
7     Start Bit                (0=Inactive/Ready, 1=Start/Active)
8-11  Not used
12    Transfer Length          (0=8bit, 1=32bit)
13    Must be "0" for Normal Mode
14    IRQ Enable               (0=Disable, 1=Want IRQ upon completion)
15    Not used
```

The Start bit is automatically reset when the transfer completes, ie. when all 8 or 32 bits are transferred, at that time an IRQ may be generated.

12Ah - SIODATA8 (SCCNT_H) - SIO Normal Communication 8bit Data (R/W)
For 8bit normal mode. Contains 8bit data (only 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 8bit value.

120h - SIODATA32_L (SCD0) - SIO Normal Communication lower 16bit data (R/W)
122h - SIODATA32_H (SCD1) - SIO Normal Communication upper 16bit data (R/W)
Same as above SIODATA8, for 32bit normal transfer mode respectively.

Initialization
First, initialze RCNT register. Second, set mode/clock bits in SIOCNT with startbit cleared. For master: select internal clock, and (in most cases) specify 256KHz as transfer rate. For slave: select external clock, the local transfer rate selection is then ignored, as the transfer rate is supplied by the remote GBA (or other computer, which might supply custom transfer rates).
Third, set the startbit in SIOCNT with mode/clock bits unchanged.

Synchronization
The SI and SO Bits in control register may be optionally used to determine whether the opponent is ready for starting a transmission (the actual transmission is then automatically synchronized by the shift clock signal).
Bit 2 (SI) always reflects the current SI state (ie. the opponents SO state). Obviously, Bit 3 (SO) is output to SO during transfer inactivity only.
Note that only GBA models support SI and SO synchronization bits - these bits cannot be used when communicating with CGBs or monochrome gameboys.

Recommended Communication Procedure for SLAVE unit (external clock)
- Initialize data which is to be sent to master.
- Set Start flag.
- 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.
- Repeat procedure if more data is to be transferred.
(or is so=high done automatically ??? would be fine - more stable - otherwise master may still need delay)

Recommended Communication Procedure for MASTER unit (internal clock)
- Initialize data which is to be sent to slave.
- 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.
- Repeat procedure if more data is to be transferred.

Cable Protocol
During inactive transfer, the shift clock (SC) is high. The transmit (SO) and receive (SI) data lines may be manually controlled as described above.
When master sends SC=low, each master and slave must output the next outgoing data bit to SO. When master sends SC=HIGH, each master and slave must read out the opponents data bit from SI. This is repeated for each of the 8 or 32 bits, and when completed SC will be kept high again.

Transfer Rates
Either 256KHz or 2MHz rates can be selected for SC, so max 32KBytes (256KBit) or 128KBytes (2MBit) can be transferred per second. However, the software must process each 8bit or 32bit of transmitted data separately, so the actual transfer rate will be reduced by the time spent on handling each data unit.
Only 256KHz provides stable results in most cases (such like when linking between two GBAs). The 2MHz rate is intended for special expansion hardware only.

Using Normal mode for One-Way Multiplayer communication
Whem more than two GBAs are connected, data isn't exchanged between first and second GBA as usually. Instead, data is rotated from first to last GBA (and then back to first ???).
This behaviour may be used for fast one-way data transfer from master (or childs ???) to all other GBAs.
For example (3 GBAs linked):

```
Step         Sender      1st Recepient   2nd Recipient
Transfer 1:  DATA #0 --> UNDEF      -->  UNDEF     -->
Transfer 2:  DATA #1 --> DATA #0    -->  UNDEF     -->
Transfer 3:  DATA #2 --> DATA #1    -->  DATA #0   -->
Transfer 4:  DATA #3 --> DATA #2    -->  DATA #1   -->
```

The recepients should not output any own data, instead they should forward the previously received data to the next reciepint during next transfer (just keep the incoming data unmodified in the data register).
Due to the delayed forwarding, 2nd recepient should ignore the first incoming data. After the last transfer, the sender must send one (or more) dummy data unit(s), so that the last data is forwarded to the 2nd (or further) recepient(s).

SIO Multi-Player Mode

Multi-Player mode can be used to communicate between up to 4 units.

134h - RCNT (R) - Mode Selction, in Normal/Multiplayer/UART modes (R/W)

```
Bit   Expl.
0-14  Not used
15    Must be zero (0) for Normal/Multiplayer/UART modes
```

128h - SIOCNT (SCCNT_L) - SIO Control, usage in MULTI-PLAYER Mode (R/W)

```
Bit   Expl.
0-1   Baud Rate     (0-3: 9600,38400,57600,115200 bps)
2     SI-Terminal   (0=Parent, 1=Child)                  (Read Only)
3     SD-Terminal   (0=Bad connection, 1=All GBAs Ready) (Read Only)
4-5   Multi-Player ID     (0=Parent, 1-3=1st-3rd child)  (Read Only?)
6     Multi-Player Error  (0=Normal, 1=Error)            (Read Only?)
7     Start/Busy Bit      (0=Inactive, 1=Start/Busy) (Read Only for Slaves)
8-11  Not used
12    Must be "0" for Multi-Player mode
13    Must be "1" for Multi-Player mode
14    IRQ Enable          (0=Disable, 1=Want IRQ upon completion)
15    Not used
```

The ID Bits are undefined until the first transfer has completed.

12Ah - SIOMLT_SEND (SCCNT_H) - Data Send Register (R/W)
Outgoing data (16 bit) which is to be sent to the other GBAs.

120h - SIOMULTI0 (SCD0) - SIO Multi-Player Data 0 (Parent) (R/W)
122h - SIOMULTI1 (SCD1) - SIO Multi-Player Data 1 (1st child) (R/W)
124h - SIOMULTI2 (SCD2) - SIO Multi-Player Data 2 (2nd child) (R/W)
126h - SIOMULTI3 (SCD3) - SIO Multi-Player Data 3 (3rd child) (R/W)
These registers are automatically reset to FFFFh upon transfer start.
After transfer, these registers contain incoming data (16bit each) from all remote GBAs (if any / otherwise still FFFFh), as well as the local outgoing SIOMLT_SEND data.
Ie. after the transfer, all connected GBAs will contain the same values in their SIOMULTI0-3 registers.

Initialization
- Initialize RCNT Bit 14-15 and SIOCNT Bit 12-13 to select Multi-Player mode.
- Read SIOCNT Bit 3 to verify that all GBAs are in Multi-Player mode.
- Read SIOCNT Bit 2 to detect whether this is the Parent/Master unit.

Recommended Transmission Procedure
- Write outgoing data to SIODATA_SEND.
- Master must set Start bit.
- All units must process received data in SIOMULTI0-3 when transfer completed.
- After the first succesful transfer, ID Bits in SIOCNT are valid.
- If more data is to be transferred, repeat procedure.

The parent unit blindly sends data regardless of whether childs have already processed old data/supplied new data. So, parent unit might be required to insert delays between each transfer, and/or perform error checking.
Also, slave units may signalize that they are not ready by temporarily switching into another communication mode (which does not output SD High, as Multi-Player mode does during inactivity).

Transfer Protocol
Beginning
- The masters SI pin is always LOW.
- When all GBAs are in Multiplayer mode (ready) SD is HIGH.
- When master starts the transfer, it sets SC=LOW, slaves receive Busy bit.

Step A
- ID Bits in master unit are set to 0.
- Master outputs Startbit (LOW), 16bit Data, Stopbit (HIGH) through SD.
- This data is written to SIOMULTI0 of all GBAs (including master).
- Master forwards LOW from its SO to 1st childs SI.
- Transfer ends if next child does not output data after certain time.

Step B
- ID Bits in 1st child unit are set to 1.

- 1st Child outputs Startbit (LOW), 16bit Data, Stopbit (HIGH) through SD.
- This data is written to SIOMULTI1 of all GBAs (including 1st child).
- 1st child forwards LOW from its SO to 2nd childs SI.
- Transfer ends if next child does not output data after certain time.

Step C
- ID Bits in 2nd child unit are set to 2.
- 2nd Child outputs Startbit (LOW), 16bit Data, Stopbit (HIGH) through SD.
- This data is written to SIOMULTI2 of all GBAs (including 2nd child).
- 2nd child forwards LOW from its SO to 3rd childs SI.
- Transfer ends if next child does not output data after certain time.

Step D
- ID Bits in 3rd child unit are set to 3.
- 3rd Child outputs Startbit (LOW), 16bit Data, Stopbit (HIGH) through SD.
- This data is written to SIOMULTI3 of all GBAs (including 3rd child).
- Transfer ends (this was the last child).

Transfer end
- Master sets SC=HIGH, all GBAs set SO=HIGH.
- The Start/Busy bits of all GBAs are automatically cleared.
- Interrupts are requested in all GBAs (as far as enabled).

Error Bit
This bit is set when a slave did not receive SI=LOW even though SC=LOW signlized a transfer (this might happen when connecting more than 4 GBAs, or when the previous child is not connected). Also, the bit is set when a Stopbit wasn't HIGH.
The error bit may be undefined during active transfer - read only after transfer completion (the transfer continues and completes as normal even if errors have occured for some or all GBAs).
Don't know: The bit is automatically reset/inititalized with each transfer, or must be manually reset ???

Transmission Time
The transmission time depends on the selected Baud rate. And on the amount of Bits (16 data bits plus start/stop bits for each GBA), delays between each GBA, plus final timeout (if less than 4 GBAs). That is, depending on the number of connected GBAs:

GBAs	Bits	Delays	Timeout
1	18	None	Yes
2	36	1	Yes
3	54	2	Yes
4	72	3	None

(The average Delay and Timeout periods are unknown ???)
Above is not counting the additional CPU time that must be spent on initiating and processing each transfer.

Fast One-Way Transmission
Beside for the actual SIO Multiplayer mode, you could also use SIO Normal mode for fast one-way data transfer from Master unit to all Child unit(s). See chapter about SIO Normal mode for details.

SIO UART Mode

This mode works much like a RS232 port, however, the voltages are unknown, probably 0/3V rather than +/-12V ???. SI and SO are data lines (with crossed wires), SC and SD signalize Clear to Send (with crossed wires also, which requires special cable when linking between two GBAs ???)

134h - RCNT (R) - Mode Selction, in Normal/Multiplayer/UART modes (R/W)

Bit	Expl.

```
0-14  Not used
15    Must be zero (0) for Normal/Multiplayer/UART modes
```

128h - SCCNT_L - SIO Control, usage in UART Mode (R/W)

```
Bit   Expl.
0-1   Baud Rate  (0-3: 9600,38400,57600,115200 bps)
2     CTS Flag   (0=Send always/blindly, 1=Send only when SC=LOW)
3     Parity Control (0=Even, 1=Odd)
4     Send Data Flag      (0=Not Full,  1=Full)     (Read Only)
5     Receive Data Flag   (0=Not Empty, 1=Empty)    (Read Only)
6     Error Flag          (0=No Error,  1=Error)    (Read Only)
7     Data Length         (0=7bits,   1=8bits)
8     FIFO Enable Flag    (0=Disable, 1=Enable)
9     Parity Enable Flag  (0=Disable, 1=Enable)
10    Send Enable Flag    (0=Disable, 1=Enable)
11    Receive Enable Flag (0=Disable, 1=Enable)
12    Must be "1" for UART mode
13    Must be "1" for UART mode
14    IRQ Enable          (0=Disable, 1=IRQ when any Bit 4/5/6 become set)
15    Not used
```

12Ah - SIODATA8 (SCCNT_H) - usage in UART Mode (R/W)

Addresses the send/receive shift register, or (when FIFO is used) the send/receive FIFO. In either case only the lower 8bit of SIODATA8 are used, the upper 8bit are not used.
The send/receive FIFO may store up to four 8bit data units each. For example, while 1 unit is still 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.

Send/Receive Enable, CTS Feedback

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.

Error Flag

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 SIOCNT 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!

SIO JOY BUS Mode

This communication mode uses Nintendo's standarized JOY Bus protocol. When using this communication mode, the GBA is always operated as SLAVE!

In this mode, SI and SO pins are data lines (apparently synchronized by Start/Stop bits ???), SC and SD are set to low (including during active transfer ???), the transfer rate is unknown ???

134h - RCNT (R) - Mode Selction, in JOY BUS mode (R/W)

```
Bit   Expl.
```

```
0-14  Not used
14    Must be "1" for JOY BUS Mode
15    Must be "1" for JOY BUS Mode
```

128h - SIOCNT - SIO Control, not used in JOY BUS Mode

This register is not used in JOY BUS mode.

140h - JOYCNT (HS_CTRL) - JOY BUS Control Register (R/W)

```
Bit   Expl.
0     Device Reset Flag     (Command FFh)          (Read/Acknowledge)
1     Receive Complete Flag (Command 14h or 15h?)  (Read/Acknowledge)
2     Send Complete Flag    (Command 15h or 14h?)  (Read/Acknowledge)
3-5   Not used
6     IRQ when receiving a Device Reset Command  (0=Disable, 1=Enable)
7-15  Not used
```

Bit 0-2 are woring much like the bits in the IF register: Write a "1" bit to reset (acknowledge) the respective bit.
UNCLEAR: Interrupts can be requested for Send/Receive commands also ???

150h - JOY_RECV_L (JOYRE_L) - Receive Data Register low (R/W)
152h - JOY_RECV_H (JOYRE_H) - Receive Data Register high (R/W)
154h - JOY_TRANS_L (JOYTR_L) - Send Data Register low (R/W)
156h - JOY_TRANS_H (JOYTR_H) - Send Data Register high (R/W)

Send/receive data registers.

158h - JOYSTAT (JSTAT) - Receive Status Register (R/W)

```
Bit   Expl.
0     Not used
1     Receive Status Flag   (0=Remote GBA is/was receiving) (Read Only?)
2     Not used
3     Send Status Flag      (1=Remote GBA is/was sending)   (Read Only?)
4-5   General Purpose Flag  (Not assigned, may be used for whatever purpose)
6-15  Not used
```

Bit 1 is automatically set when writing to local JOY_TRANS.
Bit 3 is automatically reset when reading from local JOY_RECV.

Below are the four possible commands which can be received by the GBA. Note that the GBA (slave) cannot send any commands itself, all it can do is to read incoming data, and to provide 'reply' data which may (or may not) be read out by the master unit.

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)
```

SIO General-Purpose Mode

In this mode, the SIO is 'misused' as a 4bit bi-directional parallel port, each of the SI,SO,SC,SD pins may be directly controlled, each can be separately declared as input (with internal pull-up) or as output signal.

134h - RCNT (R) - SIO Mode, usage in GENERAL-PURPOSE Mode (R/W)
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.

```
Bit   Expl.
0     SC Data Bit          (0=Low, 1=High)
1     SD Data Bit          (0=Low, 1=High)
2     SI Data Bit          (0=Low, 1=High)
3     SO Data Bit          (0=Low, 1=High)
4     SC Direction         (0=Input, 1=Output)
5     SD Direction         (0=Input, 1=Output)
6     SI Direction         (0=Input, 1=Output, but see below)
7     SO Direction         (0=Input, 1=Output)
8     Interrupt Request    (0=Disable, 1=Enable)
9-13  Not used
14    Must be "0" for General-Purpose Mode
15    Must be "1" for General-Purpose or JOYBUS Mode
```

SI should be always used as Input to avoid problems with other hardware which does not expect data to be output there.

128h - SIOCNT - SIO Control, not used in GENERAL-PURPOSE Mode
This register is not used in general purpose mode.

Infrared Communication

Early GBA prototypes have been intended to include a built-in IR port for sending and receiving IR signals. Among others, this port could have been used to communicate with other GBAs, or older CGB models, or TV Remote Controls, etc.

[THE INFRARED COMMUNICATION FEATURE IS -NOT- SUPPORTED ANYMORE]
Anyways, the prototype specifications have been as shown below...

Keep in mind that the IR signal may be interrupted by whatever objects moved between sender and receiver - the IR port isn't recommended for programs that require realtime data exchange (such like action games).

136h - IR - Infrared Register (R/W)

```
Bit   Expl.
0     Transmission Data  (0=LED Off, 1=LED On)
1     READ Enable        (0=Disable, 1=Enable)
2     Reception Data     (0=None, 1=Signal received) (Read only)
3     AMP Operation      (0=Off, 1=On)
4     IRQ Enable Flag    (0=Disable, 1=Enable)
5-15  Not used
```

When IRQ is enabled, an interrupt is requested if the incoming signal was 0.119us Off (2 cycles), followed by 0.536us On (9 cycles) - minimum timing periods each.

Transmission Notes
When transmitting an IR signal, note that it'd be not a good idea to keep the LED turned On for a very long period (such like sending a 1 second synchronization pulse). The recipient's circuit would treat such a long signal as "normal IR pollition which is in the air" after a while, and thus ignore the signal.

Reception Notes
Received data is internally latched. Latched data may be read out by setting both READ and AMP bits. Note: Provided that you don't want to receive your own IR signal, be sure to set Bit 0 to zero before attempting to receive data.

Power-consumption
After using the IR port, be sure to reset the register to zero in order to reduce battery power consumption.

Keypad Input

The built-in GBA gamepad has 4 direction keys, and 6 buttons.

130h - KEYINPUT (formerly P1) - Key Status (R)

```
Bit   Expl.
0     Button A         (0=Pressed, 1=Released)
1     Button B         (etc.)
2     Select           (etc.)
3     Start            (etc.)
4     Right            (etc.)
5     Left             (etc.)
6     Up               (etc.)
7     Down             (etc.)
8     Button R         (etc.)
9     Button L         (etc.)
10-15 Not used
```

It'd be usually recommended to read-out this register only once per frame, and to store the current state in memory. As a side effect, this method avoids problems caused by switch bounce when a key is newly released or pressed.

132h - KEYCNT (formerly P1CNT) - Key Interrupt Control (R/W)

```
Bit   Expl.
0     Button A         (0=Ignore, 1=Select)
1     Button B         (etc.)
2     Select           (etc.)
3     Start            (etc.)
4     Right            (etc.)
5     Left             (etc.)
6     Up               (etc.)
7     Down             (etc.)
8     Button R         (etc.)
```

```
9     Button L        (etc.)
10-13 Not used
14    IRQ Enable Flag (0=Disable, 1=Enable)
15    IRQ Condition   (0=Logical OR, 1=Logical AND)
```

In logical OR mode, an interrupt is requested when ANY of the selected buttons is pressed.
In logical AND mode, an interrupt is requested when ALL of the selected buttons are pressed.

Interrupt Control

208h - IME - Interrupt Master Enable Register (R/W)

```
Bit   Expl.
0     Disable all interrupts         (0=Disable All, 1=See IE register)
1-15  Not used
```

200h - IE - Interrupt Enable Register (R/W)

```
Bit   Expl.
0     LCD V-Blank                    (0=Disable)
1     LCD H-Blank                    (etc.)
2     LCD V-Counter Match            (etc.)
3     Timer 0 Overflow               (etc.)
4     Timer 1 Overflow               (etc.)
5     Timer 2 Overflow               (etc.)
6     Timer 3 Overflow               (etc.)
7     Serial Communication           (etc.)
8     DMA 0                          (etc.)
9     DMA 1                          (etc.)
10    DMA 2                          (etc.)
11    DMA 3                          (etc.)
12    Keypad                         (etc.)
13    Game Pak (external IRQ source) (etc.)
14-15 Not used
```

Note that there is another 'master enable flag' directly in the CPUs Status Register (CPSR) accessable in privileged modes, see CPU reference for details.

202h - IF - Interrupt Request Flags / IRQ Acknowledge (R/W, see below)

```
Bit   Expl.
0     LCD V-Blank                    (1=Request Interrupt)
1     LCD H-Blank                    (etc.)
2     LCD V-Counter Match            (etc.)
3     Timer 0 Overflow               (etc.)
4     Timer 1 Overflow               (etc.)
5     Timer 2 Overflow               (etc.)
6     Timer 3 Overflow               (etc.)
7     Serial Communication           (etc.)
8     DMA 0                          (etc.)
9     DMA 1                          (etc.)
10    DMA 2                          (etc.)
11    DMA 3                          (etc.)
12    Keypad                         (etc.)
13    Game Pak (external IRQ source) (etc.)
14-15 Not used
```

Interrupts must be manually acknowledged by writing a "1" to one of the IRQ bits, the IRQ bit will then be cleared.

"[Cautions regarding clearing IME and IE]
A corresponding interrupt could occur even while a command to clear IME or each flag of the IE register is being executed. When clearing a flag of IE, you need to clear IME in advance so that mismatching of

interrupt checks will not occur." ???

"[When multiple interrupts are used]
When the timing of clearing of IME and the timing of an interrupt agree, multiple interrupts will not occur during that interrupt. Therefore, set (enable) IME after saving IME during the interrupt routine." ???

BIOS Interrupt handling
Upon interrupt execution, the CPU is switched into IRQ mode, and the physical interrupt vector is called - as this address is located in BIOS ROM, the BIOS will always always execute the following code before it forwards control to the user handler:

```
00000018  b      128h                ;IRQ vector: jump to actual BIOS handler
00000128  stmfd  r13!,r0-r3,r12,r14  ;save registers to SP_irq
0000012C  mov    r0,4000000h         ;ptr+4 to 03FFFFFC (mirror of 03007FFC)
00000130  add    r14,r15,0h          ;retadr for USER handler $+8=138h
00000134  ldr    r15,[r0,-4h]        ;jump to [03FFFFFC] USER handler
00000138  ldmfd  r13!,r0-r3,r12,r14  ;restore registers from SP_irq
0000013C  subs   r15,r14,4h          ;return from IRQ (PC=LR-4, CPSR=SPSR)
```

As shown above, a pointer to the 32bit/ARM-code user handler must be setup in [03007FFCh]. By default, 160 bytes of memory are reserved for interrupt stack at 03007F00h-03007F9Fh.

Recommended User Interrupt handling
- If necessary switch to THUMB state manually (handler is called in ARM state)
- Determine reason(s) of interrupt by examining IF register
- User program may freely assign priority to each reason by own logic
- Process the most important reason of your choice
- User MUST manually acknowledge by writing to IF register
- If user wants to allow nested interrupts, save SPSR_irq, then enable IRQs.
- If using other registers than BIOS-pushed R0-R3, manually save R4-R11 also.
- Note that Interrupt Stack is used (which may have limited size)
- So, for memory consuming stack operations use system mode (=user stack).
- When calling subroutines in system mode, save LSR_usr also.
- Restore SPSR_irq and/or R4-R11 if you've saved them above.
- Finally, return to BIOS handler by BX LR (R14_irq) instruction.

Default memory usuage at 03007FXX (and mirrored to 03FFFFXX)

```
Addr. Size Expl.
7FFCh 4    Pointer to user IRQ handler (32bit ARM code)
7FF8h 4    Interrupt Check Flag (for IntrWait/VBlankIntrWait functions)
7FF4h 4    Allocated Area
7FF0h 4    Pointer to Sound Buffer
7FE0h 16   Allocated Area
7FA0h 64   Default area for SP_svc Supervisor Stack (4 words/time)
7F00h 160  Default area for SP_irq Interrupt Stack (6 words/time)
```

Memory below 7F00h is free for User Stack and user data. The three stack pointers are initially initialized at the TOP of the respective areas:

```
SP_svc=03007FE0h
SP_irq=03007FA0h
SP_usr=03007F00h
```

The user may redefine these addresses and move stacks into other locations, however, the addresses for system data at 7FE0h-7FFFh are fixed.

Not sure, is following free for user ???
Registers R8-R12_fiq, R13_fiq, R14_fiq, SPSR_fiq
Registers R13-R14_abt, SPSR_abt
Registers R13-R14_und, SPSR_und

Fast Interrupt (FIQ)
The ARM CPU provides two interrupt sources, IRQ and FIQ. In the GBA only IRQ is used. In normal GBAs, the FIQ signal is shortcut to VDD35, ie. the signal is always high, and there is no way to generate a FIQ by hardware. The registers R8..12_fiq could be used by software (when switching into FIQ mode by writing to CPSR) - however, this might make the game incompatible with hardware debuggers (which are reportedly using FIQs for debugging purposes).

