

# **The Cycle-Accurate Game Boy Docs**

Version 0.X

by Antonio Niño Díaz (AntonioND)



Antonio Niño Díaz (AntonioND), 2014

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.

# Index

1. Introduction.....	4
2. Memory.....	5
General Memory Map.....	5
Jump Vectors in ROM0.....	5
Cartridge Header in ROM0.....	5
External Memory and Hardware.....	5
I/O Register Unreadable Bits.....	6
Boot ROMs.....	6
FF70h – SVBK – GBC Mode – WRAM Bank (R/W).....	6
FF4Fh – VBK – GBC Mode – VRAM Bank (R/W).....	6
Unused Memory Area at E000h – FDFFh.....	6
Unused Memory Area at FEA0h – FEFFh.....	7
3. CPU.....	9
General Information.....	9
Initial Register Values.....	9
CPU Instruction Set.....	10
HALT Mode.....	10
STOP Mode.....	10
Entering STOP Mode correctly.....	10
FF4Dh – KEY1 – GBC Mode – Speed Switch (R/W).....	11
Speed Switch.....	11
Undefined Opcodes.....	12
4. Interrupts.....	13
Vector 0040h – Vertical Blanking Interrupt.....	13
Vector 0048h – LCD STAT Interrupt.....	13
Vector 0050h – Timer Interrupt.....	13
Vector 0058h – Serial Interrupt.....	13
Vector 0060h – Joypad Interrupt.....	13
IME – Interrupt Master Enable Flag.....	13
FF0Fh – IF – Interrupt Flags (R/W).....	14
FFFFh – IE – Interrupt Enable (R/W).....	14
Interrupt Handling.....	14
The HALT Instruction Behaviour.....	15
5. Timer.....	17
FF04h – DIV – Divider Register (R/W*).....	17
Interaction with APU.....	18
FF05h – TIMA – Timer Counter (R/W).....	18
FF06h – TIMA – Timer Modulo (R/W).....	18
FF07h – TAC – Timer Control (R/W).....	18
Timer Obscure Behaviour.....	18
Timer Overflow Behaviour.....	21
6. Serial.....	23
7. Joypad.....	24
FF00h – P1 – Joypad (R/W).....	24
Reading the Joypad.....	24
Joypad Interrupt.....	25
Software-Triggered Joypad Interrupt.....	25
8. Video controller.....	26

The LCD.....	26
FF40h – LCDC – LCD Control (R/W).....	26
FF44h – LY – LCD Current Scanline (R).....	27
FF45h – LYC – LY Compare (R/W).....	27
FF41h – STAT – LCD Status (R/W).....	27
VBL Interrupt.....	27
STAT Interrupt.....	28
LY, LYC, STAT and IF Timings. LY=LYC interrupt.....	28
Timings in DMG.....	29
Timings in CGB in DMG mode.....	30
Timings in CGB in CGB mode (single speed).....	31
Timings in CGB in CGB mode (double speed).....	33
LY, STAT and IF Timings. VBL and STAT (mode 1) interrupts.....	34
LY, STAT and IF Timings. STAT interrupt (mode 2).....	34
LY, STAT and IF Timings. STAT interrupt (mode 0).....	34
FF42h – SCY – BG Scroll Y (R/W).....	34
FF43h – SCX – BG Scroll X (R/W).....	34
FF4Ah – WY – Window Y Position (R/W).....	34
FF4Bh – WX – Window X Position (R/W).....	34
Window Display Depending on WX and WY.....	35
Accessing Unavailable VRAM and OAM.....	35
Reading from OAM in Mode 2.....	35
Reading from OAM in Mode 3.....	35
Reading from VRAM in Mode 3.....	35
Reading from GBC Palette RAM in Mode 2.....	35
Reading from GBC Palette RAM in Mode 3.....	35
9. DMA.....	36
FF46h – DMA – OAM DMA Transfer (R/W).....	36
FF51h – HDMA1 – GBC Mode – HDMA Source, High (W).....	37
FF52h – HDMA2 – GBC Mode – HDMA Source, Low (W).....	37
FF53h – HDMA3 – GBC Mode – HDMA Destination, High (W).....	37
FF54h – HDMA4 – GBC Mode – HDMA Destination, Low (W).....	37
FF55h – HDMA5 – GBC Mode – HDMA Length/Mode/Start (R/W).....	37
GDMA – General Purpose DMA.....	37
HDMA – H-Blank DMA.....	38
GDMA/HDMA Allowed Source Addresses.....	38
10. Audio Processing Unit.....	40
11. The Game Boy Cartridge.....	41
The Cartridge Header.....	41
Memory Bank Controllers.....	43
None (32KB ROM).....	44
MBC1 (2MB ROM. 32KB RAM. DMG, SGB).....	44
MBC2 (256KB ROM. 512 × 4 bits RAM. DMG, SGB).....	45
MBC3 (2MB ROM. 64KB RAM. RTC. DMG, SGB, CGB).....	45
MBC5 (8MB ROM. 128KB RAM. DMG, SGB, CGB).....	47
12. Credits.....	48
13. Changelog.....	49

# 1. Introduction

Since nobody seems to care about documenting the Game Boy good enough to make a cycle-accurate emulator (the only “documentation” there is right now is the source code of the emulator [Gambatte](#), and it requires a lot of time to understand how it works), I've decided to document it myself. The reason is that I'm making my own GB emulator and I just can't find any of the information I want, so I decided to document the GB while I code my own open-source emulator ([GiiBiiAdvance](#)) and check if my assumptions are correct. I started with the timer, but I plan to document the complete Game Boy and Game Boy Color.

I'm using information from the [Pan Docs](#) (a lot from here, in fact), the source code of [Gambatte](#) and some other documents by other authors (all of them in the credits of this file), but the most important source of knowledge about the Game Boy is the Game Boy itself! That's why I've created lots of test ROMs to verify specific behaviours of the original hardware. I have an original GB, a GB Pocket, two different GB Color (both are GBC-D), a GB Advance and a GBA SP. While most of the test give the same results in every hardware, there are some things that are different, so I've tried to document them as detailed as I can.

Here are some abbreviations and nomenclature I'll use in this document:

- DMG/GB: Original Game Boy (Dot Matrix Game)
- MGB/GBP: Game Boy Pocket / Game Boy Light
- SGB/SGB2: Super Game Boy / Super Game Boy 2
- CGB/GBC: Game Boy Color
- AGB/GBA: Game Boy Advance
- AGS/GBA SP: Game Boy Advance SP
- Clock: Oscillator clock frequency is 4194304 Hz (8388608 Hz in double speed mode).
- Cycle: CPU cycle frequency is 1048576 Hz (2097152 Hz in double speed mode).

Whenever I write sample code I'll use [RGBDS](#) syntax, and I'll use the definitions of the file “Gameboy Hardware definitions” (hardware.inc).

## 2. Memory

### General Memory Map

Addresses	Name	Description
0000h – 3FFFh	ROM0	Non-switchable ROM Bank.
4000h – 7FFFh	ROMX	Switchable ROM bank.
8000h – 9FFFh	VRAM	Video RAM, switchable (0-1) in GBC mode.
A000h – BFFFh	SRAM	External RAM in cartridge, often battery buffered.
C000h – CFFFh	WRAM0	Work RAM.
D000h – DFFFh	WRAMX	Work RAM, switchable (1-7) in GBC mode
E000h – FDFh	ECHO	Description of the behaviour below.
FE00h – FE9Fh	OAM	(Object Attribute Table) Sprite information table.
FEA0h – FEFFh	UNUSED	Description of the behaviour below.
FF00h – FF7Fh	I/O Registers	I/O registers are mapped here.
FF80h – FFEh	HRAM	Internal CPU RAM
FFFFh	IE Register	Interrupt enable flags.

### Jump Vectors in ROM0

Some addresses in ROM0 are jump vectors:

0000h, 0008h, 0010h, 0018h, 0020h, 0028h, 0030h, 0038h – For RST instruction of CPU.  
0040h, 0048h, 0050h, 0058h, 0060h – Interrupt Vectors (VBL, LCD, Timer, Serial, Joypad)

Unused vectors may be used for whatever purpose the program wants. RST are special instructions like CALL, but they are 1 byte long instructions (CALL are 3 bytes long) and need 2 cycles less.

### Cartridge Header in ROM0

The data from 100h to 14Fh of the ROM bank 0 is the ROM header. Contains some information about the program and the cartridge used. Some games don't fill this area correctly, though. For more information, read the chapter about it. This is really important because if the checksums here are incorrect the Game Boy won't jump to the cartridge code. The header is also important for enabling GBC and SGB functions (read the chapter for more information).

### External Memory and Hardware

Areas from 0000h – 7FFFh and A000h – BFFFh are mapped to the cartridge. The first area is usually used for ROM, and the second one is used for RAM and external registers (if any). Writes to

ROM area are interpreted by the Memory Bank Controller (MBC) chip. If there is no cartridge connected both areas will return FFh when read.

## I/O Register Unreadable Bits

All unreadable bits of I/O registers return 1. In general, all unused bits in I/O registers are unreadable so they return 1. Some exceptions are:

- Unknown purpose (if any) registers. Some bits of them can be read and written.
- The IE register (only the 5 lower bits are used, but the upper 3 can hold any value).