System Control

204h - WAITCNT (formerly WSCNT) - Waitstate Control (R/W)
This register is used to configure game pak access timings. The game pak ROM is mirrored to three address regions at 08000000h, 0A000000h, and 0C000000h, these areas are called Wait State 0-2. Different access timings may be assigned to each area (this might be useful in case that a game pak contains several ROM chips with different access times each).

```
Bit   Expl.
0-1   SRAM Wait Control          (0..3 = 4,3,2,8 cycles)
2-3   Wait State 0 First Access  (0..3 = 4,3,2,8 cycles)
4     Wait State 0 Second Access (0..1 = 2,1 cycles)
5-6   Wait State 1 First Access  (0..3 = 4,3,2,8 cycles)
7     Wait State 1 Second Access (0..1 = 4,1 cycles; unlike above WS0)
8-9   Wait State 2 First Access  (0..3 = 4,3,2,8 cycles)
10    Wait State 2 Second Access (0..1 = 8,1 cycles; unlike above WS0,WS1)
11-12 PHI Terminal Output        (0..3 = Disable, 4.19MHz, 8.38MHz, 16.76MHz)
13    Not used
14    Game Pak Prefetch Buffer (Pipe) (0=Disable, 1=Enable)
15    Game Pak Type Flag  (Read Only) (0=GBA, 1=CGB)
```

At startup, the default setting is 0000h. Currently manufactured cartridges are using the following settings: WS0/ROM=3,1 clks; SRAM=8 clks; WS2/EEPROM: 8,8 clks; prefetch enabled; that is, WAITCNT=4317h, for more info see "Cartridges" chapter.

First Access (Non-sequential) and Second Access (Sequential) define the waitstates for N and S cycles, the actual access time is 1 clock cycle PLUS the number of waitstates.
GamePak uses 16bit data bus, so that a 32bit access is split into TWO 16bit accesses (of which, the second fragment is always sequential, even if the first fragment was non-sequential).

When prefetch buffer is enabled, the GBA attempts to read opcodes from Game Pak ROM during periods when the CPU is not using the bus (if any). Memory access is then performed with 0 Waits if the CPU requests data which is already stored in the buffer.

The PHI Terminal output (PHI Pin of Gamepak Bus) should be disabled.

300h - HALTCNT - Undocumented - Power Down Control (W)
This 16bit register is split into two 8bit registers which are typically addressed separately for different purposes:
300h - BYTE - Undocumented - First Boot / Debug Control (R/W)
After initial reset, the GBA BIOS initializes the register to 01h, and any further execution of the Reset vector (00000000h) will pass control to the Debug vector (0000001Ch) when sensing the register to be still set to 01h.

```
Bit   Expl.
0     Undocumented. First Boot Flag  (0=First, 1=Further)
1-7   Undocumented. Not used.
```

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, much like nocash

burst boot for multiboot software). I am not sure if it is possible to reset the GBA externally without automatically resetting register 300h though.
301h - BYTE - Undocumented - Low Power Mode Control (W)
Writing to this register switches the GBA into battery saving mode.
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.

```
Bit   Expl.
0-6   Undocumented. Not used.
7     Undocumented. Power Down Mode  (0=Halt, 1=Stop)
```

The current GBA BIOS addresses only the upper eight bits of this register (by writing 00h or 80h to address 04000301h), 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.

Also, eventually there should be an undocumented register that is used to mask out cartridge memory (except first 4KBytes of ROM) in Single Game Pak slave mode ???

410h - Undocumented - Purpose Unknown ??? 8bit (W)
The BIOS writes the 8bit value 0FFh to this address. Purpose Unknown.
Probably just another bug in the BIOS.

800h - 32bit - Undocumented - Internal Memory Control (R/W)
Initialized to 0D000020h (by hardware). Unlike all other I/O registers, this register is mirrored accross the whole 4XXXXXXh I/O area (in increments of 64K, ie. at 800h, 10800h, 20800h, etc.)

```
Bit   Expl.
0     Purpose Unknown  (Seems to lock up the GBA when set to 1)
1-3   Purpose Unknown  (Read/Write able)
4     Purpose Unknown  (Always zero - not used or write only)
5     Purpose Unknown  (Seems to lock up the GBA when set to 0)
6-23  Purpose Unknown  (Always zero - not used or write only)
24-27 Wait Control WRAM 256K (0-14 = 15..1 Waitstates, 15=Lockup)
28-31 Purpose Unknown  (Read/Write able)
```

The value 0Dh in Bits 24-27 selects 2 waitstates for 256K WRAM (ie. 3/3/6 cycles 8/16/32bit accesses). The fastest possible setting would be 0Eh (1 waitstate, 2/2/4 cycles for 8/16/32bit). Don't use! Or only at own risk! No promises that it runs stable, and/or that it works on other GBAs.

Note: One cycle equals approx. 59.59ns (ie. 16.78MHz clock).

Cartridges

ROM
Cartridge Header
Cartridge ROM

Backup Media
Aside from ROM, cartridges may also include one of the following backup medias, used to store game positions, highscore tables, options, or other data.
Backup SRAM
Backup EEPROM
Backup Flash ROM

Backup DACS

Cartridge Header

The first 192 bytes at 8000000h-80000BFh in ROM are used as cartridge header. The same header is also used for Multiboot images at 2000000h-20000BFh (plus some additional multiboot entries at 20000C0h and up).

Header Overview

```
Address Bytes Expl.
000h    4     ROM Entry Point  (32bit ARM branch opcode, eg. "B rom_start")
004h    156   Nintendo Logo    (compressed bitmap, required!)
0A0h    12    Game Title       (uppercase ascii, max 12 characters)
0ACh    4     Game Code        (uppercase ascii, 4 characters)
0B0h    2     Maker Code       (uppercase ascii, 2 characters)
0B2h    1     Fixed value      (must be 96h, required!)
0B3h    1     Main unit code   (00h for current GBA models)
0B4h    1     Device type      (huh ???)
0B5h    7     Reserved Area    (should be zero filled)
0BCh    1     Software version (usually 00h)
0BDh    1     Complement check (header checksum, required!)
0BEh    2     Reserved Area    (should be zero filled)
--- Additional Multiboot Header Entries ---
0C0h    4     RAM Entry Point  (32bit ARM branch opcode, eg. "B ram_start")
0C4h    1     Boot mode        (init as 00h - BIOS overwrites this value!)
0C5h    1     Slace ID Number  (init as 00h - BIOS overwrites this value!)
0C6h    26    Not used         (seems to be unused)
0E4h    4     JOYBUS Entry Pt. (32bit ARM branch opcode, eg. "B joy_start")
```

Note: With all entry points, the CPU is initially set into system mode.

000h - Entry Point, 4 Bytes
Space for a single 32bit ARM opcode that redirects to the actual startaddress of the cartridge, this should be usually a "B <start>" instruction.
Note: This entry is ignored by Multiboot slave GBAs (in fact, the entry is then overwritten and redirtected to a separate Multiboot Entry Point, as described below).

004h..09Fh - Nintendo Logo, 156 Bytes
Contains the Nintendo logo which is displayed during the boot procedure. Cartridge won't work if this data is missing or modified.
In detail: This area contains Huffman compression data (but excluding the compression header which is hardcoded in the BIOS, so that it'd be probably not possible to hack the GBA by producing de-compression buffer overflows).
A copy of the compression data is stored in the BIOS, the GBA will compare this data and lock-up itself if the BIOS data isn't exactly the same as in the cartridge (or multiboot header). The only exception are the two entries below which are allowed to have variable settings in some bits.

09Ch Bit 2,7 - Debugging Enable
This is part of the above Nintendo Logo area, and must be commonly set to 21h, however, Bit 2 and Bit 7 may be set to other values.
When both bits are set (ie. A5h), the FIQ/Undefined Instruction handler in the BIOS becomes unlocked, the handler then forwards these exceptions to the user handler in cartridge ROM (entry point defined in 80000B4h, see below).
Other bit combinations currently do not seem to have special functions.

09Eh Bit 0,1 - Cartridge Key Number MSBs
This is part of the above Nintendo Logo area, and must be commonly set to F8h, however, Bit 0-1 may be

set to other values.
During startup, the BIOS performs some dummy-reads from a stream of pre-defined addresses, even though these reads seem to be meaningless, they might be intended to unlock a read-protection inside of commercial cartridge. There are 16 pre-defined address streams - selected by a 4bit key number - of which the upper two bits are gained from 800009Eh Bit 0-1, and the lower two bits from a checksum across header bytes 09Dh..0B7h (bytewise XORed, divided by 40h).

0A0h - Game Title, Uppercase Ascii, max 12 characters
Space for the game title. If less than 12 chars, SPACE or ZERO padded ???

0ACh - Game Code, Uppercase Ascii, 4 characters
This is the same code as the AGB-XXXX code which is printed on the package and sticker on (commercial) cartridges (excluding the leading "AGB-" part).

0B0h - Maker code, Uppercase Ascii, 2 characters
Identifies the (commercial) developer. For example, "01"=Nintendo.

0B2h - Fixed value, 1 Byte
Must be 96h.

0B3h - Main unit code, 1 Byte
Identifies the required hardware. Should be 00h for current GBA models.

0B4h - Device type, 1 Byte
Normally, this entry should be zero. With Nintendos hardware debugger Bit 7 identifies the debugging handlers entry point and size of DACS (Debugging And Communication System) memory: Bit7=0: 9FFC000h/8MBIT DACS, Bit7=1: 9FE2000h/1MBIT DACS. The debugging handler can be enabled in 800009Ch (see above), normal cartridges do not have any memory (nor any mirrors) at these addresses though.

0B5h - Reserved Area, 7 Bytes
Reserved, zero filled.

0BCh - Software version number
Version number of the game. Usually zero.

0BDh - Complement check, 1 Byte
Header checksum, cartridge won't work if incorrect. Calculate as such:
chk=0:for i=0A0h to 0BCh:chk=chk-[i]:next:chk=(chk-19h) and 0FFh

0BEh - Reserved Area, 2 Bytes
Reserved, zero filled.

Below required for Multiboot/slave programs only. For Multiboot, the above 192 bytes are required to be transferred as header-block (loaded to 2000000h-20000BFh), and some additional header-information must be located at the beginning of the actual program/data-block (loaded to 20000C0h and up). This extended header consists of Multiboot Entry point(s) which must be set up correctly, and two reserved bytes which are overwritten by the boot procedure:

0C0h - Normal/Multiplay mode Entry Point
This entry is used only if the GBA has been booted by using Normal or Multiplay transfer mode (but not by Joybus mode).
Typically deposit a ARM-32bit "B <start>" branch opcode at this location, which is pointing to your actual initialization procedure.

0C4h (BYTE) - Boot mode
The slave GBA download procedure overwrites this byte by a value which is indicating the used multiboot transfer mode.

```
Value  Expl.
01h    Joybus mode
02h    Normal mode
03h    Multiplay mode
```

Typically set this byte to zero by inserting DCB 00h in your source.
Be sure that your uploaded program does not contain important program code or data at this location, or at the ID-byte location below.

0C5h (BYTE) - Slave ID Number
If the GBA has been booted in Normal or Multiplay mode, this byte becomes overwritten by the slave ID number of the local GBA (that'd be always 01h for normal mode).

```
Value  Expl.
01h    Slave #1
02h    Slave #2
03h    Slave #3
```

Typically set this byte to zero by inserting DCB 00h in your source.
When booted in Joybus mode, the value is NOT changed and remains the same as uploaded from the master GBA.

0C6h..0DFh - Not used
Appears to be unused.

0E0h - Joybus mode Entry Point
If the GBA has been booted by using Joybus transfer mode, then the entry point is located at this address rather than at 20000C0h. Either put your initialization procedure directly at this address, or redirect to the actual boot procedure by depositing a "B <start>" opcode here (either one using 32bit ARM code). Or, if you are not intending to support joybus mode (which is probably rarely used), ignore this entry.

Cartridge ROM

ROM Size
The games F-ZERO and Super Mario Advance use ROMs of 4 MBytes each.
Not sure if other sizes are available.

ROM Waitstates
The GBA starts the cartridge with 4,2 waitstates (N,S) and prefetch disabled. The program may change these settings by writing to WAITCNT, the games F-ZERO and Super Mario Advance use 3,1 waitstates (N,S) each, with prefetch enabled.
Third-party flashcards are reportedly running unstable with these settings. Also, prefetch and shorter waitstates are allowing to read more data and opcodes from ROM is less time, the downside is that it increases the power consumption.

ROM Chip
Because of how 24bit addresses are squeezed through the Gampak bus, the cartridge must include a circuit that latches the lower 16 address bits on non-sequential access, and that increments these bits on sequential access. Nintendo includes this circuit directly in the ROM chip.
Also, the ROM must have 16bit data bus, or otherwise another circuit is required which converts two 8bit data units into one 16bit unit - by not exceeding the waitstate timings.

Backup SRAM

32 KBytes - 256Kbit Battery buffered SRAM - Lifetime: Depends on battery

Addressing and Waitstates
SRAM is mapped to E000000h-E007FFFh, it should be accessed with 8 waitstates (write a value of 3 into Bit0-1 of WAITCNT).

Databus Width
The SRAM databus is restricted to 8 bits, it should be accessed by LDRB, LDRSB, and STRB opcodes only.

Reading and Writing
Reading from SRAM should be performed by code exectued in WRAM only (but not by code executed in ROM). There is no such restriction for writing.

Preventing Data Loss
The GBA SRAM carts do not include a write-protect function (unlike older 8bit gameboy carts). This seems to be a problem and may cause data loss when a cartridge is removed or inserted while the GBA is still turned on. As far as I understand, this is not so much a hardware problem, but rather a software problem, ie. theoretically you could remove/insert the cartridge as many times as you want, but you should take care that your program does not crash (and write blindly into memory).

Recommended Workaround
Enable the Gamepak Interrrupt (it'll most likely get triggered when removing the cartridge), and hang-up the GBA in an endless loop when your interrupt handler senses a Gamepak IRQ. For obvious reason, your interrupt handler should be located in WRAM, ie. not in the (removed) ROM cartridge. The handler should process Gamepak IRQs at highest priority. Periods during which interrupts are disabled should be kept as short as possible, if necessary allow nested interrupts.

When to use the above Workaround
A program that relies wholly on code and data in WRAM, and that does not crash even when ROM is removed, may keep operating without having to use the above mechanism.
Do NOT use the workaround for programs that run without a cartridge inserted (ie. single gamepak/multiboot slaves), or for programs that use Gamepak IRQ/DMA for other purposes.
All other programs should use it. It'd be eventually a good idea to include it even in programs that do not use SRAM themselves (eg. otherwise removing a SRAM-less cartridge may lock up the GBA, and may cause it to destroy backup data when inserting a SRAM cartridge).

Note
SRAM is used by the game F-ZERO, and possibly others.

In SRAM cartridges, the /REQ pin (Pin 31 of Gamepak bus) should be a little bit shorter as than the other pins; when removing the cartridge, this causes the gamepak IRQ signal to go off before the other pins are disconnected.

Backup EEPROM

512 Bytes (0200h) - 4Kbit EEPROM - Lifetime: 100,000 writes per address
8 KBytes (2000h) - 64Kbit EEPROM - Lifetime: No info.

Addressing and Waitstates

The eeprom is connected to Bit0 of the data bus, and the MSB of the cartridge ROM address bus, communication with the chip takes place serially. With this circuit, only 16MB of ROM can be used (!), the upper 16MB of the "ROM" area are all mirrors of the EEPROM.
The chip should be accessed at 8 waitstates (set WAITCNT=X3XXh; 8,8 clks in WS2 area), to access the eeprom, use address D000000h (the first address in the upper half of the WS2 area).

Data and Address Width
Data can be read from (or written to) the EEPROM in units of 64bits (8 bytes). Writing automatically erases the old 64bits of data. Addressing works in units of 64bits respectively, that is, for 512 Bytes EEPROMS: an address range of 0-3Fh, 6bit bus width; and for 8KByte EEPROMs: a range of 0-3FFh, apparently 14bit bus width ??? (if so, note that only the lower 10 address bits are used, upper 4 bits should be zero).

Set Address (For Reading)
Prepare the following bitstream in memory:

```
  2 bits "11" (Read Request)
  n bits eeprom address (MSB first, 6 or 14 bits, depending on EEPROM)
  1 bit "0"
```

Then transfer the stream to eeprom by using DMA.

Read Data
Read a stream of 68 bits from EEPROM by using DMA,
then decipher the received data as follows:

```
  4 bits - ignore these
 64 bits - data (conventionally MSB first)
```

Write Data to Address
Prepare the following bitstream in memory, then transfer the stream to eeprom by using DMA, it'll take ca. 108368 clock cycles (ca. 6.5ms) until the old data is erased and new data is programmed.

```
  2 bits "10" (Write Request)
  n bits eeprom address (MSB first, 6 or 14 bits, depending on EEPROM)
 64 bits data (conventionally MSB first)
  1 bit "0"
```

After the DMA, keep reading from the chip, by normal LDRH [D000000h], until Bit 0 of the returned data becomes "1" (Ready). To prevent your program from locking up in case of malfunction, generate a timeout if the chip does not reply after 10ms or longer.

Using DMA
Transfering a bitstream to/from the EEPROM by LDRH/STRH opcodes does not work, this might be because of timing problems, or because how the GBA squeezes non-sequential memory addresses through the external address/data bus.
For this reason, a buffer in memory must be used (that buffer would be typically allocated temporarily on stack, one halfword for each bit, bit1-15 of the halfwords are don't care, only bit0 is of interest).
The buffer must be transfered as a whole to/from EEPROM by using DMA3 (only DMA 3 is valid to read & write external memory), use 16bit transfer mode, both source and destinal address incrementing (ie. DMA3CNT=80000000h+length).
DMA channels of higher priority should be disabled during the transfer (ie. H/V-Blank or Sound FIFO DMAs). And, of course any interrupts that might mess with DMA registers should be disabled.

Pin-Outs
The EEPROM chips are having only 8 pins, these are connected, Pin 1..8, to ROMCS, RD, WR, AD0, GND, GND, A23, VDD of the GamePak bus.

Notes

There seems to be no autodection mechanism, so that a hardcoded bus width must be used. The game Super Mario Advance uses a 512 Byte EEPROM, no idea which games use 8KBytes (if any) ???

Backup Flash ROM

64 KBytes - 512Kbits Flash ROM - Lifetime: 10,000 writes per sector

The chip is connected to the "SRAM" area at 0E000000h-0E00FFFFh. No programming info available, except that writing takes place in units of 4KBytes (sectors). Reading may be performed bytewise. Nintendo supports chips manufactured by Sanyo and Amtel, GBA software that uses Flash memory should be compatible with both types.

Backup DACS

128 KBytes - 1Mbit DACS - Lifetime: 100,000 writes.
1024 KBytes - 8Mbit DACS - Lifetime: 100,000 writes.

DACS (Debugging And Communication System) is used in Nintendos hardware debugger only, DACS is NOT used in normal game cartridges.

Parts of DACS memory is used to store the debugging exception handlers (entry point/size defined in cartridge header), the remaining memory could be used to store game positions or other data. The address space is the upper end of the 32MB ROM area, the memory can be read directly by the CPU, including for ability to execute program code in this area.

BIOS Functions

The GBA BIOS includes several System Call Functions which can be accessed by SWI instructions. Incoming parameters are usually passed through registers R0,R1,R2,R3. Outgoing registers R0,R1,R3 are typically containing either garbage, or return value(s). All other registers (R2,R4-R14) are kept unchanged.

Caution
When invoking SWIs from inside of ARM state specify SWI NN*10000h, instead of SWI NN as in THUMB state.

Overview
BIOS Function Summary

All Functions Described

How BIOS Processes SWIs
SWIs can be called from both within THUMB and ARM mode. In ARM mode, only the upper 8bit of the 24bit comment field are interpreted.
Each time when calling a BIOS function 4 words (SPSR, R11, R12, R14) are saved on Supervisor stack (_svc). Once it has saved that data, the SWI handler switches into System mode, so that all further stack operations are using user stack.
In some cases the BIOS may allow interrupts to be executed from inside of the SWI procedure. If so, and if the interrupt handler calls further SWIs, then care should be taken that the Supervisor Stack does not overflow.

BIOS Function Summary

SWI	Hex	Function
0	00h	SoftReset
1	01h	RegisterRamReset
2	02h	Halt
3	03h	Stop
4	04h	IntrWait
5	05h	VBlankIntrWait
6	06h	Div
7	07h	DivArm
8	08h	Sqrt
9	09h	ArcTan
10	0Ah	ArcTan2
11	0Bh	CpuSet
12	0Ch	CpuFastSet
13	0Dh	-Undoc- ("GetBiosChecksum")
14	0Eh	BgAffineSet
15	0Fh	ObjAffineSet
16	10h	BitUnPack
17	11h	LZ77UnCompWram
18	12h	LZ77UnCompVram
19	13h	HuffUnComp
20	14h	RLUnCompWram
21	15h	RLUnCompVram
22	16h	Diff8bitUnFilterWram
23	17h	Diff8bitUnFilterVram
24	18h	Diff16bitUnFilter
25	19h	SoundBias
26	1Ah	SoundDriverInit
27	1Bh	SoundDriverMode
28	1Ch	SoundDriverMain
29	1Dh	SoundDriverVSync
30	1Eh	SoundChannelClear
31	1Fh	MidiKey2Freq
32-36	20h-24h	-Undoc- (Sound Related ???)
37	25h	MultiBoot
38	26h	-Undoc- ("HardReset")
39	27h	-Undoc- ("CustomHalt")
40	28h	SoundDriverVSyncOff
41	29h	SoundDriverVSyncOn
42	2Ah	-Undoc- ("GetJumpList" for Sound ???)
43-255	2Bh-FFh	-Not used-

The BIOS SWI handler does not perform any range checks, so calling SWI 43-255 will blindly lock up the GBA.

BIOS Arithmetic Functions

Div
DivArm
Sqrt
ArcTan
ArcTan2

SWI 6 - Div
Signed Division, r0/r1.

```
r0  signed 32bit Number
r1  signed 32bit Denom
```

Return:

```
r0  Number DIV Denom
r1  Number MOD Denom
r3  ABS (Number DIV Denom)
```

For example, incoming -1234, 10 should return -123, -4, +123.
The function usually gets caught in an endless loop upon division by zero.

SWI 7 - DivArm
Same as above (SWI 6 Div), but incoming parameters are exchanged, r1/r0 (r0=Denom, r1=number). For compatibility with ARM's library. Slightly slower (3 clock cycles) than SWI 6.

SWI 8 - Sqrt
Calculate square root.

```
r0   unsigned 32bit number
```

Return:

```
r0   unsigned 16bit number
```

The result is an integer value, for example Sqrt(2) would return 1, to avoid this inaccuracy, shift left incoming number by 2*N as much as possible (the result is then shifted left by 1*N). Ie. Sqrt(2 shl 30) would return 1.41421 shl 15.

SWI 9 - ArcTan
Calculates the arc tangent.

```
r0   Tan, 16bit (1bit sign, 1bit integral part, 14bit decimal part)
```

Return:

```
r0   "-PI/2<THETA/<PI/2" in a range of C000h-4000h.
```

Note: there is a problem in accuracy with "THETA<-PI/4, PI/4<THETA".

SWI 10 (0Ah) - ArcTan2
Calculates the arc tangent after correction processing.
Use this in normal situations.

```
r0   X, 16bit (1bit sign, 1bit integral part, 14bit decimal part)
r1   Y, 16bit (1bit sign, 1bit integral part, 14bit decimal part)
```

Return:

```
r0   0000h-FFFFh for 0<=THETA<2PI.
```

BIOS Rotation/Scaling Functions

BgAffineSet
ObjAffineSet

SWI 14 (0Eh) - BgAffineSet
Used to calculate BG Rotation/Scaling parameters.

```
r0   Pointer to Source Data Field with entries as follows:
      s32  Original data's center X coordinate (8bit fractional portion)
      s32  Original data's center Y coordinate (8bit fractional portion)
      s16  Display's center X coordinate
      s16  Display's center Y coordinate
      s16  Scaling ratio in X direction (8bit fractional portion)
      s16  Scaling ratio in Y direction (8bit fractional portion)
      u16  Angle of rotation (8bit fractional portion) Effective Range 0-FFFF
r1   Pointer to Destination Data Field with entries as follows:
      s16  Difference in X coordinate along same line
      s16  Difference in X coordinate along next line
      s16  Difference in Y coordinate along same line
      s16  Difference in Y coordinate along next line
      s32  Start X coordinate
      s32  Start Y coordinate
r2   Number of Calculations
```

Return: No return value, Data written to destination address.

SWI 15 (0Fh) - ObjAffineSet
Calculates and sets the OBJ's affine parameters from the scaling ratio and angle of rotation.
The affine parameters are calculated from the parameters set in Srcp.
The four affine parameters are set every Offset bytes, starting from the Destp address.
If the Offset value is 2, the parameters are stored contiguously. If the value is 8, they match the structure of OAM.
When Srcp is arrayed, the calculation can be performed continuously by specifying Num.

```
r0   Source Address, pointing to data structure as such:
      s16  Scaling ratio in X direction (8bit fractional portion)
      s16  Scaling ratio in Y direction (8bit fractional portion)
      u16  Angle of rotation (8bit fractional portion) Effective Range 0-FFFF
r1   Destination Address, pointing to data structure as such:
      s16  Difference in X coordinate along same line
      s16  Difference in X coordinate along next line
      s16  Difference in Y coordinate along same line
      s16  Difference in Y coordinate along next line
r2   Number of calculations
r3   Offset in bytes for parameter addresses (2=contigiously, 8=OAM)
```

Return: No return value, Data written to destination address.

For both Bg- and ObjAffineSet, Rotation angles are specified as 0-FFFFh (covering a range of 360 degrees), however, the GBA BIOS recurses only the upper 8bit; the lower 8bit may contain a fractional portion, but it is ignored by the BIOS.

BIOS Decompression Functions

BitUnPack
Diff8bitUnFilterWram
Diff8bitUnFilterVram
Diff16bitUnFilter
HuffUnComp
LZ77UnCompWram
LZ77UnCompVram

RLUnCompVram
RLUnCompWram

SWI 16 (10h) - BitUnPack
Used to increase the color depth of bitmaps or tile data. For example, to convert a 1bit monochrome font into 4bit or 8bit GBA tiles. The Unpack Info is specified separately, allowing to convert the same source data into different formats.

```
r0  Source Address      (no alignment required)
r1  Destination Address (must be 32bit-word aligned)
r2  Pointer to UnPack information:
     16bit  Length of Source Data in bytes     (0-FFFFh)
     8bit   Width of Source Units in bits      (only 1,2,4,8 supported)
     8bit   Width of Destination Units in bits (only 1,2,4,8,16,32 supported)
     32bit  Data Offset (Bit 0-30), and Zero Data Flag (Bit 31)
    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 32bit 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.
Return: No return value, Data written to destination address.

SWI 22 (16h) - Diff8bitUnFilterWram
SWI 23 (17h) - Diff8bitUnFilterVram
SWI 24 (18h) - Diff16bitUnFilter
These aren't actually real decompression functions, destination data will have exactly the same size as source data. However, assume a bitmap or wave form to contain a stream of increasing numbers such like 10..19, the filtered/unfiltered data would be:

```
unfiltered:   10  11  12  13  14  15  16  17  18  19
filtered:     10  +1  +1  +1  +1  +1  +1  +1  +1  +1
```

In this case using filtered data (combined with actual compression algorhytms) will obviously produce better compression results.
Data units may be either 8bit or 16bit used with Diff8bit or Diff16bit functions respectively. The 8bitVram function allows to write to VRAM directly (which uses 16bit data bus) by writing two 8bit values at once, the downside is that it is eventually slower as the 8bitWram function.

```
r0  Source address (must be aligned by 4) pointing to data as follows:
     Data Header (32bit)
       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
       ...
r1  Destination address
```

Return: No return value, Data written to destination address.

SWI 19 (13h) - HuffUnComp
Expands Huffman-compressed data and writes in units of 32bits.
If the size of the compressed data is not a multiple of 4, please adjust it as much as possible by padding with 0.
Align the source address to a 4Byte boundary.

```
r0  Source Address, aligned by 4, pointing to:
     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:
        u8
         6bit  offset to next node -1 (2 byte units)
         1bit  right node end flag (if set, data is in next node)
         1bit  left node end flag
       1 node  Root node
       2 nodes Left, and Right node
       4 nodes LeftLeft, LeftRight, RightLeft, and RightRight node
       ...
      Compressed data
       ...
r1  Destination Address
```

Return: No return value, Data written to destination address.

SWI 17 (11h) - LZ77UnCompWram
SWI 18 (12h) - LZ77UnCompVram
Expands LZ77-compressed data. The Wram function is faster, and writes in units of 8bits. For the Vram function the destination must be halfword aligned, data is written in units of 16bits.
If the size of the compressed data is not a multiple of 4, please adjust it as much as possible by padding with 0. Align the source address to a 4-Byte boundary.

```
r0   Source address, pointing to data as such:
      Data header (32bit)
        Bit 0-3   Reserved
        Bit 4-7   Compressed type (must be 1 for LZ77)
        Bit 8-31  Size of decompressed data
      Repeat below. Each Flag Byte followed by eight Blocks.
      Flag data (8bit)
        Bit 0-7   Type Flags for next 8 Blocks, MSB first
      Block Type 0 - Uncompressed - Copy 1 Byte from Source to Dest
        Bit 0-7   One data byte to be copied to dest
      Block Type 1 - Compressed - Copy N+3 Bytes from Dest-Disp-1 to Dest
        Bit 0-3   Disp MSBs
        Bit 4-7   Number of bytes to copy (minus 3)
        Bit 8-15  Disp LSBs
r1   Destination address
```

Return: No return value.

SWI 21 (15h) - RLUnCompVram
SWI 20 (14h) - RLUnCompWram
Expands run-length compressed data. The Wram function is faster, and writes in units of 8bits. For the Vram function the destination must be halfword aligned, data is written in units of 16bits.
If the size of the compressed data is not a multiple of 4, please adjust it as much as possible by padding with 0. Align the source address to a 4Byte boundary.

```
r0  Source Address, pointing to data as such:
     Data header (32bit)
       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
r1  Destination Address
```

Return: No return value, Data written to destination address.

BIOS Memory Copy

CpuFastSet
CpuSet

Note: These two functions will silently reject to do anything if the start or end address of the source area are reaching into the BIOS area.

SWI 12 (0Ch) - CpuFastSet
Memory copy/fill in units of 32 bytes. Memcopy is implemented as repeated LDMIA/STMIA [Rb]!,r2-r9 instructions. Memfill as single LDR followed by repeated STMIA [Rb]!,r2-r9.
The length must be a multiple of 32 bytes. The wordcount in r2 must be length/4, ie. length in word units rather than byte units.

```
r0    Source address        (must be aligned by 4)
r1    Destination address   (must be aligned by 4)
r2    Length/Mode
        Bit 0-15  Wordcount (must be multiple of 8 WORDs, ie. 32 bytes)
        Bit 24    Fixed Source Address (0=Copy, 1=Fill by WORD[r0])
```

Return: No return value, Data written to destination address.

SWI 11 (0Bh) - CpuSet
Memory copy/fill in units of 4 bytes or 2 bytes. Memcopy is implemented as repeated LDMIA/STMIA [Rb]!,r3 or LDRH/STRH r3,[r0,r5] instructions. Memfill as single LDMIA or LDRH followed by repeated STMIA [Rb]!,r3 or STRH r3,[r0,r5].
The length must be a multiple of 4 bytes (32bit mode) or 2 bytes (16bit mode). The (half)wordcount in r2 must be length/4 (32bit mode) or length/2 (16bit mode), ie. length in word/halfword units rather than byte units.

```
r0    Source address        (must be aligned by 4 for 32bit, by 2 for 16bit)
r1    Destination address   (must be aligned by 4 for 32bit, by 2 for 16bit)
r2    Length/Mode
        Bit 0-15  Wordcount (for 32bit), or Halfwordcount (for 16bit)
        Bit 24    Fixed Source Address (0=Copy, 1=Fill by {HALF}WORD[r0])
        Bit 26    Datasize (0=16bit, 1=32bit)
```

Return: No return value, Data written to destination address.

BIOS Halt Functions

Halt
IntrWait
VBlankIntrWait
Stop

SWI 2 - Halt
Switch the CPU into low-power mode, all other circuits (video, sound, timers, serial, keypad, system clock) are kept operating.
The Halt mode is terminated when any interrupt becomes executed (requires that interrupt(s) are enabled in IE register).
Use this whenever the CPU is not busy, ie. when waiting interrupt events.
No parameters, no return value.

SWI 4 - IntrWait
Continues to wait in Halt state until one (or more) of the specified interrupt(s) do occur. When using multiple interrupts at the same time, this function is having less overhead as when repeatedly calling SWI 2 until the desired interrupt occurs.

```
r0    0=Return immediately if an old flag was already set.
      1=Discard old flags, wait until a NEW flag becomes set.
r1    Specification of IE/IF interrupt flag(s) to wait for.
```

The interrupt handler must update the 16bit value at [3007FF8h] in WRAM by software: When the IRQ handler acknowledges interrupt(s) by writing a value to the IF register, it should also logically OR the same value to [3007FF8h].
Return: No return value.
The selected-and-occured flag(s) are automatically reset in [3007FF8h] upon return.

SWI 5 - VBlankIntrWait
Continues to wait in Halt status until a new V-Blank interrupt occurs.
The function sets r0=1 and r1=1 and then executes IntrWait (SWI 4), see Intr Wait for details.
No parameters, no return value.

SWI 3 - Stop (formerly FullStop)
Switches the GBA into very low power mode (to be used similiar as a screen-saver). The CPU, System Clock, Sound, Video, SIO-Shift Clock, DMAs, and Timers are stopped.
Stop state can be terminated by the following interrupts only (as far as enabled in IE register): Joypad, Game Pak, or General-Purpose-SIO.
"The system clock is stopped so the IF flag is not set."
Preparation for Stop:
Disable Video before implementing Stop (otherwise Video just freezes, but still keeps consuming battery power). Possibly required to disable Sound also ??? Obviously, it'd be also recommended to disable any external hardware (such like Rumble or Infra-Red) as far as possible.
No parameters, no return value.

BIOS Reset Functions

SoftReset
RegisterRamReset

SWI 0 - SoftReset
Clears the CPU internal RAM area from 3007E00h-3007FFFh, initializes system, supervisor, and irq stack pointers, and sets R0-R12 to zero, and enters system mode.

```
[3007FFAh] Return Address Select (00h for 8000000h, 01h-FFh for 2000000h)
```

Return: Does not return to calling procedure, instead, jumps to 8000000h (ROM) or 2000000h (WRAM), in ARM state.

SWI 1 - RegisterRamReset
Resets the I/O registers and RAM specified in ResetFlags. However, it does not clear the CPU internal RAM area from 3007E00h-3007FFFh.

```
r0  ResetFlags
     Bit   Expl.
     0     Clear 256K on-board WRAM  ;-don't use when returning to WRAM
     1     Clear 32K in-chip WRAM    ;-excluding last 200h bytes
     2     Clear Palette
     3     Clear VRAM
     4     Clear OAM              ;-zerofilled! does NOT disable OBJs!
     5     Reset SIO registers    ;-switches to general purpose mode!
     6     Reset Sound registers
     7     Reset all other registers (except SIO, Sound)
```

Return: No return value.
Bug: LSBs of SIODATA32 are always destroyed, even if Bit5 of R0 was cleared.

The function always switches the screen into forced blank by setting DISPCNT=0080h (regardless of incoming R0, screen becomes white).

BIOS Multi Boot (Single Game Pak)

MultiBoot

SWI 37 (25h) - MultiBoot
This function uploads & starts program code to slave GBAs, allowing to launch programs on slave units even if no cartridge is inserted into the slaves (this works because all GBA BIOSes contain built-in download procedures in ROM).
However, the SWI 37 BIOS upload function covers only 45% of the required Transmission Protocol, the other 55% must be coded in the master cartridge (see Transmission Protocol below).

```
r7  Pointer to MultiBootParam structure
r1  Transfer Mode (undocumented)
     0=256KHz, 32bit, Normal mode    (fast and stable)
     1=115KHz, 16bit, MultiPlay mode (default, slow, up to three slaves)
     2=2MHz,   32bit, Normal mode    (fastest but maybe unstable)
Note: HLL-programmers that are using the MultiBoot(param_ptr) macro cannot
specify the transfer mode and will be forcefully using MultiPlay mode.
```

Return:

```
r0  0=okay, 1=failed
```

See below for more details.

Multiboot Parameter Structure
Size of parameter structure should be 4Ch bytes (the current GBA BIOS uses only first 44h bytes though). The following entries must be set before calling SWI 37:

```
Addr Size Name/Expl.
14h  1    handshake_data (entry used for normal mode only)
19h  3    client_data[1,2,3]
1Ch  1    palette_data
1Eh  1    client_bit (Bit 1-3 set if child 1-3 detected)
20h  4    boot_srcp  (typically 8000000h+0C0h)
24h  4    boot_endp  (typically 8000000h+0C0h+length)
```

The transfer length (excluding header data) should be a multiple of 10h, minimum length 100h, max 3FF40h (ca. 256KBytes). Set palette_data as "81h+color*10h+direction*8+speed*2", or as "0f1h+color*2" for fixed palette, whereas color=0..6, speed=0..3, direction=0..1. The other entries (handshake_data, client_data[1-3], and client_bit) must be same as specified in Transmission Protocol (see below hh,cc,y).

Multiboot Transfer Protocol
Below describes the complete transfer protocol, normally only the Initiation part must be programmed in the master cartridge, the main data transfer can be then performed by calling SWI 37, the slave program is started after SWI 37 completion.
The ending handshake is normally not required, when using it, note that you will need custom code in BOTH master and slave programs.

```
Times  Send   Receive  Expl.
----------------------Required Transfer Initiation in master program
...    6200   FFFF     Slave not in multiplay/normal mode yet
1      6200   0000     Slave entered correct mode now
15     6200   720x     Repeat 15 times, if failed: delay 1/16s and restart
1      610y   720x     Recognition okay, exchange master/slave info
60h    xxxx   NN0x     Transfer C0h bytes header data in units of 16bits
1      6200   000x     Transfer of header data completed
1      620y   720x     Exchange master/slave info again
```

```
...     63pp   720x     Wait until all slaves reply 73cc instead 720x
1       63pp   73cc     Send palette_data and receive client_data[1-3]
1       64hh   73uu     Send handshake_data for final transfer completion
-----------------------Below is SWI 37 MultiBoot handler in BIOS
DELAY   -      -        Wait 1/16 seconds at master side
1       llll   73rr     Send length information and receive random data[1-3]
LEN     yyyy   nnnn     Transfer main data block in units of 16 or 32 bits
1       0065   nnnn     Transfer of main data block completed, request CRC
...     0065   0074     Wait until all slaves reply 0075 instead 0074
1       0065   0075     All slaves ready for CRC transfer
1       0066   0075     Signalize that transfer of CRC follows
1       zzzz   zzzz     Exchange CRC must be same for master and slaves
-----------------------Optional Handshake (NOT part of master/slave BIOS)
...     ....   ....     Exchange whatever custom data
```

Legend for above Protocol

```
y     client_bit, bit(s) 1-3 set if slave(s) 1-3 detected
x     bit 1,2,or 3 set if slave 1,2,or 3
xxxx  header data, transferred in 16bit (!) units (even in 32bit normal mode)
nn    response value for header transfer, decreasing 60h..01h
pp    palette_data
cc    random client_data[1..3] from slave 1-3, FFh if slave not exists
hh    handshake_data, 11h+client_data[1]+client_data[2]+client_data[3]
uu    random data, not used, ignore this value
```

Below automatically calculated by SWI 37 BIOS function (don't care about)

```
llll  download length/4-34h
rr    random data from each slave for encryption, FFh if slave not exists
yyyy  encoded data in 16bit (multiplay) or 32bit (normal mode) units
nnnn  response value, lower 16bit of destadr in GBA memory (00C0h and up)
zzzz  16bit download CRC value, must be same for master and slaves
```

Multiboot Communication

In Multiplay mode, master sends 16bit data, and receives 16bit data from each slave (or FFFFh if none). In Normal mode, master sends 32bit data (upper 16bit zero, lower 16bit as for multipay mode), and receives 32bit data (upper 16bit as for multiplay mode, and lower 16bit same as lower 16bit previously sent by master). Because SIODATA32 occupies same addresses as SIOMULTI0-1, the same transfer code can be used for both multiplay and normal mode (in normal mode SIOMULTI2-3 should be forced to FFFFh though). After each transfer, master should wait for Start bit cleared in SIOCNT register, followed by a 36us delay.

Note: The multiboot slave would also recognize data being sent in Joybus mode, however, master GBAs cannot use joybus mode (because GBA hardware cannot act as master in joybus mode).

Multiboot Slave Header

The transferred Header block is written to 2000000-20000BFh in slave RAM, the header must contain valid data (identically as for normal ROM-cartridge headers, including a copy of the Nintendo logo, correct header CRC, etc.), in most cases it'd be recommended just to transfer a copy of the master cartridges header from 8000000h-80000BFh.

Multiboot Slave Program/Data

The transferred main program/data block is written to 20000C0h and up (max 203FFFFh) in slave RAM, note that absolute addresses in the program must be then originated at 2000000h rather than 8000000h. In case that the master cartridge is 256K or less, it could just transfer a copy of the whole cartridge at 80000C0h and up, the master should then copy & execute its own ROM data into RAM as well.

Multiboot Slave Extended Header

For Multiboot slaves, separate Entry Point(s) must be defined at the beginning of the Program/Data block (the Entry Point in the normal header is ignored), also some reserved bytes in this section are overwritten by the Multiboot procedure. For more information see chapter about Cartridge Header.

Multiboot Slave with Cartridge
Beside for slaves without cartridge, multiboot can be also used for slaves which do have a cartridge inserted, if so, SELECT and START must be kept held down during power-on in order to switch the slave GBA into Multiboot mode (ie. to prevent it from starting the cartridge as normally).
The general idea is to enable newer programs to link to any existing older GBA programs, even if these older programs originally didn't have been intended to support linking.
The uploaded program may access the slaves SRAM, Flash ROM, or EEPROM (if any, allowing to read out or modify slave game positions), as well as cartridge ROM at 80000A0h-8000FFFh (the first 4KBytes, excluding the nintendo logo, allowing to read out the cartridge name from the header, for example).
The main part of the cartridge ROM is meant to be locked out in order to prevent software pirates from uploading "intruder" programs which would send back a copy of the whole cartridge to the master, however, for good or evil, at present time, current GBA models and GBA carts do not seem to contain any such protection.

Uploading Programs from PC
Beside for the ability to upload a program from one GBA to another, this feature can be also used to upload small programs from a PC to a GBA. For more information see chapter about External Connectors.

BIOS Sound Functions

MidiKey2Freq
SoundBias
SoundChannelClear
SoundDriverInit
SoundDriverMain
SoundDriverMode
SoundDriverVSync
SoundDriverVSyncOff
SoundDriverVSyncOn

SWI 31 (1Fh) - MidiKey2Freq
Calculates the value of the assignment to ((SoundArea)sa).vchn[x].fr when playing the wave data, wa, with the interval (MIDI KEY) mk and the fine adjustment value (halftones=256) fp.

```
r0  WaveData* wa
r1  u8 mk
r2  u8 fp
```

Return:

```
r0  u32
```

This function is particulary popular because it allows to read from BIOS memory without copy protection range checks. The formula to read one byte (a) from address (i, 0..3FF) is:
a = (MidiKey2Freq(i-(((i AND 3)+1)OR 3), 168, 0) * 2) SHR 24

SWI 25 (19h) - SoundBias
Increments or decrements the current level of the SOUNDBIAS register (with short delays) until reaching the desired new level. The upper bits of the register are kept unchanged.

```
r0   BIAS level (0=Level 000h, any other value=Level 200h)
```

Return: No return value.

SWI 30 (1Eh) - SoundChannelClear

Clears all direct sound channels and stops the sound.
This function may not operate properly when the library which expands the sound driver feature is combined afterwards. In this case, do not use it.
No parameters, no return value.

SWI 26 (1Ah) - SoundDriverInit
Initializes the sound driver. Call this only once when the game starts up.
It is essential that the work area already be secured at the time this function is called.
You cannot execute this driver multiple times, even if separate work areas have been prepared.

```
r0  Pointer to work area for sound driver, SoundArea structure as follows:
     SoundArea (sa) Structure
      u32    ident      Flag the system checks to see whether the
                        work area has been initialized and whether it
                        is 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
      SndCh  vchn[MAX]  The structure array for controlling the direct
                        sound channels (currently 8 channels are
                        available). The term "channel" here does
                        not refer to hardware channels, but rather to
                        virtual constructs inside the sound driver.
      s8     pcmbuf[PCM_BF*2]
     SoundChannel Structure
      u8          sf     The flag indicating the status of this channel.
                         When 0 sound is stopped.
                         To start sound, set other parameters and
                         then write 80h to here.
                         To stop sound, logical OR 40h for a
                         release-attached off (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. When the
                         sound starts, the volume begins at zero and
                         increases every 1/60 second. When it
                         reaches 255, the process moves on to the
                         next decay value.
      u8          de     The decay value of the envelope. It is
                         multiplied by "this value/256" every 1/60
                         sec. and when sustain value is reached, the
                         process moves to the sustain condition.
      u8          su     The sustain value of the envelope. The
                         sound is sustained by this amount.
                         (Actually, multiplied by rv/256, lv/256 and
                         output left and right.)
      u8          re     The release value of the envelope. Key-off
                         (logical OR 40h in sf) to enter this state.
                         The value is multiplied by "this value/256"
                         every 1/60 sec. and when it reaches zero,
                         this channel is completely stopped.
      u8          r2[4]  User access prohibited
      u32         fr     The frequency of the produced sound.
                         Write the value obtained with the
                         MidiKey2Freq function here.
      WaveData*   wp     Pointer to the sound's waveform data. The waveform
                         data can be generated automatically from the AIFF
                         file using the tool (aif2agb.exe), so users normally
                         do not need to create this themselves.
      u32         r3[6]  User access prohibited
      u8          r4[4]  User access prohibited
     WaveData Structure
```

```
u16   type    Indicates the data type. This is currently not used.
u16   stat    At the present time, non-looped (1 shot) waveform
              is 0000h and forward loop is 4000h.
u32   freq    This value is used to calculate the frequency.
              It is obtained using the following formula:
              sampling rate x 2^((180-original MIDI key)/12)
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.
```

Return: No return value.

SWI 28 (1Ch) - SoundDriverMain
Main of the sound driver.
Call every 1/60 of a second. The flow of the process is to call SoundDriverVSync, which is explained later, immediately after the V-Blank interrupt.
After that, this routine is called after BG and OBJ processing is executed.
No parameters, no return value.

SWI 27 (1Bh) - SoundDriverMode
Sets the sound driver operation mode.