## Boot ROMs

All Game Boy models have a small program that is run when the GB is powered on. This program initializes some registers and some memory areas (depending on the model) and performs some checks with values in the ROM header. So far (2014/12) have been dumped:

- DMG: Mapped to 0000h – 00FFh.
- MGB: Mapped to 0000h – 00FFh. It has been dumped from a Game Boy Pocket, it is unknown if Game Boy Light has a different one. Probably not, since DMG and MGB are only 1 byte different.
- SGB: Mapped to 0000h – 00FFh.
- CGB: Mapped to 0000h – 00FFh and 0200h – 08FFh.

## FF70h – SVBK – GBC Mode – WRAM Bank (R/W)

This register selects the WRAM bank mapped to D000h – DFFFh. If bank 0 is selected, the actual mapped bank is 1, but the register will still read 0. The bank at C000h – CFFFh is always 0.

Only the lower 3 bits are (R/W), the rest return '1'. In DMG and GBC in DMG mode it returns FFh.

Bits 0-2 - Select WRAM Bank

## FF4Fh – VBK – GBC Mode – VRAM Bank (R/W)

This register selects the VRAM bank mapped to 8000h – 9FFFh.

Only the lower bit is (R/W), the rest return '1'. In DMG it returns Fh. In GBC in DMG mode it returns FEh (always bank 0 selected).

Bit 0 - Select VRAM Bank

Memory at 8000h – 97FFh contains 192 tiles in both banks. Memory at 9800h – 9FFFh contains two 32×32 tile maps in bank 0, and the corresponding attribute maps in bank 1.

## Unused Memory Area at E000h – FDFFh

In CGB/AGB/AGS this area is just a mirror of the WRAM. Reads and writes from this area are redirected to C000h – DDFFh.

In DMG/MGB/(SGB?) this area works a bit different:

- Writes to this area are redirected to both WRAM (C000h – DDFFh) and SRAM (A000h – BDFFh) areas. This is probably caused because both chip enable signals are set to '1' because of a too simple chip selection circuit.

- Reads from this area are calculated by reading from WRAM and SRAM, and then performing a bitwise AND to the two values. IMPORTANT: There is the possibility that this could damage the hardware, as it short-circuits two different signals! This shouldn't happen with writing because the chip data buses are in high-impedance mode.

This means that when SRAM is disabled this is just a mirror of WRAM like CGB (as SRAM would read FFh and writes to SRAM are ignored).

## Unused Memory Area at FEA0h – FFFFh

This small area also has a different behaviour depending on the GB model. It doesn't matter if the Game Boy is in DMG or CGB mode. Reading and writing is restricted to video modes where the OAM memory isn't being accessed by the video hardware, the same way as OAM.

- DMG/MGB(/SGB?): Writes are ignored. Reading from this area returns 00h.

- CGB: There are another 32 bytes at FEA0h – FEBFh. At FEC0h – FECFh there are another 16 bytes that are repeated in FED0h – FEDFh, FEE0h – FEEFh and FEF0h – FFFFh. Reading and writing to any of this 4 blocks will change the same 16 bytes. This is true for revision D of the GBC. For example:

FEA0h	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
FEB0h	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
FEC0h	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
FED0h	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
FEE0h	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
FEF0h	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F

BGB emulates this in a different way, with three groups of 8 bytes. The first 8 bytes are repeated in FEA0h – FEBFh, the second group is repeated in FEC0h – FEDFh and the last one in FEE0h – FFFFh. Example:

FEA0h	00	01	02	03	04	05	06	07	00	01	02	03	04	05	06	07
FEB0h	00	01	02	03	04	05	06	07	00	01	02	03	04	05	06	07
FEC0h	10	11	12	13	14	15	16	17	10	11	12	13	14	15	16	17
FED0h	10	11	12	13	14	15	16	17	10	11	12	13	14	15	16	17
FEE0h	20	21	22	23	24	25	26	27	20	21	22	23	24	25	26	27
FEF0h	20	21	22	23	24	25	26	27	20	21	22	23	24	25	26	27

Other revisions may have different behaviours.

- AGB/AGS: Writes are ignored. Reading from this area returns NNh where N is the high nibble of the lower byte of the address: FENXh. The value of X doesn't matter.



## 3. CPU

### General Information

The Game Boy CPU has an 8-bit data bus and a 16-bit address bus. It has 8 bit and 16 bit registers, and some of the 8 bit registers can be used together to form a 16-bit value and use it like a number or an address:

CPU Registers

16bit	Hi	Lo	Name/Function
AF	A	-	Accumulator & Flags
BC	B	C	
DE	D	E	
HL	H	L	
SP	-	-	Stack Pointer
PC	-	-	Program Counter

Flag Register

Bit	Name	Explanation
7	Z	Zero Flag
6	N	Add/Sub Flag (BCD)
5	H	Half Carry Flag (BCD)
4	C	Carry Flag
3-0	-	Not used (always zero)

Contains information about the last instruction that affected the flags.