```
r0  Sound driver operation mode
     Bit    Expl.
     0-6    Direct Sound Reverb value (0-127, default=0) (ignored if Bit7=0)
     7      Direct Sound Reverb set (0=ignore, 1=apply reverb value)
     8-11   Direct Sound Simultaneously-produced (1-12 channels, default 8)
     12-15  Direct Sound Master volume (1-15, default 15)
     16-19  Direct Sound Playback Frequency (1-12 = 5734,7884,10512,13379,
            15768,18157,21024,26758,31536,36314,40137,42048, def 4=13379 Hz)
     20-23  Final number of D/A converter bits (8-11 = 9-6bits, def. 9=8bits)
     24-31  Not used.
```

Return: No return value.

SWI 29 (1Dh) - SoundDriverVSync
An extremely short system call that resets the sound DMA. The timing is extremely critical, so call this function immediately after the V-Blank interrupt every 1/60 second.
No parameters, no return value.

SWI 40 (28h) - SoundDriverVSyncOff
Due to problems with the main program if the V-Blank interrupts are stopped, and SoundDriverVSync cannot be called every 1/60 a second, this function must be used to stop sound DMA.
Otherwise, even if you exceed the limit of the buffer the DMA will not stop and noise will result.
No parameters, no return value.

SWI 41 (29h) - SoundDriverVSyncOn
This function restarts the sound DMA stopped with the previously described SoundDriverVSyncOff.
After calling this function, have a V-Blank occur within 2/60 of a second and call SoundDriverVSync.
No parameters, no return value.

Unpredictable Things

Forward
Most of the below is caused by 'traces' from previous operations which have used the databus. No promises that the results are stable on all current or future GBA models, and/or under all temperature and

interference circumstances.
Also, below specifies 32bit data accesses only. When reading units less than 32bit, data is rotated depending on the alignment of the originally specified address, and 8bit or 16bit are then isolated from the 32bit value as usually.

Reading from BIOS Memory (00000000-00003FFF)
The BIOS memory is protected against reading, the GBA allows to read opcodes or data only if the program counter is located inside of the BIOS area. If the program counter is not in the BIOS area, reading will return the most recent successfully fetched BIOS opcode (eg. the opcode at [00DCh+8] after startup, the opcode at [013Ch+8] after IRQ execution, or the opcode at [0188h+8] after SWI execution).

Reading from Unused Memory (00004000-1FFFFFF0,10000000-FFFFFFFF)
Accessing unused memory returns the recently pre-fetched opcode, ie. the 32bit opcode at $+8 in ARM state, or the 16bit-opcode at $+4 in THUMB state, in the later case the 16bit opcode is mirrored across both upper/lower 16bits of the returned 32bit data.
Note: This is caused by the prefetch pipeline in the CPU itself, not by the external gamepak prefetch, ie. it works for code in RAM as well.

Reading from Unused or Write-Only I/O Ports
Works like above unused memory when the entire 32bit memory fragment is Unused (eg. 0E0h) and/or Write-Only (eg. DMA0SAD). And otherwise, returns zero if the lower 16bit fragment is readable (eg. 04C=MOSAIC, 04E=NOTUSED/ZERO).

Reading from GamePak ROM when no Cartridge is inserted
Because Gamepak uses the same signal-lines for both 16bit data and for lower 16bit halfword address, the entire gamepak ROM area is effectively filled by incrementing 16bit values (Address/2 AND FFFFh).

Memory Mirrors
Most internal memory is mirrored across the whole 24bit/16MB address space in which it is located: On-board RAM at 2XXXXXX, In-Chip RAM at 3XXXXXXh, Palette RAM at 5XXXXXXh, VRAM at 6XXXXXXh, and OAM at 7XXXXXXh. Even though VRAM is sized 96K (64K+32K), it is repeated in steps of 128K (64K+32K+32K, the two 32K blocks itself being mirrors of each other).
BIOS ROM, Normal ROM Cartridges, and I/O area are NOT mirrored, the only exception is the undocumented I/O port at 4000800h (repeated each 64K).
The 64K SRAM area is mirrored across the whole 32MB area at E000000h-FFFFFFFh, also, inside of the 64K SRAM field, 32K SRAM chips are repeated twice.

External Connectors

External Connectors
AUX Game Pak Bus
AUX Link Port
AUX Sound/Headphone Socket
AUX Battery/Power Supply

Getting access to Internal Pins
AUX Opening the GBA
AUX Mainboard

Multiboot Cable
AUX Multiboot PC-to-GBA Cable
AUX Burst Boot Backdoor

AUX Game Pak Bus

Game Pak Bus - 32pin cartridge slot
The cartridge bus may be used for both CGB and GBA game paks. In GBA mode, it is used as follows:

Pin	Name	Dir	Expl.
1	VDD	O	Power Supply 3.3V DC
2	PHI	O	System Clock (selectable none, 4.19MHz, 8.38MHz, 16.76MHz)
3	/WR	O	Write Select
4	/RD	O	Read Select
5	/CS	O	ROM Chip Select
6-21	AD0-15	I/O	lower 16bit Address and/or 16bit ROM-data (see below)
22-29	A16-23	I/O	upper 8bit ROM-Address or 8bit SRAM-data (see below)
30	/CS2	O	SRAM Chip Select
31	/REQ	I	Interrupt request (/IREQ) or DMA request (/DREQ)
32	GND	O	Ground 0V

When accessing game pak SRAM, a 16bit address is output through AD0-AD15, then 8bit of data are transferred through A16-A23.
When accessing game pak ROM, a 24bit address is output through AD0-AD15 and A16-A23, then 16bit of data are transferred through AD0-AD15. The 24bit address is formed from the actual 25bit memory address (byte-steps), divided by two by two (halfword-steps).

8bit-Gamepak-Switch
A small switch is located inside of the cartridge slot, the switch is pushed down when an 8bit cartridge is inserted, it is released when a GBA cartridge is inserted (or if no cartridge is inserted).
The switch mechanically controls whether VDD3 or VDD5 are output at VDD35; ie. in GBA mode 3V power supply/signals are used for the cartridge slot and link port, while in 8bit mode 5V are used.
Also, the current state of the switch can be read-out in GBA mode from Port 204h (WAITCNT). The GBA boot procedure in BIOS uses this to detect 8bit carts and to set the GBA into 8bit mode.
In 8bit mode, the cartridge bus works much like for GBA SRAM, however, the 8bit /CS signal is expected at Pin 5, while GBA SRAM /CS2 at Pin 30 is interpreted as /RESET signal by the 8bit MBC chip (if any). In practice, this appears to result in 00h being received as data when attempting to read-out 8bit cartridges from inside of GBA mode.

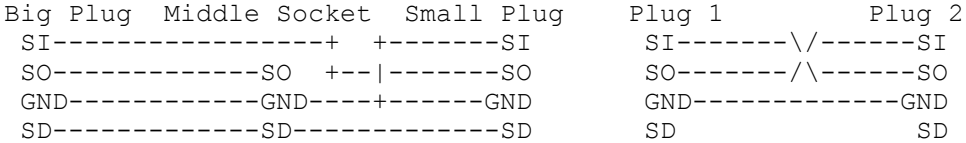
AUX Link Port

Serial Link Port Pin-Out

Pin	Name	Cable
1	VDD35	N/A
2	SO	Red
3	SI	Orange
4	SD	Brown
5	SC	Green
6	GND	Blue
Shield		Shield

The pins are arranged from left to right as 2,4,6 in upper row, and 1,3,5 in lower row (outside view of GBA socket; flat side of socket upside). Note: the pin numbers and names are printed on the GBA mainboard, colors as used in Nintendos AGB-005 and older 8bit cables.

Cable Diagrams (Left: GBA Cable, Right: 8bit Gameboy Cable)



```
SC-------------SC-------------SC       SC---------------SC
Shield-------Shield-------Shield       Shield-------Shield
```

Normal Connection
Just connect the plugs to the two GBAs and leave the Middle Socket disconnected, in this mode both GBAs may behave as master or slave, regardless of whether using big or small plugs.
The GBA is (NOT ???) able to communicate in Normal mode with MultiPlay cables which do not have crossed SI/SO lines.

Multi-Play Connection
Connect two GBAs as normal, for each further GBAs connect an additional cable to the Middle socket of the first (or further) cable(s), up to four GBAs may be connected by using up to three cables.
The GBA which is connected to a Small Plug is master, the slaves are all connected to Large Plugs. (Only small plugs fit into the Middle Socket, so it's not possible to mess up something here).

Multi-Boot Connection
MultiBoot (SingleGamepak) is typically using Multi-Play communication, in this case it is important that the Small plug is connected to the master/sender (ie. to the GBA that contains the cartridge).

Non-GBA Mode Connection
First of all, it is not possible to link between 32bit GBA games and 8bit games, parts because of different cable protocol, and parts because of different signal voltages.
However, when a 8bit cartridge is inserted (the GBA is switched into 8bit compatibility mode) it may be connected to other 8bit games (monochrome gameboys, CGBs, or to other GBAs which are in 8bit mode also, but not to GBAs in 32bit mode).
When using 8bit link mode, an 8bit link cable must be used. The GBA link cables won't work, see below modification though.

Using a GBA 32bit cable for 8bit communication
Open the middle socket, and disconnect Small Plugs SI from GND, and connect SI to Large Plugs SO instead. You may also want to install a switch that allows to switch between SO and GND, the GND signal should be required for MultiPlay communication only though.
Also, cut off the plastic ledge from the plugs so that they fit into 8bit gameboy sockets.

Using a GBA 8bit cable for 32bit communication
The cable should theoretically work as is, as the grounded SI would be required for MultiPlay communication only. However, software that uses SD for Slave-Ready detection won't work unless when adding a SD-to-SD connection (the 8bit plugs probably do not even contain SD pins though).

AUX Sound/Headphone Socket

Stereo Sound Connector (3.5mm, female)

Pin	Expl.
Tip	Sound Left
Middle	Sound Right
Base	Ground

AUX Battery/Power Supply

Power Supply Input

The GBA does not have a separate power supply input, however external power supplies can be connected into the battery socket. The recommended external voltage is 3.3V DC - that is then used as is, or internally lowered to 3V by the battery socket adapter ???

Using PC +5V DC as Power Supply
Developers whom are using a PC for GBA programming will probably want to use the PC power supply (gained from disk drive power supply cable) for the GBA as well rather than dealing with batteries or external power supplies.
To lower the voltage to approximately 3 Volts use two (or three) diodes (type 1N 4004 or similiar) connected as such:

```
PC +5V (red)   --------|>|---|>|--------  GBA BT+
PC GND (black) ------------------------  GBA BT-
```

When using only 2 diodes voltage is a bit too high, when using 3 diodes voltage is a bit too low, in the later case power led may glow red when using battery hungry flashcards, both should work okay though, no warranty.
Note: The ring printed onto the diodes points to the GBA side.

Using Parallel Port Data Lines as Power Supply
When using a Multiboot cable connected to PC parallel port it'd be comfortable to use the existing cable for power supply as well. This method definetely exceeds all official ratings, in fact almost, it is ethically wrong, absolutely no warranty that it will work and that it won't destroy hardware and/or burn down the house, USE AT OWN RISK, among others the following may cause problems:
The parallel port data signals aren't intended to be mis-used as power supply, many BIOSes and operating systems will initialize some data lines as LOW and some as HIGH, when directly shortcutting data lines any low signals will most likely forcefully pull-down high signals, different parallel ports output between 3.66V and 5.00V data HIGH, and not all parallel ports will provide enough Watts even when all data lines are high.
The most violent method is to shortcut all Data Lines (Pin 2-9) and connect these to the GBA BT+ input, also connect Ground (Pin 19 or else) to GBA BT- input (unless already connected to GBA link port GND). If necessary insert 1 or 2 diodes (depending on the parallel port type) between Data Lines and BT+ (as describe for 5V input above). The above Data/Ground pin numbers are the same for 25-pin PC sockets and 36-pin printer plugs.
I've tried above with 4 different parallel ports with different results: One didn't worked at all (not enough power), one worked even though power LED signalized low power, another worked just fine, and the fourth worked only after inserting two diodes. Note that the GBA would consume even more power when inserting cartridges into it.
Finally, to turn the power on, output FFh to parallel port base address, and if necessary output 0Bh to base address+2. (Or launch the no$gba multiboot function which automatically initializes these ports).

Minimum and Maximum Ratings
Too less voltage: The GBA does not work with 2V, the display will remain blank, the power LED will be flashing, and the speaker will produce scratching noise. Too high voltage: When using 5V, the battery LED will be perfectly green, but the GBA refuses to work, display remains blank.
Both doesn't seem to destroy the hardware, however, either one doesn't work, so don't mess around with it.

AUX Opening the GBA

Since Nintendo uses special screws with Y-shaped heads to seal the GBA (as well as older 8bit gameboys), it's always a bit difficult to loosen these screws.

Using Screwdrivers

One possible method is to use a small flat screwdriver, which might work, even though it'll most likely damage the screwdriver.
Reportedly, special Y-shaped screwdrivers for gameboys are available for sale somewhere (probably not at your local dealer, but you might find some in the internet or elsewhere).

Destroying the Screws
A more violent method is to take an electric drill, and drill-off the screw heads, this might also slightly damage the GBA plastic chase, also take care that the metal spoons from the destroyed screws don't produce shortcuts on the GBA mainboard.

Using a selfmade Screwdriver
A possible method is to take a larger screw (with a normal I-shaped, or X-shaped head), and to cut the screw-tip into Y-shape, you'll then end up with an "adapter" which can be placed in the middle between a normal scrwdriver and gameboy screws.
Preferably, first cut the screw-tip into a shape like a "sharp three sided pyramid", next cut notches into each side. Access to a grinding-machine will be a great benefit, but you might get it working by using a normal metal-file as well.

AUX Mainboard

Other possibly useful signals on the mainboard...

FIQ Signal
The FIQ (Fast Interrupt) signal (labeled FIQ on the mainboard) could be used as external interrupt (or debugging break) signal.
Caution: By default, the FIQ input is directly shortcut to VDD35 (+3V or +5V power supply voltage), this can be healed by scratching off the CL1 connection located close to the FIQ pin (FIQ still appears to have an internal pull-up, so that an external resistor is not required).
The GBA BIOS rejects FIQs if using normal ROM cartridge headers (or when no cartridge is inserted). When using a FIQ-compatible ROM header, Fast Interrupts can be then requested by pulling FIQ to ground, either by a push button, or by remote controlled signals.

RESET Signal
The RESET signal (found on the mainboard) could be used to reset the GBA by pulling the signal to ground for a few microseconds (or longer). The signal can be directly used (it is not shortcut to VDD35, unlike FIQ).
Note: A reset always launches nintendos time-consuming and annoying boot/logo procedure, so that it'd be recommend to avoid this "feature" when possible.

Joypad Signals
The 10 direction/button signals are each directly shortcut to ground when pressed, and pulled up high otherwise (unlike 8bit gameboys which used a 2x4 keyboard matrix), it'd be thus easy to connect a remote keyboard, keypad, joypad, or read-only 12bit parallel port.

AUX Multiboot PC-to-GBA Cable

Below describes how to connect a PC parallel port to the GBA link port, allowing to upload small programs (max 256 KBytes) from no$gba's Utility menu into real GBAs.

This is possible because the GBA BIOS includes a built-in function for downloading & executing program code even when no cartridge is inserted. The program is loaded to 2000000h and up in GBA memory, and

must contain cartridge header information just as for normal ROM cartridges (nintendo logo, checksum, etc., plus some additional multiboot info).

Basic Cable Connection
The general connection is very simple (only needs four wires), the only problem is that you need a special GBA plug or otherwise need to solder wires directly to the GBA mainboard (see Examples below).

```
GBA  Name  Color                SUBD CNTR Name
2    SO    Red    ------------- 10   10   /ACK
3    SI    Orange ------------- 14   14   /AUTOLF
5    SC    Green  ------------- 1    1    /STROBE
6    GND   Blue   ------------- 19   19   GND
```

Optionally, also connect the following signals (see notes below):

```
4    SD    Brown  ------------- 17   36   /SELECT  (double speed burst)
-    -     -       +----------- 2..9 2..9 D0..7    (pull-up)
-    -     -       |---[===]--- 14   14   /AUTOLF  (pull-up)
-    -     -       |---[===]--- 1    1    /STROBE  (pull-up)
-    -     -       +---[===]--- 17   36   /SELECT  (pull-up)
RESET (mainboard) ------|>|---- 16   31   /INIT    (automatic reset)
```

Notes: The GBA Pins are arranged from left to right as 2,4,6 in upper row, and 1,3,5 in lower row; outside view of GBA socket; flat side of socket upside. The above "Colors" are as used in most or all standard Nintendo link cables, note that Red/Orange will be exchanged at one end in cables with crossed SO/SI lines. At the PC side, use the SUBD pin numbers when connecting to a 25-pin SUBD plug, or CNTR pin numbers for 36-pin Centronics plug.

Optional SD Connection (Double Speed Burst)
The SD line is used for Double Speed Burst transfers only, in case that you are using a gameboy link plug for the connection, and if that plug does not have a SD-pin (such like from older 8bit gameboy cables), then you may leave out this connection. Burst Boot will then only work half as fast though.

Optional Pull-Ups (Improves Low-to-High Transition Speed)
If your parallel port works only with medium or slow delay settings, try to connect 570 Ohm resistors to each of the strobe/autolf/select outputs, and the other resistor pin to any or all of the parallel port data lines (no$gba outputs high to pins 2..9).

Optional Reset Connection
The Reset connection allows to reset & upload data even if a program in the GBA has locked up (or if you've loaded a program that does not support nocash burst boot). - Without reset connection you'd then manually have to reset the GBA by switching it off and on.
The RESET signal is labeled as such on the GBA mainboard. The diode (1N4148 or similiar) is required because otherwise strong INIT signals would pull-up the RESET signal, preventing the GBA from automatically resetting itself when switched on.

Optional Power Supply Connection
Also, you may want to connect the power supply to parallel port data lines, see chapter Power Supply for details.

Transmission Speed
The first transfer will be very slow, and the GBA BIOS will display the boot logo for at least 4 seconds, even if the transfer has completed in less time. Once when you have uploaded a program with burst boot backdoor, further transfers will be ways faster. The table below shows transfer times for 0KByte - 256KByte files:

```
Boot Mode_____Delay 0_______Delay 1_______Delay 2_____
Double Burst  0.1s - 1.8s   0.1s - 3.7s   0.1s - 5.3s
Single Burst  0.1s - 3.6s   0.1s - 7.1s   0.1s - 10.6s
Normal Bios   4.0s - 9.0s   4.0s - 12.7s  4.0s - 16.3s
```

All timings measured on a 66MHz computer, best possible transmission speed should be 150KBytes/second. Timings might slightly vary depending on the CPU speed and/or operating system. Synchronization is done by I/O waitstates, that should work even on faster computers. Non-zero delays are eventually required for cables without pull-ups.

Requirements
Beside for the cable and plugs, no special requirements.
The cable should work with all parallel ports, including old-fashined uni-directional printer ports, as well as modern bi-directional EPP ports. Transfer timings should work stable regardless of the PCs CPU speed (see above though), and regardless of multitasking interruptions.
Both no$gba and the actual transmission procedure are using some 32bit code, so that either one currently requires 80386SX CPUs or above.

Windows NT/2000/etc.
NT/2000/etc. prevent to access parallel ports directly, this problem can be reportedly healed by using special drivers (such like giveio, totalio, or userport), which would be possibly required to be called from inside of no$gba. If anybody can supply information on where to download & how to use these drivers, please let me know!
Note: Windows 95/98/etc. are working fine without such drivers.

Connection Examples
As far as I can imagine, there are four possible methods how to connect the cable to the GBA. The first two methods don't require to open the GBA, and the other methods also allow to connect optional power supply and reset signal.

```
1) Connect it to the GBA link port. Advantage: No need to
   open/modify the GBA. Disadvantage: You need a special plug,
   (typically gained by removing it from a gameboy link cable).
2) Solder the cable directly to the GBA link port pins. Advantages:
   No plug required & no need to open the GBA. Disadvantages:
   You can't remove the cable, and the link port becomes unusable.
3) Solder the cable directly to the GBA mainboard. Advantage: No
   plug required at the GBA side. Disadvantage: You'll always
   have a cable leaping out of the GBA even when not using it,
   unless you put a small standard plug between GBA and cable.
4) Install a Centronics socket in the GBA (between power switch
   and headphone socket). Advantage: You can use a standard
   printer cable. Disadvantages: You need to cut a big hole into
   the GBAs battery box (which cannot be used anymore), the big
   cable might be a bit uncomfortable when holding the GBA.
```

Personally, I've decided to use the lastmost method as I don't like ending up with hundreds of special cables for different purposes, and asides, it's been fun to damage the GAB as much as possible.

Note
The above used PC parallel port signals are typically using 5V=HIGH while GBA link ports deal with 3V=HIGH. From my experiences, the different voltages do not cause communication problems (and do not damage the GBA and/or PC hardware), and after all real men don't care about a handful of volts, however, use at own risk.

AUX Burst Boot Backdoor

When writing multiboot compatible programs, always include a burst boot "backdoor", this will allow yourself (and other people) to upload programs much faster as when using the normal GBA BIOS multiboot function. Aside from the improved transmission speed, there's no need to reset the GBA each time (eventually manually if you do not have reset connect), and, most important, the time-consuming nintendo-logo intro is bypassed.

The Burst Boot Protocol

In your programs IRQ handler, add some code that watches out for burst boot IRQ requests. When sensing a burst boot request, download the actual boot procedure, and pass control to that procedure.

```
Send (PC)     Reply (GBA)
"BRST"        "BOOT"         ;request burst, and reply <prepared> for boot
<wait 1/16s>  <process IRQ>  ;long delay, allow slave to enter IRQ handler
llllllll      "OKAY"         ;send length in bytes, reply <ready> to boot
dddddddd      --------       ;send data in 32bit units, reply don't care
cccccccc      cccccccc       ;exchange crc (all data units added together)
```

Use normal mode, 32bit, external clock for all transfers. The received highspeed loader (currently approx. 180h bytes) is to be loaded to and started at 3000000h, which will then handle the actual download operation.

Below is an example program which works with multiboot, burstboot, and as normal rom/flashcard. The source can be assembled with a22i (the no$gba built-in assembler, see no$gba utility menu). When using other/mainstream assemblers, you'll eventually have to change some directives, convert numbers from NNNh into 0xNNN format, and define the origin somewhere in linker/makefile instead of in source code.