The Zero Flag and the Carry Flag are used for conditional instructions. The Carry flag is also used by arithmetic and logic instructions. The BCD Flags are used only by DAA instruction.

The F register can't be accessed normally, only by doing a “push af/pop bc”, for example. The lower four bits are always zero, even if a “pop af” instruction tries to write other values.

### Initial Register Values

The initial register values depend on the Game Boy model. In GBC, GBA and GBA SP they also change if the game is DMG only or not. The values don't depend on the user interaction (during GBC boot ROM if the game is DMG only). To detect the hardware, most games use the values of registers A and B. To check if it is a SGB games send the command packet to enable multiplayer.

DMG mode

Reg	DMG	MGB	SGB	SGB2	CGB	AGB	AGS
AF	01B0h	FFB0h	0100h	FF??h	1180h	1100h	1100h
BC	0013h	0013h	0014h	????h	0000h	0100h	0100h
DE	00D8h	00D8h	0000h	????h	0008h	0008h	0008h
HL	014Dh	014Dh	C060h	????h	007Ch	007Ch	007Ch
SP	FFFEh	FFFEh	FFFEh	????h	FFFEh	FFFEh	FFFEh
PC	0100h	0100h	0100h	0100h	0100h	0100h	0100h

GBC mode

Reg	CGB	AGB	AGS
AF	1180h	1100h	1100h
BC	0000h	0100h	0100h
DE	FF56h	FF56h	FF56h
HL	000Dh	000Dh	000Dh
SP	FFFEh	FFFEh	FFFEh
PC	0100h	0100h	0100h

Note: SGB values haven't been verified on hardware.

# CPU Instruction Set

Random notes:

- EI instruction enables IME the following cycle to its execution. RETI doesn't have this behaviour, it enables them right away. DI doesn't have any delay either.
- HALT, when it doesn't enter the HALT mode, needs 4 clocks to complete.

## HALT Mode

HALT mode is exited when a flag in register IF is set and the corresponding flag in IE is also set, regardless of the value of IME. The only difference is that IME='1' will make the CPU jump to the interrupt vector (and clear the IF flag), while IME='0' will only make the CPU continue executing instructions, but the jump won't be performed (and the IF flag won't be cleared).

...

...

## STOP Mode

During STOP mode all the Game Boy is halted, even peripherals like the sound. The LCD behaves differently in each hardware.

LCD OFF:

- DMG/MGB: LCD OFF. No sound.
- GBC/GBA: Black screen. Sound.

LCD ON:

- DMG/MGB: White screen. No sound.
- GBC/GBA: White screen. Sound.

## Entering STOP Mode correctly

In order to enter it correctly some of the P1 select bits should be selected (to select buttons to exit STOP mode) and IE should be 00h. If no output lines of P1 are selected (set to '0') STOP mode will never exit. Sample code:

```
ld  a,[rIE]
ld  b,a      ; Save IE.
xor a,a
ld  [rIE],a ; Clear IE.
ld  a,$00
ld  [rP1],a ; Select both joypad lines to exit with any button.

stop
```

```
ld a,b
ld [rIE],a ; Restore IE
```

## FF4Dh – KEY1 – GBC Mode – Speed Switch (R/W)

This register is used to prepare the GBC to switch between double and single speed modes. It also tells the current mode the CPU is on. GBC starts in single speed mode.

```
Bit 7 - Current Speed      (0=Normal, 1=Double) (Read Only)
Bit 0 - Prepare Speed Switch (0=No, 1=Prepare)  (Read/Write)
```

The other bits are unused and return '1'. In DMG and GBC in DMG mode this register returns FFh (even though the CPU is in single speed mode).

After switching speed, bit 0 will be cleared automatically.

The only circuits not affected by a speed switch are the video and audio ones.

## Speed Switch

CPU speed switch takes  $128 \times 1024 - 76$  clocks (including the clocks to fetch and execute the STOP instruction). It needs the same clock amount when switching from single speed and from double speed. I've measured it with the timer, so the 76 clocks could be there because at some point during the speed change the clock does weird things. Sample code to switch speed:

```
CPU_fast:
    ld a,[rKEY1]
    bit 7,a
    jr z,CPU_switch ; Check if CPU is in double speed mode.
    ret
```

```
CPU_slow::
    ld a,[rKEY1]
    bit 7,a
    jr nz,CPU_switch ; Check if CPU is in single speed mode.
    ret
```

```
CPU_switch:
    ld a,[rIE]
    ld b,a      ; Save IE.
    xor a,a
    ld [rIE],a  ; Clear IE.
    ld a,$30
    ld [rP1],a  ; Disable joypad lines.
    ld a,$01
    ld [rKEY1],a ; Prepare CPU for speed switch.

    stop        ; Switch speed.

    ld a,b
    ld [rIE],a  ; Restore IE.
    ret
```

- If P1 register bits 4 or 5 are selected (set to '0') and any key is pressed during the speed switch, the CPU will hang, probably because of the STOP mode being cancelled.
- During speed switch, no HDMA is executed, even though all peripherals keeps running, because it depends on the CPU.

## Undefined Opcodes

Undefined opcodes hang the CPU, but sound (even fade in/out and that kind of things) and video hardware keep working. Probably serial keeps working, but that has not been tested.

The exception is opcodes in the form 10h XXh where XXh is not 00h. That format of opcode (corrupted STOP) will switch the LCD on.

## 4. Interrupts

### Vector 0040h – Vertical Blanking Interrupt

This interrupt is triggered when the LCD controller enters V-Blank at scanline 144. This doesn't happen if the LCD is off. For more information read the video controller chapter.

### Vector 0048h – LCD STAT Interrupt

This interrupt can be configured to be triggered when some LCD events happen (like starting to draw a scanline specified in LYC register). For more information read the video controller chapter.

### Vector 0050h – Timer Interrupt

This interrupt is requested when TIMA overflows. There is a delay of one CPU cycle between the overflow and the IF flag being set. For more information read the timer chapter.

### Vector 0058h – Serial Interrupt

This interrupt is requested when a serial transfer of 1 byte is complete. For more information read the serial chapter.

### Vector 0060h – Joypad Interrupt

This interrupt is triggered when there is a transition from '1' to '0' in one of the P1 input lines. For more information read the joypad chapter.

### IME – Interrupt Master Enable Flag

This flag is not mapped to memory and can't be read by any means. The meaning of the flag is not to enable or disable interrupts. In fact, what it does is enable the jump to the interrupt vectors.

0 = Disable jump to interrupt vectors.  
1 = Enable jump to interrupt vectors.

IME can only be set to '1' by the instructions EI and RETI, and can only be set to '0' by DI (and the CPU when jumping to an interrupt vector).

Note that EI doesn't enable the interrupts the same cycle it is executed, but the next cycle:

```
di
ld  a, IEF_TIMER
ld  [rIE], a
ld  [rIF], a
ei
inc a ; This is still executed before jumping to the interrupt vector.
inc a ; This is executed after returning.
ld  [hl+], a
```

## FF0Fh – IF – Interrupt Flags (R/W)

Only the 5 lower bits of this register are (R/W), the others return '1' always when read.

Bit 4	- Joypad Interrupt Requested	(1=Requested)
Bit 3	- Serial Interrupt Requested	(1=Requested)
Bit 2	- Timer Interrupt Requested	(1=Requested)
Bit 1	- LCD STAT Interrupt Requested	(1=Requested)
Bit 0	- Vertical Blank Interrupt Requested	(1=Requested)

Each bit is set to 1 automatically when an internal signal from that subsystem goes from '0' to '1', it doesn't matter if the corresponding bit in IE is set. This is specially important in the case of LCD STAT interrupt, as it will be explained in the video controller chapter. The bits in this register can be set or reset manually too, the CPU will handle them the same way as when they are set by a real event.

## FFFFh – IE – Interrupt Enable (R/W)

All 8 bits of this register are (R/W), but only the 5 lower ones are used by the interrupt handler.

Bit 4	- Joypad Interrupt Enable	(1=Enable, 0=Disable)
Bit 3	- Serial Interrupt Enable	(1=Enable, 0=Disable)
Bit 2	- Timer Interrupt Enable	(1=Enable, 0=Disable)
Bit 1	- LCD STAT Interrupt Enable	(1=Enable, 0=Disable)
Bit 0	- Vertical Blank Interrupt Enable	(1=Enable, 0=Disable)

## Interrupt Handling

Interrupts are checked before fetching a new instruction. If any IF flag and the corresponding IE flag are both '1' and IME is set to '1' too, the CPU will push the current PC into the stack, will jump to the corresponding interrupt vector and set IME to '0'. If IME is '0', this won't happen. This flags are only cleared when the CPU jumps to an interrupt vector because of an interrupt (or IF is written manually).

If 2 or more interrupts are requested at the same time and the corresponding IE bits are set, the vector with lower address has higher priority (vertical blank has the highest priority, joypad the lowest priority).

It takes 20 clocks to dispatch an interrupt. If CPU is in HALT mode, another extra 4 clocks are needed. This timings are the same in every Game Boy model or in double/single speeds in CGB/AGB/AGS.

### Interrupt Service Routine

According to Z80 datasheets, the following occurs when control is being transferred to an interrupt handler:

1. Two wait states are executed (2 machine cycles pass while nothing occurs, presumably the CPU is executing NOPs during this time).
2. The current PC is pushed onto the stack, this process consumes 2 more machine cycles.
3. The high byte of the PC is set to 0, the low byte is set to the address of the handler (\$40,\$48,\$50,\$58,\$60). This consumes one last machine cycle.