```
.arm            ;select 32bit ARM instruction set
.gba            ;indicate that it's a gameboy advance program
.fix            ;automatically fix the cartridge header checksum
org 2000000h    ;origin in RAM for multiboot-cable/no$gba-cutdown programs
;------------------
;cartridge header/multiboot header
 b     rom_start                ;-rom entry point
 dcb   ...insert logo here...   ;-nintento logo (156 bytes)
 dcb   'XBOO SAMPLE '           ;-title (12 bytes)
 dcb   0,0,0,0,  0,0            ;-game code (4 bytes), maker code (2 bytes)
 dcb   96h,0,0                  ;-fixed value 96h, main unit code, device type
 dcb   0,0,0,0,0,0,0            ;-reserved (7 bytes)
 dcb   0                        ;-software version number
 dcb   0                        ;-header checksum (set by .fix)
 dcb   0,0                      ;-reserved (2 bytes)
 b     ram_start                ;-multiboot ram entry point
 dcb   0,0                      ;-multiboot reserved bytes (destroyed by BIOS)
 dcb   0,0                      ;-blank padded (32bit alignment)
;------------------
irq_handler:  ;interrupt handler (note: r0-r3 are pushed by BIOS)
 mov    r1,4000000h             ;\get I/O base address,
 ldr    r0,[r1,200h] ;IE/IF     ; read IE and IF,
 and    r0,r0,r0,lsr 16         ; isolate occurred AND enabled irqs,
 add    r3,r1,200h   ;IF        ; and acknowledge these in IF
 strh   r0,[r3,2]               ;/
 ldrh   r3,[r1,-8]              ;\mix up with BIOS irq flags at 3007FF8h,
 orr    r3,r3,r0                ; aka mirrored at 3FFFFF8h, this is required
 strh   r3,[r1,-8]              ;/when using the (VBlank-)IntrWait functions
 and    r3,r0,80h ;IE/IF.7 SIO  ;\
 cmp    r3,80h                  ; check if it's a burst boot interrupt
 ldreq  r2,[r1,120h] ;SIODATA32 ; (if interrupt caused by serial transfer,
 ldreq  r3,[msg_brst]           ; and if received data is "BRST",
 cmpeq  r2,r3                   ; then jump to burst boot)
 beq    burst_boot              ;/
 ;... insert your own interrupt handler code here ...
 bx     lr                      ;-return to the BIOS interrupt handler
;------------------
burst_boot:     ;requires incoming r1=4000000h
 ;... if your program uses DMA, disable any active DMA transfers here ...
 ldr   r4,[msg_okay]            ;\
 bl    sio_transfer             ; receive transfer length/bytes & reply "OKAY"
 mov   r2,r0 ;len               ;/
 mov   r3,3000000h   ;dst       ;\
 mov   r4,0  ;crc               ;
@@lop:                          ;
```

```
 bl    sio_transfer             ; download burst loader to 3000000h and up
 stmia [r3]!,r0      ;dst       ;
 add   r4,r4,r0      ;crc       ;
 subs  r2,r2,4       ;len       ;
 bhi   @@lop                    ;/
 bl    sio_transfer             ;-send crc value to master
 b     3000000h  ;ARM state!    ;-launch actual transfer / start the loader
;------------------
sio_transfer:  ;serial transfer subroutine, 32bit normal mode, external clock
 str   r4,[r1,120h]  ;siodata32 ;-set reply/send data
 ldr   r0,[r1,128h]  ;siocnt    ;\
 orr   r0,r0,80h                ; activate slave transfer
 str   r0,[r1,128h]  ;siocnt    ;/
@@wait:                         ;\
 ldr   r0,[r1,128h]  ;siocnt    ; wait until transfer completed
 tst   r0,80h                   ;
 bne   @@wait                   ;/
 ldr   r0,[r1,120h]  ;siodata32 ;-get received data
 bx    lr
;---
msg_boot dcb 'BOOT'     ;\
msg_okay dcb "OKAY"     ; ID codes for the burstboot protocol
msg_brst dcb "BRST"     ;/
;------------------
download_rom_to_ram:
 mov  r0,8000000h  ;src/rom     ;\
 mov  r1,2000000h  ;dst/ram     ;
 mov  r2,40000h/16 ;length      ; transfer the ROM content
@@lop:                          ; into RAM (done in units of 4 words/16 bytes)
 ldmia [r0]!,r4,r5,r6,r7        ; currently fills whole 256K of RAM,
 stmia [r1]!,r4,r5,r6,r7        ; even though the proggy is smaller
 subs  r2,r2,1                  ;
 bne   @@lop                    ;/
 sub   r15,lr,8000000h-2000000h ;-return (retadr rom/8000XXXh -> ram/2000XXXh)
;------------------
init_interrupts:
 mov  r4,4000000h               ;-base address for below I/O registers
 ldr  r0,=irq_handler           ;\install IRQ handler address
 str  r0,[r4,-4]   ;IRQ HANDLER ;/at 3FFFFFC aka 3007FFC
 mov  r0,0008h                  ;\enable generating vblank irqs
 strh r0,[r4,4h]   ;DISPSTAT    ;/
 mrs  r0,cpsr                   ;\
 bic  r0,r0,80h                 ; cpu interrupt enable (clear i-flag)
 msr  cpsr,r0                   ;/
 mov  r0,0                      ;\
 str  r0,[r4,134h] ;RCNT        ; init SIO normal mode, external clock,
 ldr  r0,=5080h                 ; 32bit, IRQ enable, transfer started
 str  r0,[r4,128h] ;SIOCNT      ; output "BOOT" (indicate burst boot prepared)
 ldr  r0,[msg_boot]             ;
 str  r0,[r4,120h] ;SIODATA32   ;/
 mov  r0,1                      ;\interrupt master enable
 str  r0,[r4,208h] ;IME=1       ;/
 mov  r0,81h                    ;\enable execution of vblank IRQs,
 str  r0,[r4,200h] ;IE=81h      ;/and of SIO IRQs (burst boot)
 bx   lr
;------------------
rom_start:   ;entry point when booted from flashcart/rom
 bl   download_rom_to_ram       ;-download ROM to RAM (returns to ram_start)
ram_start:   ;entry point for multiboot/burstboot
 mov  r0,0feh                   ;\reset all registers, and clear all memory
 swi  10000h ;RegisterRamReset  ;/(except program code in wram at 2000000h)
 bl   init_interrupts           ;-install burst boot irq handler
 mov  r4,4000000h               ;\enable video,
 strh r4,[r4,000h] ;DISPCNT     ;/by clearing the forced blank bit
@@mainloop:
 swi  50000h ;VBlankIntrWait    ;-wait one frame (cpu in low power mode)
 mov  r5,5000000h               ;\increment the backdrop palette color
 str  r8,[r5]                   ; (ie. display a blinking screen)
 add  r8,r8,1                   ;/
```

```
 b    @@mainloop
;------------------
.pool
end
```

CPU Reference

General ARM7TDMI Information
CPU Overview
CPU Register Set
CPU Flags
CPU Exceptions

The ARM7TDMI Instruction Sets
THUMB Instruction Set
ARM Instruction Set
Pseudo Instructions and Directives

Further Information
CPU Instruction Cycle Times
CPU Data Sheet

CPU Overview

The ARM7TDMI is a 32bit RISC (Reduced Instruction Set Computer) CPU, designed by ARM (Advanced RISC Machines), and designed for both high performance and low power consumption.

Fast Execution
Depending on the CPU state, all opcodes are sized 32bit or 16bit (that's counting both the opcode bits and its parameters bits) providing fast decoding and execution. Additionally, pipelining allows - (a) one instruction to be executed while (b) the next instruction is decoded and (c) the next instruction is fetched from memory - all at the same time.

Data Formats
The CPU manages to deal with 8bit, 16bit, and 32bit data, that are called:

```
  8bit - Byte
16bit - Halfword
32bit - Word
```

The two CPU states
As mentioned above, two CPU states exist:
- ARM state: Uses the full 32bit instruction set (32bit opcodes)
- THUMB state: Uses a cutdown 16bit instruction set (16bit opcodes)
Regardless of the opcode-width, both states are using 32bit registers, allowing 32bit memory addressing as well as 32bit arithmetic/logical operations.

When to use ARM state
Basically, there are two advantages in ARM state:

```
- Each single opcode provides more functionality, resulting
  in faster execution when using a 32bit bus memory system
  (such like opcodes stored in GBA Work RAM).
```

```
- All registers R0-R15 can be accessed directly.
```

The downsides are:

```
- Not so fast when using 16bit memory system
  (but it still works though).
- Program code occupies more memory space.
```

When to use THUMB state
There are two major advantages in THUMB state:

```
- Faster execution up to approx 160% when using a 16bit bus
  memory system (such like opcodes stored in GBA GamePak ROM).
- Reduces code size, decreases memory overload down to approx 65%.
```

The disadvantages are:

```
- Not as multi-functional opcodes as in ARM state, so it will
  be sometimes required use more than one opcode to gain a
  similiar result as for a single opcode in ARM state.
- Most opcodes allow only registers R0-R7 to be used directly.
```

Combining ARM and THUMB state
Switching between ARM and THUMB state is done by a normal branch (BX) instruction which takes only a handful of cycles to execute (allowing to change states as often as desired - with almost no overload).

Also, as both ARM and THUMB are using the same register set, it is possible to pass data between ARM and THUMB mode very easily.

The best memory & execution performance can be gained by combining both states: THUMB for normal program code, and ARM code for timing critical subroutines (such like interrupt handlers, or complicated algorithms).

Note: ARM and THUMB code cannot be executed simultaneously.

Automatic state changes
Beside for the above manual state switching by using BX instructions, the following situations involve automatic state changes:
- CPU switches to ARM state when executing an exception
- User switches back to old state when leaving an exception

CPU Register Set

Overview
The following table shows the ARM7TDMI register set which is available in each mode. There's a total of 37 registers (32bit each), 31 general registers (Rxx) and 6 status registers (xPSR).
Note that only some registers are 'banked', for example, each mode has it's own R14 register: called R14, R14_fiq, R14_svc, etc. for each mode respectively.
However, other registers are not banked, for example, each mode is using the same R0 register, so writing to R0 will always affect the content of R0 in other modes also.

System/User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3

```
R4          R4        R4         R4        R4        R4
R5          R5        R5         R5        R5        R5
R6          R6        R6         R6        R6        R6
R7          R7        R7         R7        R7        R7
-------------------------------------------------------------
R8          R8_fiq    R8         R8        R8        R8
R9          R9_fiq    R9         R9        R9        R9
R10         R10_fiq   R10        R10       R10       R10
R11         R11_fiq   R11        R11       R11       R11
R12         R12_fiq   R12        R12       R12       R12
R13 (SP)    R13_fiq   R13_svc    R13_abt   R13_irq   R13_und
R14 (LR)    R14_fiq   R14_svc    R14_abt   R14_irq   R14_und
R15 (PC)    R15       R15        R15       R15       R15
-------------------------------------------------------------
CPSR        CPSR      CPSR       CPSR      CPSR      CPSR
--          SPSR_fiq  SPSR_svc   SPSR_abt  SPSR_irq  SPSR_und
-------------------------------------------------------------
```

R0-R12 Registers (General Purpose Registers)
These thirteen registers may be used for whatever general purposes. Basically, each is having same functionality and performance, ie. there is no 'fast accumulator' for arithmetic operations, and no 'special pointer register' for memory addressing.
However, in THUMB mode only R0-R7 (Lo registers) may be accessed freely, while R8-R12 and up (Hi registers) can be accessed only by some instructions.

R13 Register (SP)
This register is used as Stack Pointer (SP) in THUMB state. While in ARM state the user may decided to use R13 and/or other register(s) as stack pointer(s), or as general purpose register.
As shown in the table above, there's a separate R13 register in each mode, and (when used as SP) each exception handler may (and MUST!) use its own stack.

R14 Register (LR)
This register is used as Link Register (LR). That is, when calling to a sub-routine by a Branch with Link (BL) instruction, then the return address (ie. old value of PC) is saved in this register.
Storing the return address in the LR register is obviously faster than pushing it into memory, however, as there's only one LR register for each mode, the user must manually push its content before issuing 'nested' subroutines.
Same happens when an exception is called, PC is saved in LR of new mode.
Note: In ARM mode, R14 may be used as general purpose register also, provided that above usage as LR register isn't required.

R15 Register (PC)
R15 is always used as program counter (PC). Note that when reading R15, this will usually return a value of PC+nn because of read-ahead (pipelining), whereas 'nn' depends on the instruction and on the CPU state (ARM or THUMB).

CPSR and SPSR (Program Status Registers)
The current condition codes (flags) and CPU control bits are stored in the CPSR register. When an exception arises, the old CPSR is saved in the SPSR of the respective exception-mode (much like PC is saved in LR).
For details refer to chapter about CPU Flags.

CPU Flags

Current Program Status Register (CPSR)

```
Bit   Expl.
31    N - Sign Flag      (0=Not Signed, 1=Signed)
30    Z - Zero Flag      (0=Not Zero, 1=Zero)
29    C - Carry Flag     (0=No Carry, 1=Carry)
28    V - Overflow Flag  (0=No Overflow, 1=Overflow)
27-8  Reserved           (For future use) - Do not change manually!
7     I - IRQ disable    (0=Enable, 1=Disable)
6     F - FIQ disable    (0=Enable, 1=Disable)
5     T - State Bit      (0=ARM, 1=THUMB) - Do not change manually!
4-0   M4-M0 - Mode Bits  (See below)
```

Bit 31-28: Condition Code Flags (N,Z,C,V)
These bits reflect results of logical or arithmetic instructions. In ARM mode, it is often optionally whether an instruction should modify flags or not, for example, it is possible to execute a SUB instruction that does NOT modify the condition flags.
In ARM state, all instructions can be executed conditionally depending on the settings of the flags, such like MOVEQ (Move if Z=1). While In THUMB state, only Branch instructions (jumps) can be made conditionally.

Bit 27-8: Reserved Bits
These bits are reserved for possible future implementations. For best forwards compatibility, the user should never change the state of these bits, and should not expect these bits to be set to a specific value.

Bit 7-0: Control Bits (I,F,T,M4-M0)
These bits may change when an exception occurs. In privileged modes (non-user modes) they may be also changed manually.
The interrupt bits I and F are used to disable IRQ and FIQ interrupts respectively (a setting of "1" means disabled).
The T Bit signalizes the current state of the CPU (0=ARM, 1=THUMB), this bit should never be changed manually - instead, changing between ARM and THUMB state must be done by BX instructions.
The Mode Bits M4-M0 contain the current operating mode.

```
Binary Hex Dec  Expl.
10000b 10h 16 - User (non-privileged)
10001b 11h 17 - FIQ
10010b 12h 18 - IRQ
10011b 13h 19 - Supervisor (SWI)
10111b 17h 23 - Abort
11011b 1Bh 27 - Undefined
11111b 1Fh 31 - System (privileged 'User' mode)
```

Writing any other values into the Mode bits is not allowed.

Saved Program Status Registers (SPSR_<mode>)
Additionally to above CPSR, five Saved Program Status Registers exist:
SPSR_fiq, SPSR_svc, SPSR_abt, SPSR_irq, SPSR_und
Whenever the CPU enters an exception, the current status register (CPSR) is copied to the respective SPSR_<mode> register. Note that there is only one SPSR for each mode, so nested exceptions inside of the same mode are allowed only if the exception handler saves the content of SPSR in memory.
For example, for an IRQ exception: IRQ-mode is entered, and CPSR is copied to SPSR_irq. If the interrupt handler wants to enable nested IRQs, then it must first push SPSR_irq before doing so.

CPU Exceptions

Exceptions are caused by interrupts or errors. In the ARM7TDMI the following exceptions may arise, sorted by priority, starting with highest priority:
- Reset

- Data Abort
- FIQ
- IRQ
- Prefetch Abort
- Software Interrupt
- Undefined Instruction

Exception Vectors
The following are the exception vectors in memory. That is, when an exception arises, CPU is switched into ARM state, and the program counter (PC) is loaded by the respective address.

```
Address     Exception                   Mode on Entry
00000000h   Reset                       Supervisor (_svc)
00000004h   Undefined Instruction       Undefined  (_und)
00000008h   Software Interrupt (SWI)    Supervisor (_svc)
0000000Ch   Prefetch Abort              Abort      (_abt)
00000010h   Data Abort                  Abort      (_abt)
00000014h   (Reserved)                  -          -
00000018h   Normal Interrupt (IRQ)      IRQ        (_irq)
0000001Ch   Fast Interrupt   (FIQ)      FIQ        (_fiq)
```

As there's only space for one ARM opcode at each of the above addresses, it'd be usually recommended to deposit a Branch opcode into each vector, which'd then redirect to the actual exception handlers address.

Actions performed by CPU when entering an exception

```
- R14=PC+nn              ;save old PC, ie. return address
- SPSR_<new mode>=CPSR   ;save old flags
- CPSR new T,M bits      ;set to T=0 (ARM state), and M4-0=new mode
- CPSR new I,F bits      ;depends of type of exception
  ;for FIQ: F=???,I=1, and for IRQ: F=same,I=1
- PC=exception_vector    ;see table above
```

Above "PC+nn" dependends on the type of exception. Basically, in ARM state that nn-offset is caused by pipelining, and in THUMB state an identical ARM-style 'offset' is generated (even though the 'base address' may be only halfword-aligned).

Required user-handler actions when returning from an exception
Restore any general registers (R0-R14) which might have been modified by the exception handler. Use return-instruction as listed in the respective descriptions below, this will both restore PC and CPSR - that automatically involves that the old CPU state (THUMB or ARM) as well as old state of FIQ and IRQ disable flags are restored.
As mentioned above (see action on entering...), the return address is always saved in ARM-style format, so that exception handler may use the same return-instruction, regardless of whether the exception has been generated from inside of ARM or THUMB state.

FIQ (Fast Interrupt Request)
This interrupt is generated by a LOW level on the nFIQ input. It is supposed to process timing critical interrupts at a high priority, as fast as possible.
Additionally to the common banked registers (R13_fiq,R14_fiq), five extra banked registers (R8_fiq-R12_fiq) are available in FIQ mode. The exception handler may freely access these registers without modifying the main programs R8-R12 registers (and without having to save that registers on stack).
Upon FIQ, the I Bit of the Normal Interrupts (IRQ disable) is automatically set, providing that Fast Interrupts cannot be interrupted by Normal IRQs which have lower priority.
Upon FIQ, the F Bit (FIQ disable) is (NOT) automatically set ???
In privileged (non-user) modes, FIQs may be also manually disabled by setting the F Bit in CPSR.

IRQ (Normal Interrupt Request)
This interrupt is generated by a LOW level on the nIRQ input. Unlike FIQ, the IRQ mode is not having its

own banked R8-R12 registers.
IRQ is having lower priority than FIQ, and IRQs are automatically disabled when a FIQ exception becomes executed. In privileged (non-user) modes, IRQs may be also manually disabled by setting the I Bit in CPSR.
Upon IRQ, the I Bit (IRQ disable) is automatically set.
To return from IRQ Mode (continuing at following opcode):

```
SUBS PC,R14,4   ;both PC=R14_irq-4, and CPSR=SPSR_irq
```

Software Interrupt
Generated by a software interrupt instruction (SWI). Recommended to request a supervisor (operating system) function. The SWI instruction may also contain a parameter in the 'comment field' of the opcode: In case that your main program issues SWIs from both inside of THUMB and ARM states, note that your exception handler must then separate between 24bit comment fields in ARM opcodes, and 8bit comment fields in THUMB opcodes (if necessary determine old state by examining T Bit in SPSR_svc).
However, in Little Endian mode, you could alternately use only the most significant 8bits of the 24bit ARM comment field (as done in the GBA, for example) - the exception handler could then process the BYTE at [R14-2], regardless of whether it's been called from ARM or THUMB state.
To return from Supervisor Mode (continuing at following opcode):

```
MOVS PC,R14   ;both PC=R14_svc, and CPSR=SPSR_svc
```

Note: Like all other exceptions, SWIs are always executed in ARM state, no matter whether it's been caused by an ARM or THUMB state SWI instruction.

Undefined Instruction
This exception is generated when the CPU comes across an instruction which it cannot handle. Most likely signalizing that the program has locked up, and that an errormessage should be displayed.
However, it might be also used to emulate custom functions, ie. as an additional 'SWI' instruction (which'd use R14_und and SPSR_und though, and it'd thus allow to execute the Undefined Instruction handler from inside of Supervisor mode without having to save R14_svc and SPSR_svc).
To return from Undefined Mode (continuing at following opcode):

```
MOVS PC,R14   ;both PC=R14_und, and CPSR=SPSR_und
```

Note that not all unused opcodes are necessarily producing an exception, for example, an ARM state Multiply instruction with Bit 6 set to "1" would be blindly accepted as 'legal' opcode.

Abort
Aborts (page faults) are mostly supposed for virtual memory systems (ie. not used in GBA, as far as I know), otherwise they might be used just to display an error message. Two types of aborts exists:
- Prefetch Abort (occurs during an instruction prefetch)
- Data Abort (occurs during a data access)

A virtual memory systems abort handler would then most likey determine the fault address: For prefetch abort that's just "R14_abt-4". For Data abort, the THUMB or ARM instruction at "R14_abt-8" needs to be 'disassembled' in order to determine the addressed data in memory.
The handler would then fix the error by loading the respective memory page into physical memory, and then retry to execute the SAME instruction again, by returning as follows:

```
prefetch abort: SUBS PC,R14,#4   ;PC=R14_abt-4, and CPSR=SPSR_abt
data abort:     SUBS PC,R14,#8   ;PC=R14_abt-8, and CPSR=SPSR_abt
```

Separate exception vectors for prefetch/data abort exists, each should use the respective return instruction as shown above.

Reset
Forces PC=00000000h, and forces control bits of CPSR to T=0 (ARM state), F=1 and I=1 (disable FIQ and IRQ), and M4-0=10011b (Supervisor mode).

THUMB Instruction Set

When operating in THUMB state, cut-down 16bit opcodes are used.

Summary
THUMB Instruction Summary

Register Operations
THUMB.1: move shifted register
THUMB.2: add/subtract
THUMB.3: move/compare/add/subtract immediate
THUMB.4: ALU operations
THUMB.5: Hi register operations/branch exchange

Memory Addressing Operations
THUMB.6: load PC-relative
THUMB.7: load/store with register offset
THUMB.8: load/store sign-extended byte/halfword
THUMB.9: load/store with immediate offset
THUMB.10: load/store halfword
THUMB.11: load/store SP-relative
THUMB.12: get relative address
THUMB.13: add offset to stack pointer
THUMB.14: push/pop registers
THUMB.15: multiple load/store

Jumps and Calls
THUMB.16: conditional branch
THUMB.17: software interrupt
THUMB.18: unconditional branch
THUMB.19: long branch with link
(See also THUMB.5 for branch and exchange.)

Note:
Switching between ARM and THUMB state can be done by using the Branch and Exchange (BX) instruction.

THUMB Instruction Summary

The table below lists all THUMB mode instructions with clock cycles, affected CPSR flags, Format/chapter number, and description.
Only register R0..R7 can be used in thumb mode (unless R8-15,SP,PC are explicitely mentioned).

Logical Operations

```
Instruction        Cycles Flags Format Expl.
MOV Rd,Imm8bit      1S     NZ--  3    Rd=nn
MOV Rd,Rs           1S     NZ00  2    Rd=Rs+0
MOV R0..14,R8..15   1S     ----  5    Rd=Rs
MOV R8..14,R0..15   1S     ----  5    Rd=Rs
MOV R15,R0..15      2S+1N  ----  5    PC=Rs
MVN Rd,Rs           1S     NZ--  4    Rd=NOT Rs
AND Rd,Rs           1S     NZ--  4    Rd=Rd AND Rs
TST Rd,Rs           1S     NZ--  4 Void=Rd AND Rs
```

```
BIC Rd,Rs           1S     NZ--  4   Rd=Rd AND NOT Rs
ORR Rd,Rs           1S     NZ--  4   Rd=Rd OR Rs
EOR Rd,Rs           1S     NZ--  4   Rd=Rd XOR Rs
LSL Rd,Rs,Imm5bit   1S     NZc-  1   Rd=Rs SHL nn
LSL Rd,Rs           1S+1I  NZc-  4   Rd=Rd SHL (Rs AND 0FFh)
LSR Rd,Rs,Imm5bit   1S     NZc-  1   Rd=Rs SHR nn
LSR Rd,Rs           1S+1I  NZc-  4   Rd=Rd SHR (Rs AND 0FFh)
ASR Rd,Rs,Imm5bit   1S     NZc-  1   Rd=Rs SRA nn
ASR Rd,Rs           1S+1I  NZc-  4   Rd=Rd SRA (Rs AND 0FFh)
ROR Rd,Rs           1S+1I  NZc-  4   Rd=Rd ROR (Rs AND 0FFh)
NOP                 1S     ----  5   R8=R8
```

Carry flag affected only if shift amount is non-zero.

Artithmetic Operations and Multiply

```
Instruction        Cycles Flags Format Expl.
ADD Rd,Rs,Imm3bit   1S     NZCV  2   Rd=Rs+nn
ADD Rd,Imm8bit      1S     NZCV  3   Rd=Rd+nn
ADD Rd,Rs,Rn        1S     NZCV  2   Rd=Rs+Rn
ADD R0..14,R8..15   1S     ----  5   Rd=Rd+Rs
ADD R8..14,R0..15   1S     ----  5   Rd=Rd+Rs
ADD R15,R0..15      2S+1N  ----  5   PC=Rd+Rs
ADD Rd,PC,Imm8bit*4 1S     ---- 12   Rd=(($+4) AND NOT 2)+nn
ADD Rd,SP,Imm8bit*4 1S     ---- 12   Rd=SP+nn
ADD SP,Imm7bit*4    1S     ---- 13   SP=SP+nn
ADD SP,-Imm7bit*4   1S     ---- 13   SP=SP-nn
ADC Rd,Rs           1S     NZCV  4   Rd=Rd+Rs+Cy
SUB Rd,Rs,Imm3Bit   1S     NZCV  2   Rd=Rs-nn
SUB Rd,Imm8bit      1S     NZCV  3   Rd=Rd-nn
SUB Rd,Rs,Rn        1S     NZCV  2   Rd=Rs-Rn
SBC Rd,Rs           1S     NZCV  4   Rd=Rd-Rs-NOT Cy
NEG Rd,Rs           1S     NZCV  4   Rd=0-Rs
CMP Rd,Imm8bit      1S     NZCV  3 Void=Rd-nn
CMP Rd,Rs           1S     NZCV  4 Void=Rd-Rs
CMP R0-15,R8-15     1S     NZCV  5 Void=Rd-Rs
CMP R8-15,R0-15     1S     NZCV  5 Void=Rd-Rs
CMN Rd,Rs           1S     NZCV  4 Void=Rd+Rs
MUL Rd,Rs           1S+mI  NZx-  4   Rd=Rd*Rs
```

Jumps and Calls

```
Instruction        Cycles    Flags Format Expl.
B disp             2S+1N      ---- 18  PC=$+/-2048
BL disp            3S+1N      ---- 19  PC=$+/-4M, LR=$+5
B{cond=true} disp  2S+1N      ---- 16  PC=$+/-0..256
B{cond=false} disp 1S         ---- 16  N/A
BX R0..15          2S+1N      ----  5  PC=Rs, ARM/THUMB (Rs bit0)
SWI Imm8bit        2S+1N      ---- 17  PC=8, ARM SVC mode, LR=$+2
POP {Rlist,}PC   (n+1)S+2N+1I ---- 14
MOV R15,R0..15     2S+1N      ----  5  PC=Rs
ADD R15,R0..15     2S+1N      ----  5  PC=Rd+Rs
```

The thumb BL instuction occupies two 16bit opcodes, 32bit in total.

Memory Load/Store

```
Instruction        Cycles    Flags Format Expl.
LDR  Rd,[Rb,5bit*4] 1S+1N+1I  ----  9  Rd = WORD[Rb+nn]
LDR  Rd,[PC,8bit*4] 1S+1N+1I  ----  6  Rd = WORD[PC+nn]
LDR  Rd,[SP,8bit*4] 1S+1N+1I  ---- 11  Rd = WORD[SP+nn]
LDR  Rd,[Rb,Ro]     1S+1N+1I  ----  7  Rd = WORD[Rb+Ro]
LDRB Rd,[Rb,5bit*1] 1S+1N+1I  ----  9  Rd = BYTE[Rb+nn]
LDRB Rd,[Rb,Ro]     1S+1N+1I  ----  7  Rd = BYTE[Rb+Ro]
LDRH Rd,[Rb,5bit*2] 1S+1N+1I  ---- 10  Rd = HALFWORD[Rb+nn]
LDRH Rd,[Rb,Ro]     1S+1N+1I  ----  8  Rd = HALFWORD[Rb+Ro]
LDSB Rd,[Rb,Ro]     1S+1N+1I  ----  8  Rd = SIGNED_BYTE[Rb+Ro]
LDSH Rd,[Rb,Ro]     1S+1N+1I  ----  8  Rd = SIGNED_HALFWORD[Rb+Ro]
```

```
STR  Rd,[Rb,5bit*4] 2N        ----  9  WORD[Rb+nn] = Rd
STR  Rd,[SP,8bit*4] 2N        ---- 11  WORD[SP+nn] = Rd
STR  Rd,[Rb,Ro]     2N        ----  7  WORD[Rb+Ro] = Rd
STRB Rd,[Rb,5bit*1] 2N        ----  9  BYTE[Rb+nn] = Rd
STRB Rd,[Rb,Ro]     2N        ----  7  BYTE[Rb+Ro] = Rd
STRH Rd,[Rb,5bit*2] 2N        ---- 10  HALFWORD[Rb+nn] = Rd
STRH Rd,[Rb,Ro]     2N        ----  8  HALFWORD[Rb+Ro]=Rd
PUSH {Rlist}{LR}    (n-1)S+2N ---- 14
POP  {Rlist}{PC}              ---- 14
STMIA Rb!,{Rlist}   (n-1)S+2N ---- 15
LDMIA Rb!,{Rlist}   nS+1N+1I  ---- 15
```

THUMB Binary Opcode Format
This table summarizes the position of opcode/parameter bits for THUMB mode instructions, Format 1-19.