The entire ISR should consume a total of 5 machine cycles. This has yet to be tested, but is what the

## Z80 datasheet implies.

If IME='0' and CPU is halted, when any interrupt is triggered by setting any IF flag to '1' with the corresponding bit in IE set to '1', it takes 4 clocks to exit halt mode, even if the CPU doesn't jump to the interrupt vector.

The correct instruction to return from an interrupt vector is RETI, as it returns and enables interrupts in the same instruction. If the program needs to handle interrupts during an interrupt procedure, they can be enabled again with EI.

## The HALT Instruction Behaviour

HALT instruction has three different behaviours depending on IME, IE and IF. It behaves the same way in all Game Boy models.

- IME = 1

HALT executed normally. CPU stops executing instructions until  $(IE \& IF \& 1F) \neq 0$ . When a flag in IF is set and the corresponding IE flag is also set, the CPU jumps to the interrupt vector. The return address pushed to the stack is the next instruction to the HALT, not the HALT itself. The IF flag corresponding to the vector the CPU has jumped in is cleared.

- IME = 0

-  $(IE \& IF \& 1Fh) = 0$

HALT mode is entered. It works like the IME = 1 case, but when a IF flag is set and the corresponding IE flag is also set, the CPU doesn't jump to the interrupt vector, it just continues executing instructions. The IF flags aren't cleared.

-  $(IE \& IF \& 1Fh) \neq 0$

HALT mode is not entered. HALT bug occurs: The CPU fails to increase PC when executing the next instruction. The IF flags aren't cleared. This results on weird behaviour. For example:

One byte long instructions:

```
xor  a,a
halt
inc  a      ; PC fails to increase, it is executed twice.
ld   [hl+],a
```

The equivalent code with a non-bugged HALT instruction would be:

```
xor  a,a
halt
inc  a
inc  a
ld   [hl+],a ; [hl] is set to 2
```

With instructions that need more than one byte the results can be even worse:

```
xor  a,a
halt
ld   a,$14 ; $3E $14 is executed as $3E $3E $14
```

```
ld    [hl+],a
```

The equivalent code with a non-bugged HALT instruction would be:

```
xor    a,a
halt
ld     a,$3E    ; $3E $3E
inc    d        ; $14
ld     [hl+],a ; [hl] is set to $3E, and register d is incremented
```

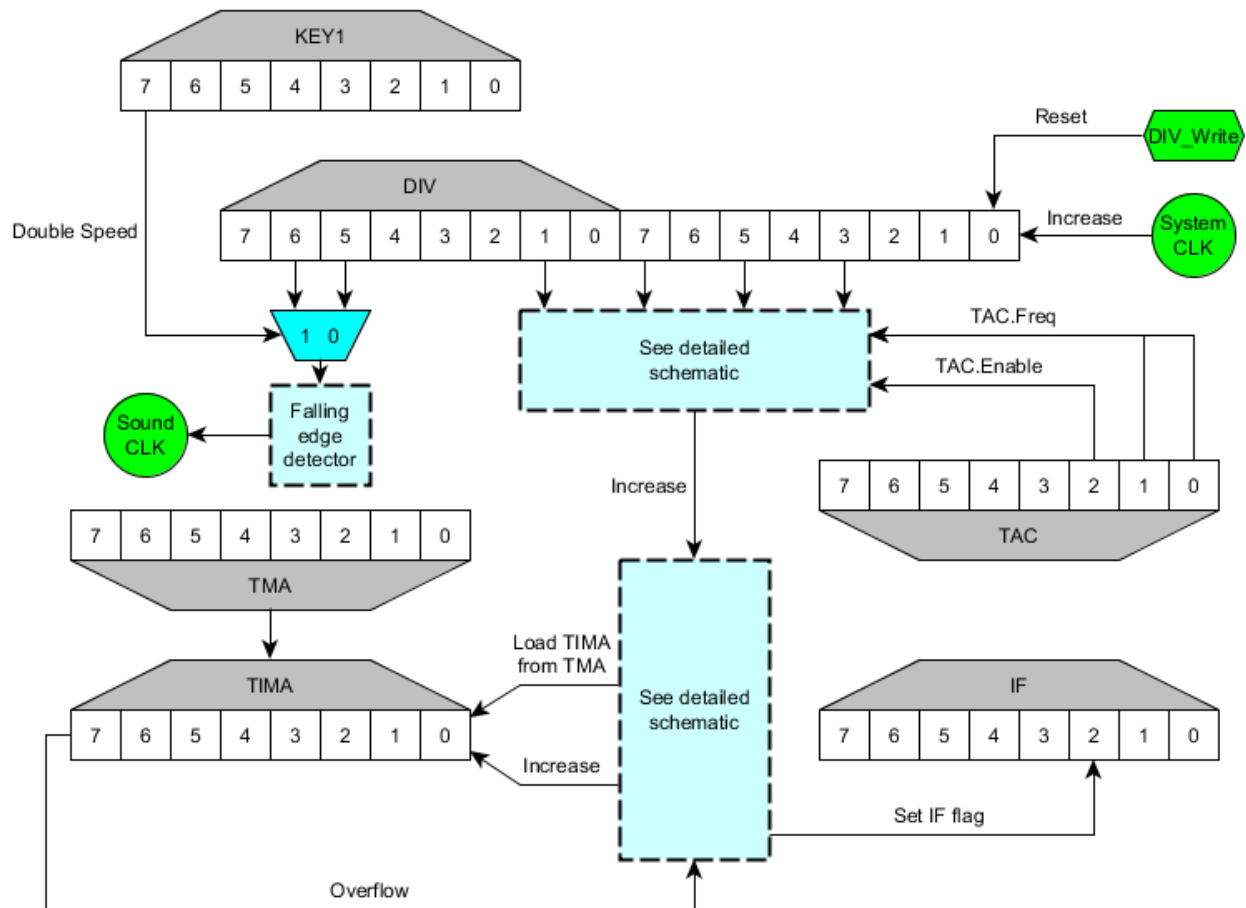
One way to prevent this bug is to always add a NOP instruction after the HALT.