```
Form|_15|_14|_13|_12|_11|_10|_9_|_8_|_7_|_6_|_5_|_4_|_3_|_2_|_1_|_0_|
__1_|_0___0___0_|__Op___|_______Offset______|____Rs_____|____Rd_____|Shifted
__2_|_0___0___0___1___1_|_I,_Op_|___Rn/nn___|____Rs_____|____Rd_____|ADD/SUB
__3_|_0___0___1_|__Op___|____Rd_____|_____________Offset_____________|Immedi.
__4_|_0___1___0___0___0___0_|______Op_______|____Rs_____|____Rd_____|AluOp
__5_|_0___1___0___0___0___1_|__Op___|Hd_|Hs_|____Rs_____|____Rd_____|HiReg/BX
__6_|_0___1___0___0___1_|____Rd_____|_____________Word_____________|LDR PC
__7_|_0___1___0___1_|__Op___|_0_|___Ro______|____Rb_____|____Rd_____|LDR/STR
__8_|_0___1___0___1_|__Op___|_1_|___Ro______|____Rb_____|____Rd_____|""H/SB/SH
__9_|_0___1___1_|__Op___|_______Offset______|____Rb_____|____Rd_____|""{B}
_10_|_1___0___0___0_|Op_|_______Offset______|____Rb_____|____Rd_____|""H
_11_|_1___0___0___1_|Op_|____Rd_____|_____________Word_____________|"" SP
_12_|_1___0___1___0_|Op_|____Rd_____|_____________Word_____________|ADD PC/SP
_13_|_1___0___1___1___0___0___0___0_|_S_|___________Word_____________|ADD SP,nn
_14_|_1___0___1___1_|Op_|_1___0_|_R_|_____________Rlist_____________|PUSH/POP
_15_|_1___1___0___0_|Op_|____Rb_____|_____________Rlist_____________|STM/LDM
_16_|_1___1___0___1_|_____Cond______|_____________Signed_Offset_____|B{cond}
_17_|_1___1___0___1___1___1___1___1_|_____________User_Data_____________|SWI
_18_|_1___1___1___0___0_|_____________________Offset_____________________|B
_19_|_1___1___1___1_|_H_|_____________Offset_Low/High_____________|BL
```

THUMB.1: move shifted register

Opcode Format

```
Bit    Expl.
15-13  Must be 000b for 'move shifted register' instructions
12-11  Opcode
         00b: LSL Rd,Rs,#Offset   (logical/arithmetic shift left)
         01b: LSR Rd,Rs,#Offset   (logical    shift right)
         10b: ASR Rd,Rs,#Offset   (arithmetic shift right)
         11b: Reserved (used for add/subtract instructions)
10-6   Offset                    (0-31)
5-3    Rs - Source register      (R0..R7)
2-0    Rd - Destination register (R0..R7)
```

Example: LSL Rd,Rs,#nn ; Rd = Rs << nn ; ARM equivalent: MOVS Rd,Rs,LSL #nn
Zero shift amount is having special meaning (same as for ARM shifts), LSL#0 performs no shift (the the carry flag remains unchanged), LSR/ASR#0 are interpreted as LSR/ASR#32. Attempts to specify LSR/ASR#0 in source code are automatically redirected as LSL#0, and source LSR/ASR#32 is redirected as opcode LSR/ASR#0.
Execution Time: 1S
Flags: Z=zeroflag, N=sign, C=carry (except LSL#0: C=unchanged), V=unchanged.

THUMB.2: add/subtract

Opcode Format

```
Bit    Expl.
15-11  Must be 00011b for 'add/subtract' instructions
10-9   Opcode (0-3)
         0: ADD Rd,Rs,Rn   ;add register        Rd=Rs+Rn
         1: SUB Rd,Rs,Rn   ;subtract register   Rd=Rs-Rn
         2: ADD Rd,Rs,#nn  ;add immediate       Rd=Rs+nn
         3: SUB Rd,Rs,#nn  ;subtract immediate  Rd=Rs-nn
       Pseudo/alias opcode with Imm=0:
         2: MOV Rd,Rs      ;move (affects cpsr) Rd=Rs+0
8-6    For Register Operand:
         Rn - Register Operand (R0..R7)
       For Immediate Operand:
         nn - Immediate Value  (0-7)
5-3    Rs - Source register       (R0..R7)
2-0    Rd - Destination register  (R0..R7)
```

Return: Rd contains result, N,Z,C,V affected (including MOV).
Execution Time: 1S

THUMB.3: move/compare/add/subtract immediate

Opcode Format

```
Bit    Expl.
15-13  Must be 001b for this type of instructions
12-11  Opcode
         00b: MOV Rd,#nn      ;move     Rd   = #nn
         01b: CMP Rd,#nn      ;compare  Void = Rd - #nn
         10b: ADD Rd,#nn      ;add      Rd   = Rd + #nn
         11b: SUB Rd,#nn      ;subtract Rd   = Rd - #nn
10-8   Rd - Destination Register  (R0..R7)
7-0    nn - Unsigned Immediate    (0-255)
```

ARM equivalents for MOV/CMP/ADD/SUB are MOVS/CMP/ADDS/SUBS same format.
Execution Time: 1S
Return: Rd contains result (except CMP), N,Z,C,V affected (for MOV only N,Z).

THUMB.4: ALU operations

Opcode Format

```
Bit    Expl.
15-10  Must be 010000b for this type of instructions
9-6    Opcode (0-Fh)
         0: AND Rd,Rs     ;AND logical       Rd = Rd AND Rs
         1: EOR Rd,Rs     ;XOR logical       Rd = Rd XOR Rs
         2: LSL Rd,Rs     ;log. shift left   Rd = Rd << (Rs AND 0FFh)
         3: LSR Rd,Rs     ;log. shift right  Rd = Rd >> (Rs AND 0FFh)
         4: ASR Rd,Rs     ;arit shift right  Rd = Rd SRA Rs
         5: ADC Rd,Rs     ;add with carry    Rd = Rd + Rs + Cy
         6: SBC Rd,Rs     ;sub with carry    Rd = Rd - Rs - NOT Cy
         7: ROR Rd,Rs     ;rotate right      Rd = Rd ROR (Rs AND 0FFh)
         8: TST Rd,Rs     ;test            Void = Rd AND (Rs AND 0FFh)
         9: NEG Rd,Rs     ;negate            Rd = 0 - Rs
         A: CMP Rd,Rs     ;compare         Void = Rd - Rs
         B: CMN Rd,Rs     ;neg.compare     Void = Rd + Rs
         C: ORR Rd,Rs     ;OR logical        Rd = Rd OR Rs
```

```
          D: MUL Rd,Rs     ;multiply          Rd = Rd * Rs
          E: BIC Rd,Rs     ;bit clear         Rd = Rd AND NOT Rs
          F: MVN Rd,Rs     ;not               Rd = NOT Rs
  5-3    Rs - Source Register       (R0..R7)
  2-0    Rd - Destination Register  (R0..R7)
```

ARM equivalent for NEG would be RSBS.
Return: Rd contains result (except TST,CMP,CMN),
Affected Flags:

```
  N,Z,C,V for  ADC,SBC,NEG,CMP,CMN
  N,Z,C   for  LSL,LSR,ASR,ROR (carry flag unchanged if zero shift amount)
  N,Z,C   for  MUL (carry flag destroyed)
  N,Z     for  AND,EOR,TST,ORR,BIC,MVN
```

Execution Time:

```
  1S      for  AND,EOR,ADC,SBC,TST,NEG,CMP,CMN,ORR,BIC,MVN
  1S+1I   for  LSL,LSR,ASR,ROR
  1S+mI   for  MUL (m=1..4 depending on MSBs of incoming Rd value)
```

THUMB.5: Hi register operations/branch exchange

Opcode Format

```
  Bit    Expl.
  15-10  Must be 010001b for this type of instructions
  9-8    Opcode (0-3)
           0: ADD Rd,Rs   ;add        Rd = Rd+Rs
           1: CMP Rd,Rs   ;compare  Void = Rd-Rs  ;CPSR affected
           2: MOV Rd,Rs   ;move       Rd = Rs
           3: BX  Rs      ;jump       PC = Rs     ;may switch THUMB/ARM
  7      MSBd - Destination Register most significant bit
  6      MSBs - Source Register most significant bit
  5-3    Rs - Source Register        (together with MSBs: R0..R15)
  2-0    Rd - Destination Register   (together with MSBd: R0..R15)
```

Restrictions: For ADD/CMP/MOV, MSBs and/or MSBd must be set, ie. it is not allowed that both are cleared.
When using R15 (PC) as operand, the value will be the address of the instruction plus 4.
For BX, MSBs may be 0 or 1, MSBd must be zero, Rd is not used.
For BX, when Bit 0 of the value in Rs is zero:

```
  Processor will be switched into ARM mode!
  If so, Bit 1 of Rs must be cleared (32bit word aligned).
  Thus, BX PC (switch to ARM) may be issued from word-aligned address
  only, the destination is PC+4 (ie. the following halfword is skipped).
```

Assemblers/Disassemblers should use MOV R8,R8 as NOP (in THUMB mode).
Return: Only CMP affects CPSR condition flags!
Execution Time:

```
 1S     for ADD/MOV/CMP
 2S+1N  for ADD/MOV with Rd=R15, and for BX
```

THUMB.6: load PC-relative

Opcode Format

```
  Bit    Expl.
  15-11  Must be 01001b for this type of instructions
```

```
N/A    Opcode (fixed)
         LDR Rd,[PC,#nn]      ;load 32bit    Rd = WORD[PC+nn]
10-8   Rd - Destination Register   (R0..R7)
7-0    nn - Unsigned offset        (0-1020 in steps of 4)
```

The value of PC will be interpreted as (($+4) AND NOT 2).
Return: No flags affected, data loaded into Rd.
Execution Time: 1S+1N+1I

THUMB.7: load/store with register offset

Opcode Format

```
Bit    Expl.
15-12  Must be 0101b for this type of instructions
11-10  Opcode (0-3)
        0: STR  Rd,[Rb,Ro]   ;store 32bit data  WORD[Rb+Ro] = Rd
        1: STRB Rd,[Rb,Ro]   ;store  8bit data  BYTE[Rb+Ro] = Rd
        2: LDR  Rd,[Rb,Ro]   ;load  32bit data  Rd = WORD[Rb+Ro]
        3: LDRB Rd,[Rb,Ro]   ;load   8bit data  Rd = BYTE[Rb+Ro]
9      Must be zero (0) for this type of instructions
8-6    Ro - Offset Register              (R0..R7)
5-3    Rb - Base Register                (R0..R7)
2-0    Rd - Source/Destination Register  (R0..R7)
```

Return: No flags affected, data loaded either into Rd or into memory.
Execution Time: 1S+1N+1I for LDR, or 2N for STR

THUMB.8: load/store sign-extended byte/halfword

Opcode Format

```
Bit    Expl.
15-12  Must be 0101b for this type of instructions
11-10  Opcode (0-3)
        0: STRH Rd,[Rb,Ro]  ;store 16bit data          HALFWORD[Rb+Ro] = Rd
        1: LDSB Rd,[Rb,Ro]  ;load sign-extended 8bit   Rd = BYTE[Rb+Ro]
        2: LDRH Rd,[Rb,Ro]  ;load zero-extended 16bit  Rd = HALFWORD[Rb+Ro]
        3: LDSH Rd,[Rb,Ro]  ;load sign-extended 16bit  Rd = HALFWORD[Rb+Ro]
9      Must be set (1) for this type of instructions
8-6    Ro - Offset Register              (R0..R7)
5-3    Rb - Base Register                (R0..R7)
2-0    Rd - Source/Destination Register  (R0..R7)
```

Return: No flags affected, data loaded either into Rd or into memory.
Execution Time: 1S+1N+1I for LDR, or 2N for STR

THUMB.9: load/store with immediate offset

Opcode Format

```
Bit    Expl.
15-13  Must be 011b for this type of instructions
12-11  Opcode (0-3)
        0: STR  Rd,[Rb,#nn]  ;store 32bit data   WORD[Rb+nn] = Rd
        1: LDR  Rd,[Rb,#nn]  ;load  32bit data   Rd = WORD[Rb+nn]
        2: STRB Rd,[Rb,#nn]  ;store  8bit data   BYTE[Rb+nn] = Rd
        3: LDRB Rd,[Rb,#nn]  ;load   8bit data   Rd = BYTE[Rb+nn]
10-6   nn - Unsigned Offset              (0-31 for BYTE, 0-124 for WORD)
```

```
5-3    Rb - Base Register                (R0..R7)
2-0    Rd - Source/Destination Register  (R0..R7)
```

Return: No flags affected, data loaded either into Rd or into memory.
Execution Time: 1S+1N+1I for LDR, or 2N for STR

THUMB.10: load/store halfword

Opcode Format

```
Bit    Expl.
15-12  Must be 1000b for this type of instructions
11     Opcode (0-1)
        0: STRH Rd,[Rb,#nn]  ;store 16bit data   HALFWORD[Rb+nn] = Rd
        1: LDRH Rd,[Rb,#nn]  ;load  16bit data   Rd = HALFWORD[Rb+nn]
10-6   nn - Unsigned Offset              (0-62, step 2)
5-3    Rb - Base Register                (R0..R7)
2-0    Rd - Source/Destination Register  (R0..R7)
```

Return: No flags affected, data loaded either into Rd or into memory.
Execution Time: 1S+1N+1I for LDR, or 2N for STR

THUMB.11: load/store SP-relative

Opcode Format

```
Bit    Expl.
15-12  Must be 1001b for this type of instructions
11     Opcode (0-1)
        0: STR  Rd,[SP,#nn]  ;store 32bit data   WORD[SP+nn] = Rd
        1: LDR  Rd,[SP,#nn]  ;load  32bit data   Rd = WORD[SP+nn]
10-8   Rd - Source/Destination Register  (R0..R7)
7-0    nn - Unsigned Offset              (0-1020, step 4)
```

Return: No flags affected, data loaded either into Rd or into memory.
Execution Time: 1S+1N+1I for LDR, or 2N for STR

THUMB.12: get relative address

Opcode Format

```
Bit    Expl.
15-12  Must be 1010b for this type of instructions
11     Opcode/Source Register (0-1)
        0: ADD  Rd,PC,#nn    ;Rd = (($+4) AND NOT 2) + nn
        1: ADD  Rd,SP,#nn    ;Rd = SP + nn
10-8   Rd - Destination Register         (R0..R7)
7-0    nn - Unsigned Offset              (0-1020, step 4)
```

Return: No flags affected, result in Rd.
Execution Time: 1S

THUMB.13: add offset to stack pointer

Opcode Format

```
Bit    Expl.
15-8   Must be 10110000b for this type of instructions
7      Opcode/Sign
        0: ADD  SP,#nn       ;SP = SP + nn
        1: ADD  SP,#-nn      ;SP = SP - nn
6-0    nn - Unsigned Offset    (0-508, step 4)
```

Return: No flags affected, SP adjusted.
Execution Time: 1S

THUMB.14: push/pop registers

Opcode Format

```
Bit    Expl.
15-12  Must be 1011b for this type of instructions
11     Opcode (0-1)
        0: PUSH {Rlist}{LR}   ;store in memory, decrements SP (R13)
        1: POP  {Rlist}{PC}   ;load from memory, increments SP (R13)
10-9   Must be 10b for this type of instructions
8      PC/LR Bit (0-1)
        0: No
        1: PUSH LR (R14), or POP PC (R15)
7-0    Rlist - List of Registers (R7..R0)
```

In THUMB mode stack is always meant to be 'full descending', ie. PUSH is equivalent to 'STMFD/STMDB' and POP to 'LDMFD/LDMIA' in ARM mode.

Examples:

```
PUSH {R0-R3}     ;push R0,R1,R2,R3
PUSH {R0,R2,LR}  ;push R0,R2,LR
POP  {R4,R7}     ;pop R4,R7
POP  {R2-R4,PC}  ;pop R2,R3,R4,PC
```

Note: When calling to a sub-routine, the return address is stored in LR register, when calling further sub-routines, PUSH {LR} must be used to save higher return address on stack. If so, POP {PC} can be later used to return from the sub-routine.
POP {PC} ignores the least significant bit of the return address (processor remains in thumb state even if bit0 was cleared), when intending to return with optional mode switch, use a POP/BX comination (eg. POP {R3} / BX R3).
Return: No flags affected, SP adjusted, registers loaded/stored.
Execution Time: nS+1N+1I (POP), (n+1)S+2N+1I (POP PC), or (n-1)S+2N (PUSH).

THUMB.15: multiple load/store

Opcode Format

```
Bit    Expl.
15-12  Must be 1100b for this type of instructions
11     Opcode (0-1)
        0: STMIA Rb!,{Rlist}   ;store in memory, increments Rb
        1: LDMIA Rb!,{Rlist}   ;load from memory, increments Rb
10-8   Rb - Base register (modified) (R0-R7)
7-0    Rlist - List of Registers     (R7..R0)
```

Both STM and LDM are incrementing the Base Register.
The lowest register in the list (ie. R0, if it's in the list) is stored/loaded at the lowest memory address.
Examples:

```
STMIA R7!,{R0-R2}  ;store R0,R1,R2
LDMIA R0!,{R1,R5}  ;store R1,R5
```

Return: No flags affected, Rb adjusted, registers loaded/stored.
Execution Time: nS+1N+1I for LDM, or (n-1)S+2N for STM.

THUMB.16: conditional branch

Opcode Format

```
Bit    Expl.
15-12  Must be 1101b for this type of instructions
11-8   Opcode/Condition (0-Fh)
        0: BEQ label   ;Z=1         ;equal (zero)
        1: BNE label   ;Z=0         ;not equal (nonzero)
        2: BCS label   ;C=1         ;unsigned higher or same (carry set)
        3: BCC label   ;C=0         ;unsigned lower (carry cleared)
        4: BMI label   ;N=1         ;negative (minus)
        5: BPL label   ;N=0         ;positive or zero (plus)
        6: BVS label   ;V=1         ;overflow (V set)
        7: BVC label   ;V=0         ;no overflowplus (V cleared)
        8: BHI label   ;C=1 and Z=0 ;unsigned higher
        9: BLS label   ;C=0 or Z=1  ;unsigned lower or same
        A: BGE label   ;N=V         ;greater or equal
        B: BLT label   ;N<>V        ;less than
        C: BGT label   ;Z=0 and N=V ;greater than
        D: BLE label   ;Z=1 or N<>V ;less or equal
        E: Undefined, should not be used
        F: Reserved for SWI instruction (see SWI opcode)
7-0    Signed Offset, step 2 ($+4-256..$+4+254)
```

Destination address must by halfword aligned (ie. bit 0 cleared)
Return: No flags affected, PC adjusted if condition true
Execution Time:

```
2S+1N   if condition true (jump executed)
1S      if condition false
```

THUMB.17: software interrupt

Opcode Format

```
Bit    Expl.
15-8   Must be 11011111b for this type of instructions
N/A    Opcode (fixed)
        SWI nn    ;perform software interrupt
7-0    nn - Comment Immediate    (0-255)
```

Supposed for calls to the operating system - Enter Supervisor mode (SVC) in ARM state.
Execution:

```
R14_svc=PC+2   ;save return address in LR_svc
SPSR_svc=CPSR  ;save CPSR flags
CPSR=<changed> ;Enter Supervisor mode (svc) in ARM state
PC=00000008h   ;jump to SWI vector address
```

Interpreting the Comment Field:
The immediate parameter is ignored by the processor, the user interrupt handler may read-out this number by examing the lower 8bit of the 16bit opcode opcode at [R14_svc-2]. In case that your program executes SWI's from inside of ARM mode also: Your SWI handler must then examine the T Bit SPSR_svc in order to determine whether it's been a ARM SWI - if so, examing the lower 24bit of the 32bit opcode opcode at

[R14_svc-4].
For Returning from SWI use this instruction:

```
MOVS PC,R14
```

That instructions does both restoring PC and CPSR, ie. PC=R14_svc, and CPSR=SPRS_svc. In this case (as called from THUMB mode), this also involves restoring THUMB mode.
Nesting SWIs:
Obviously, SPSR_svc and R14_svc may store only old flags and return address from current SWI, so, the SWI handler must push these values before nesting SWIs.
Execution Time: 2S+1N

THUMB.18: unconditional branch

Opcode Format

```
Bit    Expl.
15-11  Must be 11100b for this type of instructions
N/A    Opcode (fixed)
        B label   ;branch (jump)
10-0   Signed Offset, step 2 ($+4-2048..$+4+2046)
```

Return: No flags affected, PC adjusted.
Execution Time: 2S+1N

THUMB.19: long branch with link

Opcode Format
This may be used to call (or jump) to a subroutine, return address is saved in LR (R14).
Unlike all other THUMB mode instructions, this instruction occupies 32bit of memory which are split into two 16bit THUMB opcodes.
First Instruction - LR = PC+4+(nn SHL 12)

```
Bit    Expl.
15-11  Must be 11110b for this type of instructions
10-0   nn - Upper 11 bits of Target Address
```

Second Instruction - PC = LR + (nn SHL 1), and LR = PC+2 OR 1

```
Bit    Expl.
15-11  Must be 11111b for this type of instructions
10-0   nn - Lower 11 bits of Target Address
```

Assembler Format:

```
BL label
```

The destination address range is (PC+4)-400000h..+3FFFFEh, ie. PC+/-4M.
Target must be halfword-aligned. As Bit 0 in LR is set, it may be used to return by a BX LR instruction (keeping CPU in THUMB mode).
Return: No flags affected, PC adjusted, return address in LR.
Execution Time: 3S+1N (first opcode 1S, second opcode 2S+1N).

ARM Instruction Set

When operating in ARM state, full 32bit opcodes are used.

Summaries
ARM Instruction Summary
ARM Condition Field

Jumps and Calls
ARM.3: Branch and Exchange (BX)
ARM.4: Branch and Branch with Link (B, BL)

Register Operations
ARM.5: Data Processing
ARM.6: PSR Transfer (MRS, MSR)
ARM.7: Multiply and Multiply-Accumulate (MUL, MLA)
ARM.8: Multiply Long and Multiply-Accumulate Long (MULL, MLAL)

Memory Addressing Operations
ARM.9: Single Data Transfer (LDR, STR)
ARM.10: Halfword and Signed Data Transfer (STRH,LDRH,LDRSB,LDRSH)
ARM.11: Block Data Transfer (LDM,STM)
ARM.12: Single Data Swap (SWP)

Exception Calls and Coprocessor
ARM.13: Software Interrupt (SWI)
ARM.14: Coprocessor Data Operations (CDP)
ARM.15: Coprocessor Data Transfers (LDC,STC)
ARM.16: Coprocessor Register Transfers (MRC, MCR)
ARM.17: Undefined Instruction

Note:
Switching between ARM and THUMB state can be done by using the Branch and Exchange (BX) instruction.

ARM Instruction Summary

Modification of CPSR flags is optional for all {S} instructions.

Logical Operations

```
Instruction              Cycles   Flags Format Expl.
MOV{cond}{S} Rd,Op2       1S+x+y   NZc- 5   Rd = Op2
MVN{cond}{S} Rd,Op2       1S+x+y   NZc- 5   Rd = NOT Op2
AND{cond}{S} Rd,Rn,Op2    1S+x+y   NZc- 5   Rd = Rn AND Op2
TST{cond}       Rn,Op2    1S+x     NZc- 5 Void = Rn AND Op2
EOR{cond}{S} Rd,Rn,Op2    1S+x+y   NZc- 5   Rd = Rn XOR Op2
TEQ{cond}       Rn,Op2    1S+x     NZc- 5 Void = Rn XOR Op2
ORR{cond}{S} Rd,Rn,Op2    1S+x+y   NZc- 5   Rd = Rn OR Op2
BIC{cond}{S} Rd,Rn,Op2    1S+x+y   NZc- 5   Rd = Rn AND NOT Op2
```

Add x=1I cycles if Op2 shifted-by-register. Add y=1S+1N cycles if Rd=R15.
Carry flag affected only of Op2 contains a non-zero shift amount.

Arithmetic Operations

```
Instruction              Cycles  Flags Format Expl.
ADD{cond}{S} Rd,Rn,Op2    1S+x+y   NZCV 5   Rd = Rn+Op2
ADC{cond}{S} Rd,Rn,Op2    1S+x+y   NZCV 5   Rd = Rn+Op2+Cy
SUB{cond}{S} Rd,Rn,Op2    1S+x+y   NZCV 5   Rd = Rn-Op2
SBC{cond}{S} Rd,Rn,Op2    1S+x+y   NZCV 5   Rd = Rn-Op2+Cy-1
RSB{cond}{S} Rd,Rn,Op2    1S+x+y   NZCV 5   Rd = Op2-Rn
```

```
RSC{cond}{S} Rd,Rn,Op2   1S+x+y   NZCV 5   Rd = Op2-Rn+Cy-1
CMP{cond}       Rn,Op2   1S+x     NZCV 5 Void = Rn-Op2
CMN{cond}       Rn,Op2   1S+x     NZCV 5 Void = Rn+Op2
```

Add x=1I cycles if Op2 shifted-by-register. Add y=1S+1N cycles if Rd=R15.

Multiply

```
Instruction                    Cycles  Flags Format Expl.
MUL{cond}{S} Rd,Rm,Rs           1S+mI     NZx- 7  Rd = Rm*Rs
MLA{cond}{S} Rd,Rm,Rs,Rn        1S+mI+1I  NZx- 7  Rd = Rm*Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs  1S+mI+1I  NZx- 8  RdHiLo = Rm*Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs  1S+mI+2I  NZx- 8  RdHiLo = Rm*Rs+RdHiLo
SMULL{cond}{S} RdLo,RdHi,Rm,Rs  1S+mI+1I  NZx- 8  RdHiLo = Rm*Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs  1S+mI+2I  NZx- 8  RdHiLo = Rm*Rs+RdHiLo
```

Memory Load/Store

```
Instruction                    Cycles       Flags Format Expl.
LDR{cond}{B}{T} Rd,<Address>    1S+1N+1I +y   ---- 9  Rd=[Rn+/-<offset>]
LDR{cond}H      Rd,<Address>    1S+1N+1I +y   ---- 10 Load Unsigned halfword
LDR{cond}SB     Rd,<Address>    1S+1N+1I +y   ---- 10 Load Signed byte
LDR{cond}SH     Rd,<Address>    1S+1N+1I +y   ---- 10 Load Signed halfword
LDM{cond}{amod} Rn{!},<Rlist>{^} nS+1N+1I +y  ---- 11
STR{cond}{B}{T} Rd,<Address>    2N            ---- 9  [Rn+/-<offset>]=Rd
STR{cond}H      Rd,<Address>    2N            ---- 10 Store halfword
STM{cond}{amod} Rn{!},<Rlist>{^} (n-1)S+2N    ---- 11
SWP{cond}{B}    Rd,Rm,[Rn]      1S+2N+1I      ---- 12 Rd=[Rn], [Rn]=Rm
```

For LDR/LDM, add y=1S+1N if Rd=PC, or if Rd in Rlist.

Jumps, Calls, CPSR Mode, and others

```
Instruction             Cycles  Flags Format Expl.
B{cond} label            2S+1N    ---- 4   PC=$+8+/-32M
BL{cond} label           2S+1N    ---- 4   PC=$+8+/-32M, LR=$+4
BX{cond} Rn              2S+1N    ---- 3   PC=Rn, THUMB/ARM (Rn bit0)
MRS{cond} Rd,Psr         1S       ---- 6   Rd=Psr
MSR{cond} Psr{_field},Op 1S      (psr) 6   Psr[field]=Op
SWI{cond} Imm24bit       2S+1N    ---- 13  PC=8, ARM Svc mode, LR=$+4
The Undefined Instruction 2S+1I+1N ---- 17  PC=4, ARM Und mode, LR=$+4
cond=false               1S       ---- ..  Any opcode with condition=false
NOP                      1S       ---- 5   R0=R0
```

Coprocessor Functions (if any)

```
Instruction                        Cycles  Flags Format Expl.
CDP{cond} p#,<cpopc>,cd,cn,cm{,<cp>} 1S+bI   ---- 14 Coprocessor specific
STC{cond}{L} p#,cd,<Address>          (n-1)S+2N+bI 15 [address] = CRd
LDC{cond}{L} p#,cd,<Address>          (n-1)S+2N+bI 15 CRd = [address]
MCR{cond} p#,<cpopc>,Rd,cn,cm{,<cp>} 1S+bI+1C     16 CRn = Rn {<op> CRm}
MRC{cond} p#,<cpopc>,Rd,cn,cm{,<cp>} 1S+(b+1)I+1C 16 Rn = CRn {<op> CRm}
```

Note that no sections 1-2 exist, that is because the sections numbers comply with chapter numbers of the offical ARM docs, which described ARM opcodes in chapter 3-17.

ARM Binary Opcode Format

```
|..3 .................2 .................1 .................0|
|1_0_9_8_7_6_5_4_3_2_1_0_9_8_7_6_5_4_3_2_1_0_9_8_7_6_5_4_3_2_1_0|
|_Cond__|0_0|I|___Op__|S|__Rn___|__Rd___|_______Operand2________| DataProc
|_Cond__|0_0|I|1_0|P|L|0|_Field_|__Rd___|_______Operand________| PSR
|_Cond__|0_0_0_1_0_0_1_0_1_1_1_1_1_1_1_1_1_1_1_1_0_0_0_1|__Rn___| BranchX
|_Cond__|0_0_0_0_0_0|A|S|__Rd___|__Rn___|__Rs___|1_0_0_1|__Rm___| Multiply
```

```
|_Cond__|0_0_0_0_1|U|A|S|_RdHi__|_RdLo__|__Rn___|1_0_0_1|__Rm___| MulLong
|_Cond__|0_0_0_1_0|B|0_0|__Rn___|__Rd___|0_0_0_0_1_0_0_1|__Rm___| Swap
|_Cond__|0_0_0|P|U|0|W|L|__Rn___|__Rd___|0_0_0_0_1|S|H|1|__Rm___| HalfTransR
|_Cond__|0_0_0|P|U|1|W|L|__Rn___|__Rd___|OffsetH|1|S|H|1|OffsetL| HalfTransOf
|_Cond__|0_1|I|P|U|B|W|L|__Rn___|__Rd___|_________Offset________| DataTransf
|_Cond__|0_1_1|________________xxx____________________|1|__xxx__| Undefined
|_Cond__|1_0_0|P|U|S|W|L|__Rn___|__________Register_List________| BlockTrans
|_Cond__|1_0_1|L|____________________Offset_____________________| Branch
|_Cond__|1_1_0|P|U|N|W|L|__Rn___|__CRd__|__CP#__|____Offset_____| CoDataTrans
|_Cond__|1_1_1_0|_CPopc_|__CRn__|__CRd__|__CP#__|_CP__|0|__CRm__| CoDataOp
|_Cond__|1_1_1_0|CPopc|L|__CRn__|__Rd___|__CP#__|_CP__|1|__CRm__| CoRegTrans
|_Cond__|1_1_1_1|_____________Ignored_by_Processor______________| SWI
```

ARM Condition Field

In ARM mode, all instructions can be conditionally executed depending on the state of the CPSR flags (C,N,Z,V). The respecitive suffixes {cond} must be appended to the mnemonics. For example: BEQ = Branch if Equal, MOVMI = Move if Signed.

Code	Suffix	Flags	Meaning
0:	EQ	Z=1	equal (zero)
1:	NE	Z=0	not equal (nonzero)
2:	CS	C=1	unsigned higher or same (carry set)
3:	CC	C=0	unsigned lower (carry cleared)
4:	MI	N=1	negative (minus)
5:	PL	N=0	positive or zero (plus)
6:	VS	V=1	overflow (V set)
7:	VC	V=0	no overflowplus (V cleared)
8:	HI	C=1 and Z=0	unsigned higher
9:	LS	C=0 or Z=1	unsigned lower or same
A:	GE	N=V	greater or equal
B:	LT	N<>V	less than
C:	GT	Z=0 and N=V	greater than
D:	LE	Z=1 or N<>V	less or equal
E:	AL	-	always
F:	Reserved, don't use		

To define a non-conditional instruction which is always to be executed (regardless of any flags), the AL suffix may be used - that is the same as if no suffix is specified. For example, MOVAL would be usually abbreviated to MOV.

Execution Time: If condition=false: 1S cycle.
Otherwise as specified for the respective opcode.

ARM.3: Branch and Exchange (BX)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-4   Must be "0001.0010.1111.1111.1111.0001" for this instruction
       Opcode (fixed)
         BX{cond} Rn    ;PC = Rn
3-0    Rn - Operand Register  (R0-R14)
```

Switching to THUMB Mode: Set Bit 0 of the value in Rn to 1, program continues then at Rn-1 in THUMB mode.

Results in undefined behaviour if using R15 (PC+8 itself) as operand.
Execution Time: 2S + 1N
Return: No flags affected.

ARM.4: Branch and Branch with Link (B, BL)

Opcode Format
Branch (B) is supposed to jump to a subroutine. Branch with Link is meant to be used to call to a subroutine, return address is then saved in R14.

```
Bit    Expl.
31-28  Condition
27-25  Must be "101" for this instruction
24     Opcode (0-1)
        0: B{cond} label    ;branch            PC=PC+8+nn
        1: BL{cond} label   ;branch with Link  R14=PC, PC=PC+8+nn
23-0   nn - Signed Offset, step 4      (-32M..+32M in steps of 4)
```

Branch with Link can be used to 'call' to a sub-routine, which may then 'return' by MOV PC,R14 for example.
Execution Time: 2S + 1N
Return: No flags affected.

ARM.5: Data Processing

Opcode Format

```
Bit    Expl.
31-28  Condition
27-26  Must be 00b for this instruction
25     I - Immediate 2nd Operand Flag (0=Register, 1=Immediate)
24-21  Opcode (0-Fh)               ;*=Arithmetic, otherwise Logical
         0: AND{cond}{S} Rd,Rn,Op2    ;AND logical       Rd = Rn AND Op2
         1: EOR{cond}{S} Rd,Rn,Op2    ;XOR logical       Rd = Rn XOR Op2
         2: SUB{cond}{S} Rd,Rn,Op2 ;* ;subtract          Rd = Rn-Op2
         3: RSB{cond}{S} Rd,Rn,Op2 ;* ;subtract reversed Rd = Op2-Rn
         4: ADD{cond}{S} Rd,Rn,Op2 ;* ;add               Rd = Rn+Op2
         5: ADC{cond}{S} Rd,Rn,Op2 ;* ;add with carry    Rd = Rn+Op2+Cy
         6: SBC{cond}{S} Rd,Rn,Op2 ;* ;sub with carry    Rd = Rn-Op2+Cy-1
         7: RSC{cond}{S} Rd,Rn,Op2 ;* ;sub cy. reversed  Rd = Op2-Rn+Cy-1
         8: TST{cond}       Rn,Op2    ;test            Void = Rn AND Op2
         9: TEQ{cond}       Rn,Op2    ;test exclusive  Void = Rn XOR Op2
         A: CMP{cond}       Rn,Op2 ;* ;compare         Void = Rn-Op2
         B: CMN{cond}       Rn,Op2 ;* ;compare neg.    Void = Rn+Op2
         C: ORR{cond}{S} Rd,Rn,Op2    ;OR logical        Rd = Rn OR Op2
         D: MOV{cond}{S} Rd,Op2       ;move              Rd = Op2
         E: BIC{cond}{S} Rd,Rn,Op2    ;bit clear         Rd = Rn AND NOT Op2
         F: MVN{cond}{S} Rd,Op2       ;not               Rd = NOT Op2
20     S - Set Condition Codes (0=No, 1=Yes)
19-16  Rn - 1st Operand Register (R0..R15) (including PC=R15)
15-12  Rd - Destination Register (R0..R15) (including PC=R15)
When above Bit 25 I=0 (Register as 2nd Operand)
  When below Bit 4 R=0 - Shift by Immediate
    11-7   Is - Shift amount   (1-31, 0=Special/See below)
  When below Bit 4 R=1 - Shift by Register
    11-8   Rs - Shift register (R0-R14) - only lower 8bit 0-255 used
    7      Reserved, must be zero  (otherwise multiply or undefined opcode)
  6-5    Shift Type (0=LSL, 1=LSR, 2=ASR, 3=ROR)
  4      R - Shift by Register Flag (0=Immediate, 1=Register)
  3-0    Rm - 2nd Operand Register (R0..R15) (including PC=R15)
When above Bit 25 I=1 (Immediate as 2nd Operand)
  11-8   Is - ROR-Shift applied to nn (0-30, in steps of 2)
```

```
  7-0    nn - 2nd Operand Unsigned 8bit Immediate
```

Second Operand (Op2)
This may be a shifted register, or a shifted immediate. See Bit 25 and 11-0.
Unshifted Register: Specify Op2 as "Rm", assembler converts to "Rm,LSL#0".
Shifted Register: Specify as "Rm,SSS#Is" or "Rm,SSS Rs" (SSS=LSL/LSR/ASR/ROR).
Immediate: Specify as 32bit value, for example: "#000NN000h", assembler should automatically convert into "#0NNh,ROR#0ssh" as far as possible (ie. as far as a section of not more than 8bits of the immediate is non-zero).

Zero Shift Amount (Shift Register by Immediate, with Immediate=0)
LSL#0: No shift performed, ie. directly Op2=Rm, the C flag is NOT affected.
LSR#0: Interpreted as LSR#32, ie. Op2 becomes zero, C becomes Bit 31 of Rm.
ASR#0: Interpreted as ASR#32, ie. Op2 and C are filled by Bit 31 of Rm.
ROR#0: Interpreted as RRX#1 (RCR), like ROR#1, but Op2 Bit 31 set to old C.
In source code, LSR#32, ASR#32, and RRX#1 should be specified as such - attempts to specify LSR#0, ASR#0, or ROR#0 will be internally converted to LSL#0 by the assembler.

Using R15 (PC)
When using R15 as Destination (Rd), note below CPSR description and Execution time description.
When using R15 as operand (Rm or Rn), the returned value depends on the instruction: PC+12 if I=0,R=1 (shift by register), otherwise PC+8 (shift by immediate).

Reserved Opcode Combinations
For TST,TEQ,CMP,CMN: The destination register Rd is ignored - only CPSR flags are affected - these opcodes MUST have S=1 (even though it is not necessary to specify {S} in the source code), the blank space for TST,TEQ,CMP,CMN with S=0 is used for PSR transfers (ARM.6) and BX instruction (ARM.3).
Bit 7 must be zero when I=0 and R=1 - the combination with Bit 7 set is reserved for Swap (ARM.3), Multiply (ARM.7), Multiply Long (ARM.8), and Halfword Transfer (ARM.10) opcodes.
The TEQP instruction of older ARM processors is no longer supported.
For MOV,MVN: The first operand (Rn) is ignored.

Returned CPSR Flags
If S=1, Rd<>R15, logical operations (AND,EOR,TST,TEQ,ORR,MOV,BIC,MVN):

```
V=not affected
C=carryflag of shift operation (not affected if LSL#0 or Rs=00h)
Z=zeroflag of result
N=signflag of result (result bit 31)
```

If S=1, Rd<>R15, arithmetic operations (SUB,RSB,ADD,ADC,SBC,RSC,CMP,CMN):

```
V=overflowflag of result
C=carryflag of result
Z=zeroflag of result
N=signflag of result (result bit 31)
```

If S=1, Rd=R15; should not be used in user mode:

```
CPSR = SPSR_<current mode>
PC = result
For example: MOVS PC,R14  ;return from SWI (PC=R14_svc, CPSR=SPSR_svc).
```

If S=0: Flags are not affected (not allowed for CMP,CMN,TEQ,TST).

The instruction "MOV R0,R0" is used as "NOP" opcode in 32bit ARM state.
Execution Time: (1+p)S+rI+pN. Whereas r=1 if I=0 and R=1 (ie. shift by register); otherwise r=0. And p=1 if Rd=R15; otherwise p=0.

ARM.6: PSR Transfer (MRS, MSR)

Opcode Format
These instructions occupy an unused area (TEQ,TST,CMP,CMN with S=0) of Data Processing opcodes (ARM.5).

```
Bit    Expl.
31-28  Condition
27-26  Must be 00b for this instruction
25     I - Immediate Operand Flag  (0=Register, 1=Immediate) (Zero for MRS)
24-23  Must be 10b for this instruction
22     Psr - Source/Destination PSR  (0=CPSR, 1=SPSR_<current mode>)
21     Opcode
         0: MRS{cond} Rd,Psr          ;Rd = Psr
         1: MSR{cond} Psr{_field},Op  ;Psr[field] = Op
20     Must be 0b for this instruction (otherwise TST,TEQ,CMP,CMN)
For MRS:
  19-16   Must be 1111b for this instruction (otherwise SWP)
  15-12   Rd - Destination Register  (R0-R14)
  11-0    Not used, must be zero.
For MSR:
  19      f  write to flags field     Bit 31-24 (aka _flg)
  18      s  write to status field    Bit 23-16 (reserved, don't change)
  17      x  write to extension field Bit 15-8  (reserved, don't change)
  16      c  write to control field   Bit 7-0   (aka _ctl)
  15-12   Not used, must be 1111b.
For MSR Psr,Rm (I=0)
  11-4    Not used, must be zero. (otherwise BX)
  3-0     Rm - Source Register <op>  (R0-R14)
For MSR Psr,Imm (I=1)
  11-8    Shift applied to Imm   (ROR in steps of two 0-30)
  7-0     Imm - Unsigned 8bit Immediate
  In source code, a 32bit immediate should be specified as operand.
  The assembler should then convert that into a shifted 8bit value.
```

The field mask bits specify which bits of the destination Psr are write-able (or write-protected), one or more of these bits should be set, for example, CPSR_fsxc (aka CPSR aka CPSR_all) unlocks all bits (see below user mode restriction though).
Restrictions:
In non-privileged mode (user mode): only condition code bits of CPSR can be changed, control bits can't.
Only the SPSR of the current mode can be accessed; In User and System modes no SPSR exists.
The T-bit may not be changed; for THUMB/ARM switching use BX instruction.
Unused Bits in CPSR are reserved for future use and should never be changed (except for unused bits in the flags field).
Execution Time: 1S.

Note: The A22i assembler recognizes MOV as alias for both MSR and MRS because it is practically not possible to remember whether MSR or MRS was the load or store opcode, and/or whether it does load to or from the Psr register.

ARM.7: Multiply and Multiply-Accumulate (MUL, MLA)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-22  Must be 000000b for this instruction
21     Opcode (0-1)
         0: MUL{cond}{S} Rd,Rm,Rs     ;multiply                Rd = Rm*Rs
         1: MLA{cond}{S} Rd,Rm,Rs,Rn  ;multiply and accumulate Rd = Rm*Rs+Rn
```

```
20     S - Set Condition Codes (0=No, 1=Yes)
19-16  Rd - Destination Register (R0-R14)
15-12  Rn - Operand Register (R0-R14) (Used for MLA only, for MUL set to R0)
11-8   Rs - Operand Register (R0-R14)
7-4    Must be 1001b for this instruction
3-0    Rm - Operand Register (R0-R14)
```

Restrictions: Rd may not be same as Rm. Rd,Rn,Rs,Rm may not be R15.
Note: Only the lower 32bit of the internal 64bit result are stored in Rd, thus no sign/zero extension is required and MUL and MLA can be used for both signed and unsigned calculations!
Execution Time: 1S+mI for MUL, and 1S+(m+1)I for MLA. Whereas 'm' depends on whether/how many most significant bits of Rs are all zero or all one. That is m=1 for Bit 31-8, m=2 for Bit 31-16, m=3 for Bit 31-24, and m=4 otherwise.
Flags (if S=1): Z=zeroflag, N=signflag, C=destroyed, V=not affected.

ARM.8: Multiply Long and Multiply-Accumulate Long (MULL, MLAL)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-23  Must be 00001b for this instruction
22-21  Opcode (0-3)
         0: UMULL{cond}{S} RdLo,RdHi,Rm,Rs  ;multiply   RdHiLo = Rm*Rs
         1: UMLAL{cond}{S} RdLo,RdHi,Rm,Rs  ;mul.& acc. RdHiLo = Rm*Rs+RdHiLo
         2: SMULL{cond}{S} RdLo,RdHi,Rm,Rs  ;sign.mul.  RdHiLo = Rm*Rs
         3: SMLAL{cond}{S} RdLo,RdHi,Rm,Rs  ;sign.m&a.  RdHiLo = Rm*Rs+RdHiLo
20     S - Set Condition Codes (0=No, 1=Yes)
19-16  RdHi - Source/Destination Register High (R0-R14)
15-12  RdLo - Source/Destination Register Low  (R0-R14)
11-8   Rs - Operand Register (R0-R14)
7-4    Must be 1001b for this instruction
3-0    Rm - Operand Register (R0-R14)
```

Restrictions: RdHi,RdLo,Rm must be different registers. R15 may not be used.

Execution Time: 1S+(m+1)I for MULL, and 1S+(m+2)I for MLAL. Whereas 'm' depends on whether/how many most significant bits of Rs are "all zero" (UMULL/UMLAL) or "all zero or all one" (SMULL,SMLAL). That is m=1 for Bit 31-8, m=2 for Bit 31-16, m=3 for Bit 31-24, and m=4 otherwise.

Flags (if S=1): Z=zeroflag, N=signflag, C=destroyed, V=destroyed???.