There are no extra clocks as a result of this bug, only the corresponding to the actually executed instructions.



## 5. Timer

The Game Boy timer subsystem is formed by the three timer registers (TIMA, TMA and TAC), and the DIV register. This is a simplified schematic:



### FF04h – DIV – Divider Register (R/W\*)

It works by using an internal system 16 bit counter. The counter increases each clock (4 clocks per nop) and the value of DIV is the 8 upper bits of the counter: it increases every 256 oscillator clocks. The value of DIV is the actual bits of the system internal counter, not a mirror, not a register that increases with the system internal counter: The actual bits of the counter mapped to memory.

DIV overflows at 64 Hz in normal speed mode, 128 Hz in double speed mode. It can be written, but its value resets to 0 no matter what the value written is. In fact, the whole system internal counter is set to 0. DIV is just its the 8 MSB of that counter, so DIV is set to 0 too.

- Serial clock is not derived from this counter, it is not affected by DIV register.

The initial values (when PC=0100h) of the internal counter for the different Game Boy models are:

- DMG/MGB: ABCCh.
- SGB/SGB2: Not tested. Different from DMG (different boot ROM).

- CGB: 1EA0h (GBC game), 267Ch (DMG game, no user interaction during boot).
- AGB/AGS: 1EA4h (GBC game), 2680h (DMG game, no user interaction during boot).

## Interaction with APU

- The APU uses the DIV to update sweep (channel 1), fade in/out and time out, the same way the timer uses it to update itself.
- In normal speed mode the APU updates when bit 5 of DIV goes from 1 to 0 (256 Hz). In double speed mode, bit 6.
- Writing to DIV every instruction, for example, will make the APU produce the same frequency with the same volume even if sweep and fade out are enabled.
- Writing to DIV doesn't affect the frequency itself. The waveform generation is driven by another timer.

## FF05h – TIMA – Timer Counter (R/W)

This register holds an 8 bit value that is the timer counter, it increases at a certain frequency until it overflows. In that moment, the value in register TMA is loaded into TIMA and an interrupt is requested if the corresponding IE flag is set and IME is set. The interrupt vector address is 0050h.

## FF06h – TIMA – Timer Modulo (R/W)

The 8 bit value in this register is loaded into TIMA when it overflows.

## FF07h – TAC – Timer Control (R/W)

This register enables/disables the timer and sets its frequency.

Bit	2	- Timer Enable	(0=Disable, 1=Enable)
Bits	1-0	- Main Frequency Divider	
	00:	4096 Hz (Increase every 1024 clocks)	
	01:	262144 Hz ( " " 16 clocks)	
	10:	65536 Hz ( " " 64 clocks)	
	11:	16386 Hz ( " " 256 clocks)	