ARM.9: Single Data Transfer (LDR, STR)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-26  Must be 01b for this instruction
25     I - Immediate Offset Flag (0=Immediate, 1=Shifted Register)
24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
22     B - Byte/Word bit (0=transfer word quantity, 1=transfer byte quantity)
When above Bit 24 P=0 (Post-indexing, write-back is ALWAYS enabled):
  21     T - Memory Managment (0=Normal, 1=Force non-privileged access)
When above Bit 24 P=1 (Pre-indexing, write-back is optional):
  21     W - Write-back bit (0=no write-back, 1=write address into base)
20     L - Load/Store bit (0=Store to memory, 1=Load from memory)
```

```
        0: STR{cond}{B}{T} Rd,<Address>   ;[Rn+/-<offset>]=Rd
        1: LDR{cond}{B}{T} Rd,<Address>   ;Rd=[Rn+/-<offset>]
        Whereas, B=Byte, T=Force User Mode (only for POST-Indexing)
19-16  Rn - Base register               (R0..R15) (including R15=PC+8)
15-12  Rd - Source/Destination Register (R0..R15) (including R15=PC+12)
When above I=0 (Immediate as Offset)
  11-0   Unsigned 12bit Immediate Offset (0-4095, steps of 1)
When above I=1 (Register shifted by Immediate as Offset)
  11-7   Is - Shift amount      (1-31, 0=Special/See below)
  6-5    Shift Type             (0=LSL, 1=LSR, 2=ASR, 3=ROR)
  4      Must be 0 (Reserved, see ARM.17, The Undefined Instruction)
  3-0    Rm - Offset Register   (R0..R14) (not including PC=R15)
```

Instruction Formats for <Address>

An expression which generates an address:

```
<expression>                  ;an immediate used as address
;*** restriction: must be located in range PC+/-4095+8, if so,
;*** assembler will calculate offset and use PC (R15) as base.
```

Pre-indexed addressing specification:

```
[Rn]                          ;offset = zero
[Rn, <#{+/-}expression>]{!}   ;offset = immediate
[Rn, {+/-}Rm{,<shift>} ]{!}   ;offset = register shifted by immediate
```

Post-indexed addressing specification:

```
[Rn], <#{+/-}expression>      ;offset = immediate
[Rn], {+/-}Rm{,<shift>}       ;offset = register shifted by immediate
```

Whereas...

```
<shift>  immediate shift such like LSL#4, ROR#2, etc. (see ARM.5).
{!}      exclamation mark ("!") indicates write-back (Rn will be updated).
```

Notes

Shift amount 0 has special meaning, as described in ARM.5 Data Processing.
When writing a word (32bit) to memory, the address should be word-aligned.
When reading a byte from memory, upper 24 bits of Rd are zero-extended.

When reading a word from a halfword-aligned address (which is located in the middle between two word-aligned addresses), the lower 16bit of Rd will contain [address] ie. the addressed halfword, and the upper 16bit of Rd will contain [Rd-2] ie. more or less unwanted garbage. However, by isolating lower bits this may be used to read a halfword from memory. (Above applies to little endian mode, as used in GBA.)

In a virtual memory based environment (ie. not in the GBA), aborts (ie. page faults) may take place during execution, if so, Rm and Rn should not specify the same register when post-indexing is used, as the abort-handler might have problems to reconstruate the original value of the register.

Return: CPSR flags are not affected.
Execution Time: For normal LDR: 1S+1N+1I. For LDR PC: 2S+2N+1I. For STR: 2N.

ARM.10: Halfword and Signed Data Transfer (STRH,LDRH,LDRSB,LDRSH)

Opcode Format

```
Bit    Expl.
31-28  Condition
```

```
27-25  Must be 000b for this instruction
24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
22     I - Immediate Offset Flag (0=Register Offset, 1=Immediate Offset)
When above Bit 24 P=0 (Post-indexing, write-back is ALWAYS enabled):
  21     Not used, must be zero (0)
When above Bit 24 P=1 (Pre-indexing, write-back is optional):
  21     W - Write-back bit (0=no write-back, 1=write address into base)
20     L - Load/Store bit (0=Store to memory, 1=Load from memory)
19-16  Rn - Base register                (R0-R15) (Including R15=PC+8)
15-12  Rd - Source/Destination Register  (R0-R15) (Including R15=PC+12)
11-8   When above Bit 22 I=0 (Register as Offset):
         Not used. Must be 0000b
       When above Bit 22 I=1 (immediate as Offset):
         Immediate Offset (upper 4bits)
7      Reserved, must be set (1)
6-5    Opcode (0-3)
       When Bit 20 L=0 (Store):
        0: Reserved for SWP instruction (see ARM.12 Single Data Swap)
        1: STR{cond}H  Rd,<Address>   ;Store halfword
        2: Reserved.
        3: Reserved.
       When Bit 20 L=1 (Load):
        0: Reserved.
        1: LDR{cond}H  Rd,<Address>   ;Load Unsigned halfword (zero-extended)
        2: LDR{cond}SB Rd,<Address>   ;Load Signed byte (sign extended)
        3: LDR{cond}SH Rd,<Address>   ;Load Signed halfword (sign extended)
4      Reserved, must be set (1)
3-0    When above Bit 22 I=0:
         Rm - Offset Register            (R0-R14) (not including R15)
       When above Bit 22 I=1:
         Immediate Offset (lower 4bits)  (0-255, together with upper bits)
```

Instruction Formats for <Address>

An expression which generates an address:

```
<expression>                  ;an immediate used as address
;*** restriction: must be located in range PC+/-255+8, if so,
;*** assembler will calculate offset and use PC (R15) as base.
```

Pre-indexed addressing specification:

```
[Rn]                          ;offset = zero
[Rn, <#{+/-}expression>]{!}   ;offset = immediate
[Rn, {+/-}Rm]{!}              ;offset = register
```

Post-indexed addressing specification:

```
[Rn], <#{+/-}expression>      ;offset = immediate
[Rn], {+/-}Rm                 ;offset = register
```

Whereas...

```
{!}      exclamation mark ("!") indicates write-back (Rn will be updated).
```

Return: No Flags affected.
Execution Time: For Normal LDR, 1S+1N+1I. For LDR PC, 2S+2N+1I. For STRH 2N.

ARM.11: Block Data Transfer (LDM,STM)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-25  Must be 100b for this instruction
```

```
24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
22     S - PSR & force user bit (0=No, 1=load PSR or force user mode)
21     W - Write-back bit (0=no write-back, 1=write address into base)
20     L - Load/Store bit (0=Store to memory, 1=Load from memory)
        0: STM{cond}{amod} Rn{!},<Rlist>{^}  ;Store (Push)
        1: LDM{cond}{amod} Rn{!},<Rlist>{^}  ;Load  (Pop)
        Whereas, {!}=Write-Back (W), and {^}=PSR/User Mode (S)
19-16  Rn - Base register                (R0-R14) (not including R15)
15-0   Rlist - Register List
(Above 'offset' is meant to be the number of words specified in Rlist.)
```

Addressing Modes {amod}

The IB,IA,DB,DA suffixes directly specify the desired U and P bits:

```
IB  increment before          ;P=1, U=1
IA  increment after           ;P=0, U=1
DB  decrement before          ;P=1, U=0
DA  decrement after           ;P=0, U=0
```

Alternately, FD,ED,FA,EA could be used, mostly to simplify mnemonics for stack transfers.

```
ED  empty stack, descending   ;LDM: P=1, U=1  ;STM: P=0, U=0
FD  full stack,  descending   ;     P=0, U=1  ;     P=1, U=0
EA  empty stack, ascending    ;     P=1, U=0  ;     P=0, U=1
FA  full stack,  ascending    ;     P=0, U=0  ;     P=1, U=1
```

Ie. the following expressions are aliases for each other:

```
STMFD=STMDB=PUSH   STMED=STMDA   STMFA=STMIB   STMEA=STMIA
LDMFD=LDMIA=POP    LDMED=LDMIB   LDMFA=LDMDA   LDMEA=LDMDB
```

Note: The equivalent THUMB functions use fixed organization:

```
PUSH/POP: full descending     ;base register SP (R13)
LDM/STM:  increment after     ;base register R0..R7
```

Descending is common stack organization as used in 80x86 and Z80 CPUs, SP is decremented when pushing/storing data, and incremented when popping/loading data.

When S Bit is set (S=1)

If instruction is LDM and R15 is in the list: (Mode Changes)

```
While R15 loaded, additionally: CPSR=SPSR_<current mode>
```

Otherwise: (User bank transfer)

```
Rlist is referring to User Bank Registers, R0-R15 (rather than
register related to the current mode, such like R14_svc etc.)
Base write-back should not be used for User bank transfer.
!  When instruction is LDM:                                    !
!  If the following instruction reads from a banked register, !
!  like R14_svc, then CPU might still read R14 instead. If    !
!  necessary insert a dummy instruction such like MOV R0,R0.  !
```

Notes

The lowest Register in Rlist (R0 if its in the list) will be loaded/stored to/from the lowest memory address. The base address should be usually word-aligned.

Inclusion of the base in the register list

When write-back is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value.
A LDM will always overwrite the updated base if the base is in the list.

Return: No Flags affected.
Execution Time: For normal LDM, nS+1N+1I. For LDM PC, (n+1)S+2N+1I. For STM (n-1)S+2N.
Where n is the number of words transferred.

ARM.12: Single Data Swap (SWP)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-23  Must be 00010b for this instruction
       Opcode (fixed)
         SWP{cond}{B} Rd,Rm,[Rn]      ;Rd=[Rn], [Rn]=Rm
22     B - Byte/Word bit (0=swap word quantity, 1=swap byte quantity)
21-20  Must be 00b for this instruction
19-16  Rn - Base register                    (R0-R14)
15-12  Rd - Destination Register             (R0-R14)
11-4   Must be 00001001b for this instruction
3-0    Rm - Source Register                  (R0-R14)
```

Swap works properly including if Rm and Rn specify the same register.
R15 may not be used for either Rn,Rd,Rm. (Rn=R15 would be MRS opcode).
Upper bits of Rd are zero-expanded when using Byte quantity. For info about byte and word data memory addressing, read LDR and STR opcode description.
Execution Time: 1S+2N+1I.

ARM.13: Software Interrupt (SWI)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-24  Must be 1111b for this instruction
       Opcode (fixed)
         SWI{cond} nn
23-0   nn - Comment Field, ignored by processor (24bit value)
```

Supposed for calls to the operating system - Enter Supervisor mode (SVC) in ARM state.
Execution:

```
R14_svc=PC+4   ;save return address
SPSR_svc=CPSR  ;save CPSR flags
CPSR=<changed> ;Enter Supervisor mode (svc) in ARM state
PC=00000008h   ;jump to SWI vector address
```

Execution Time: 2S+1N.
Interpreting the Comment Field:
The immediate parameter is ignored by the processor, the user interrupt handler may read-out this number by examing the lower 24bit of the 32bit opcode opcode at [R14_svc-4]. In case that your program executes SWI's from inside of THUMB mode also: Your SWI handler must then examine the T Bit SPSR_svc in order to determine whether it's been a THUMB SWI - if so, examing the lower 8bit of the 16bit opcode opcode at [R14_svc-2].
For Returning from SWI use this instruction:

```
MOVS PC,R14
```

That instructions does both restoring PC and CPSR, ie. PC=R14_svc, and CPSR=SPRS_svc.
Nesting SWIs:

Obviously, SPSR_svc and R14_svc may store only old flags and return address from current SWI, so, the SWI handler must push these values before nesting SWIs.

ARM.14: Coprocessor Data Operations (CDP)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-24  Must be 1110b for this instruction
       ARM-Opcode (fixed)
         CDP{cond} p#,<cpopc>,cd,cn,cm{,<cp>}
23-20  CP Opc - Coprocessor operation code       (0-15)
19-16  CRn    - Coprocessor operand Register     (CR0-CR15)
15-12  CRd    - Coprocessor destination Register (CR0-CR15)
11-8   CP#    - Coprocessor number               (0-15)
7-5    CP     - Coprocessor information          (0-7)
4      Reserved, must be zero
3-0    CRm    - Coprocessor operand Register     (CR0-CR15)
```

Execution time: 1S+bI, b=number of cycles in coprocessor busy-wait loop.
Return: No flags affected, no ARM-registers used/modified.
For details refer to original ARM docs, irrelevant in GBA because no coprocessor exists.

ARM.15: Coprocessor Data Transfers (LDC,STC)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-25  Must be 110b for this instruction
24     P - Pre/Post (0=post; add offset after transfer, 1=pre; before trans.)
23     U - Up/Down Bit (0=down; subtract offset from base, 1=up; add to base)
22     N - Transfer length (0-1, interpretation depends on co-processor)
21     W - Write-back bit (0=no write-back, 1=write address into base)
20     Opcode (0-1)
        0: STC{cond}{L} p#,cd,<Address>  ;Store to memory (from coprocessor)
        1: LDC{cond}{L} p#,cd,<Address>  ;Read from memory (to coprocessor)
        whereas {L} indicates long transfer (Bit 22: N=1)
19-16  Rn     - ARM Base Register              (R0-R15)     (R15=PC+8)
15-12  CRd    - Coprocessor src/dest Register  (CR0-CR15)
11-8   CP#    - Coprocessor number             (0-15)
7-0    Offset - Unsigned Immediate, step 4     (0-1020, in steps of 4)
```

Execution time: (n-1)S+2N+bI, n=number of words transferred.
For details refer to original ARM docs, irrelevant in GBA because no coprocessor exists.

ARM.16: Coprocessor Register Transfers (MRC, MCR)

Opcode Format

```
Bit    Expl.
31-28  Condition
27-24  Must be 1110b for this instruction
23-21  CP Opc - Coprocessor operation mode         (0-7)
20     ARM-Opcode (0-1)
        0: MCR{cond} p#,<cpopc>,Rd,cn,cm{,<cp>}   ;move from ARM to CoPro
        1: MRC{cond} p#,<cpopc>,Rd,cn,cm{,<cp>}   ;move from CoPro to ARM
19-16  CRn    - Coprocessor source/dest. Register  (CR0-CR15)
```

```
15-12  Rd     - ARM source/destination Register    (R0-R15)
11-8   CP#    - Coprocessor number                 (0-15)
7-5    CP     - Coprocessor information            (0-7)
4      Reserved, must be one (1)
3-0    CRm    - Coprocessor operand Register       (CR0-CR15)
```

When using MCR with R15: Coprocessor will receive a data value of PC+12.
When using MRC with R15: Bit 31-28 of data are copied to Bit 31-28 of CPSR (ie. N,Z,C,V flags), other data bits are ignored, CPSR Bit 27-0 are not affected, R15 (PC) is not affected.
Execution time: 1S+bI+1C for MCR, 1S+(b+1)I+1C for MRC.
Return: For MRC only: Either R0-R14 modified, or flags affected (see above).
For details refer to original ARM docs, irrelevant in GBA because no coprocessor exists.

ARM.17: Undefined Instruction

Opcode Format

```
Bit    Expl.
31-28  Condition
27-25  Must be 011b for this instruction
24-5   Reserved for future use
4      Must be 1b for this instruction
3-0    Reserved for future use
```

Execution time: 2S+1I+1N.
No assembler mnemonic exists, reserved for future implementations.

Pseudo Instructions and Directives

ARM Pseudo Instructions

```
nop              mov r0,r0
ldr Rd,=Imm      ldr Rd,[r15,disp] ;use .pool as parameter field)
add Rd,=addr     add/sub Rd,r15,disp
adr Rd,addr      add/sub Rd,r15,disp
adrl Rd,addr     two add/sub opcodes with disp=xx00h+00yyh
mov Rd,Imm       mvn Rd,NOT Imm    ;or vice-versa
and Rd,Rn,Imm    bic Rd,Rn,NOT Imm ;or vice-versa
cmp Rd,Rn,Imm    cmn Rd,Rn,-Imm    ;or vice-versa
add Rd,Rn,Imm    sub Rd,Rn,-Imm    ;or vice-versa
```

All above opcodes may be made conditional by specifying a {cond} field.

THUMB Pseudo Instructions

```
nop              mov r8,r8
ldr Rd,=Imm      ldr Rd,[r15,disp] ;use .pool as parameter field
add Rd,=addr     add Rd,r15,disp
adr Rd,addr      add Rd,r15,disp
mov Rd,Rs        add Rd,Rs,0       ;with Rd,Rs in range r0-r7 each
```

A22i Directives

```
org  adr     assume following code from this address on
.gba         indicate GBA program
.fix         fix GBA header checksum
.norewrite   do not delete existing output file (keep following data in file)
.data?       following defines RAM data structure (assembled to nowhere)
.code        following is normal ROM code/data (assembled to ROM image)
.include     includes specified source code file (no nesting/error handling)
.import      imports specified binary file (optional parameters: ,begin,len)
```

```
.if expr     assembles following code only if expression is nonzero
.else        invert previous .if condition
.endif       terminate .if
.ifdef sym   assemble following only if symbol is defined
.ifndef sym  assemble following only if symbol is not defined
.align nn    aligns to an address divedable-by-nn, inserts 00's
.msg         defines a no$gba debugmessage string, such like .msg 'Init Okay'
.brk         defines a no$gba source code break opcode
l equ n      l=n
l:   [cmd]   l=$   (global label)
@@l: [cmd]   @@l=$ (local label, all locals are reset at next global label)
end          end of source code
db ...       define 8bit data (bytes)
dw ...       define 16bit data (halfwords)
dd ...       define 32bit data (words)
defs nn      define nn bytes space (zero-filled)
;...         defines a comment (ignored by the assembler)
//           alias for CRLF, eg. allows <db 'Text',0 // dw addr> in one line
```

Alias Directives

```
align        .align 4          code16    .thumb
align nn     .align nn         .code 16  .thumb
% nn         defs nn           code32    .arm
.space nn    defs nn           .code 32  .arm
..ds nn      defs nn           ltorg     .pool
x=n          x equ n           .ltorg    .pool
.equ x,n     x equ n           ..ltorg   .pool
.define x n  x equ n           dcb       db (8bit data)
incbin       .import           defb      db (8bit data)
@@@...       ;comment          .byte     db (8bit data)
@ ...        ;comment          .ascii    db (8bit string)
@*...        ;comment          dcw       dw (16bit data)
@...         ;comment          defw      dw (16bit data)
.text        .code             .hword    dw (16bit data)
.bss         .data?            dcd       dd (32bit data)
.global      (ignored)         defd      dd (32bit data)
.extern      (ignored)         .long     dd (32bit data)
.thumb_func  (ignored)         .word     dw/dd, don't use
#directive   .directive        .end      end
.fill nn,1,0 defs nn
```

Alias Conditions, Opcodes, Operands

```
hs   cs   ;condition higher or same = carry set
asl  lsl  ;arithmetic shift left = logical shift left
```

Numeric Formats & Dialects

```
Type          Normal       Alias
Decimal       85           #85
Hexadecimal   55h          #55h  0x55  #0x55  $55
Ascii         'U'          "U"
Binary        01010101b    %01010101
```

The Nocash Syntax

Even though A22i does recognize the official ARM syntax, it's also allowing to use friendly code:

```
mov   r0,0ffh         ;no C64-style "#", and no C-style "0x" required
stmia [r7]!,r0,r4-r5  ;square [base] brackets, no fancy {rlist} brackets
mov   r0,cpsr         ;no confusing MSR and MRS (whatever which is which)
ldr   r0,[score]      ;allows to use clean brackets for relative addresses
push  rlist           ;alias for stmfd [r13]!,rlist (and same for pop/ldmfd)
label:                ;label definitions recommended to use ":" colons
```

CPU Instruction Cycle Times

Instruction Cycle Summary

Instruction	Cycles	Additional
Data Processing	1S	+1S+1N if R15 loaded, +1I if SHIFT(Rs)
MSR,MRS	1S	
LDR	1S+1N+1I	+1S+1N if R15 loaded
STR	2N	
LDM	nS+1N+1I	+1S+1N if R15 loaded
STM	(n-1)S+2N	
SWP	1S+2N+1I	
BL (THUMB)	3S+1N	
B,BL	2S+1N	
SWI,trap	2S+1N	
MUL	1S+ml	
MLA	1S+(m+1)I	
MULL	1S+(m+1)I	
MLAL	1S+(m+2)I	
CDP	1S+bI	
LDC,STC	(n-1)S+2N+bI	
MCR	1N+bI+1C	
MRC	1S+(b+1)I+1C	
{cond} false	1S	

Whereas,

```
n = number of words transferred
b = number of cycles spent in coprocessor busy-wait loop
m = depends on most significant byte(s) of multiplier operand
```

Above 'trap' is meant to be the execution time for exeptions. And '{cond} false' is meant to be the execution time for conditional instructions which haven't been actually executed because the condition has been false.

The separate meaning of the N,S,I,C cycles is:

N - Non-sequential cycle
Requests a transfer to/from an address which is NOT related to the address used in the previous cycle. (Called 1st Access in GBA language).
The execution time for 1N is 1 clock cycle (plus non-sequential access waitstates).

S - Sequential cycle
Requests a transfer to/from an address which is located directly after the address used in the previous cycle. Ie. for 16bit or 32bit accesses at incrementing addresses, the first access is Non-sequential, the following accesses are sequential. (Called 2nd Access in GBA language).
The execution time for 1S is 1 clock cycle (plus sequential access waitstates).

I - Internal Cycle
CPU is just too busy, not even requesting a memory transfer for now.
The execution time for 1I is 1 clock cycle (without any waitstates).

C - Coprocessor Cycle
The CPU uses the data bus to communicate with the coprocessor (if any), but no memory transfers are requested.

Memory Waitstates

Ideally, memory may be accessed free of waitstates (1N and 1S are then equal to 1 clock cycle each). However, a memory system may generate waitstates for several reasons: The memory may be just too slow. Memory is currently accessed by DMA, eg. sound, video, memory transfers, etc. Or when data is squeezed through a 16bit data bus (in that special case, 32bit access may have more waitstates than 8bit and 16bit accesses). Also, the memory system may separate between S and N cycles (if so, S cycles would be typically faster than N cycles).

Memory Waitstates for Different Memory Areas
Different memory areas (eg. ROM and RAM) may have different waitstates. When executing code in one area which accesses data in another area, then the S+N cycles must be split into code and data accesses: 1N is used for data access, plus (n-1)S for LDM/STM, the remaining S+N are code access. If an instruction jumps to a different memory area, then all code cycles for that opcode are having waitstate characteristics of the NEW memory area (except Thumb BL which still executes 1S in OLD area).

CPU Data Sheet

This present document is an attempt to supply a brief ARM7TDMI reference, hopefully including all information which is relevant for programmers.

Some details that I have treated as meaningless for GBA programming aren't included - such like Big Endian format, and Virtual Memory data aborts, and most of the chapters listed below.

Have a look at the complete data sheet (URL see below) for more detailed verbose information about ARM7TDMI instructions. That document also includes:

- Signal Description
  `Pins of the original CPU, probably other for GBA.`

- Memory Interface
  `Optional virtual memory circuits, etc. not for GBA.`

- Coprocessor Interface
  `As far as I know, none such in GBA.`

- Debug Interface
  `For external hardware-based debugging.`

- ICEBreaker Module
  `For external hardware-based debugging also.`

- Instruction Cycle Operations
  `Detailed: What happens during each cycle of each instruction.`

- DC Parameters (Power supply)
- AC Parameters (Signal timings)

The official ARM7TDMI data sheet can be downloaded from ARMs webpage,
`http://www.arm.com/Documentation/UserMans/PDF/ARM7TDMI.html`

Be prepared for specs in PDF Format, approx 1.3 MB, about 200 pages.

About this Document

About
GBATEK written 2001-2002 by Martin Korth, programming specs for the GBA hardware, I've been trying to keep the specs both as short as possible, and as complete as possible. The document is part of the no$gba debuggers built-in help text.

Updates
The standalone docs in TXT and HTM format should be updated along with each no$gba major update (about each 1-2 months), the no$gba built-in version will be updated more regulary, along with all no$gba major & beta updates (about each 1-2 weeks).

Homepage
http://www.work.de/nocash/gba.htm - no$gba homepage
http://www.work.de/nocash/gbatek.htm - gbatek html version
http://www.work.de/nocash/gbatek.txt - gbatek text version

Feedback
If you find any information in this document to be misleading, incomplete, or incorrect, please say something! My email address will be eventually found on the no$gba webpage - depending on the amount of incoming garbage; mail from programmers only, please.