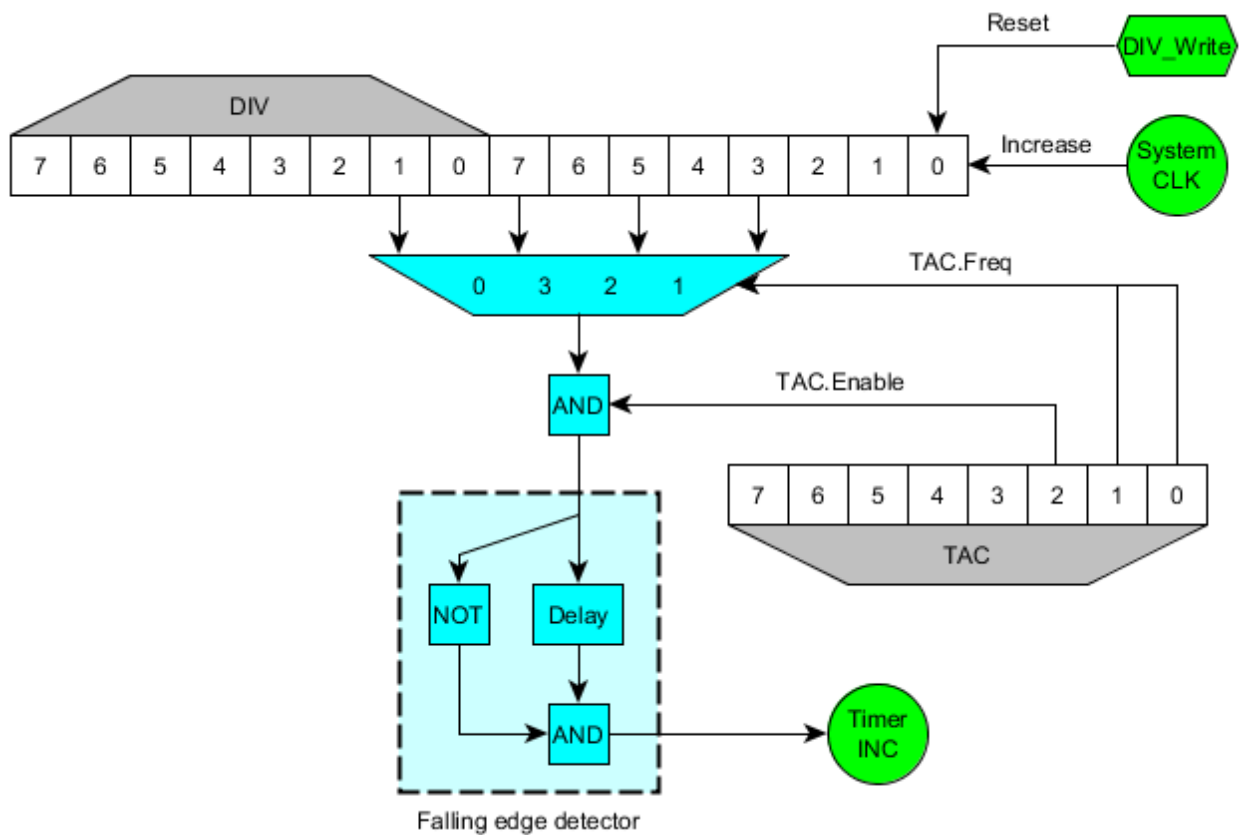
Only the lower 3 bits are (R/W).

## Timer Obscure Behaviour

The signal used to increase the TIMA register is generated in a weird way. It selects the actual value of one bit of the system internal counter, and performs an and operation with the enable bit in TAC. This means that writing to the DIV register affects the timer too (writing to timer registers doesn't affect the DIV register).

- For example, if DIV register is written every 12 clocks, even in the fastest timer configuration, it will never increase. If the timer is configured to increase every 64 clocks but the program writes to it every 20 clocks, it will never increase (but it would if it was configured to increase every 16 clocks).

The next picture is a schematic of the relationship between the DIV register and the timer in a DMG or MGB for a better understanding of its behaviour:



As you can see, TAC register only selects the bit of the system internal counter that will be used to increase the timer counter. Then, it performs a logical AND operation with the enable bit in TAC and that signal goes to a falling edge detector.

For example, if timer is configured to increase every 64 clocks, TAC will select bit 5 of the system internal counter. As soon as it overflows from bit 5 and goes back to 0, the falling edge detector will trigger a pulse to increase TIMA register.

This circuit has three problems:

- When writing to DIV register the TIMA register can be increased if the counter has reached half the clocks it needs to increase because the selected bit by the multiplexer will go from 1 to 0 (which is a falling edge, that will be detected by the falling edge detector).
- When disabling the timer, if the corresponding bit in the system counter is set to 1, the falling edge detector will see a change from 1 to 0, so TIMA will increase. This means that whenever half the clocks of the count are reached, TIMA will increase when disabling the timer.
- When changing TAC register value, if the old selected bit by the multiplexer was 0, the new one is 1, and the new enable bit of TAC is set to 1, it will increase TIMA.

This behaviour is a lot glitchier in newer models. DMG and MGB behave the same way (SGB and SGB2 haven't been tested), but GBC, AGB and AGS (GBA SP) are completely different. In fact, different revisions of the GBC have a different behaviour. The DMG behaviour when writing to TAC is this:

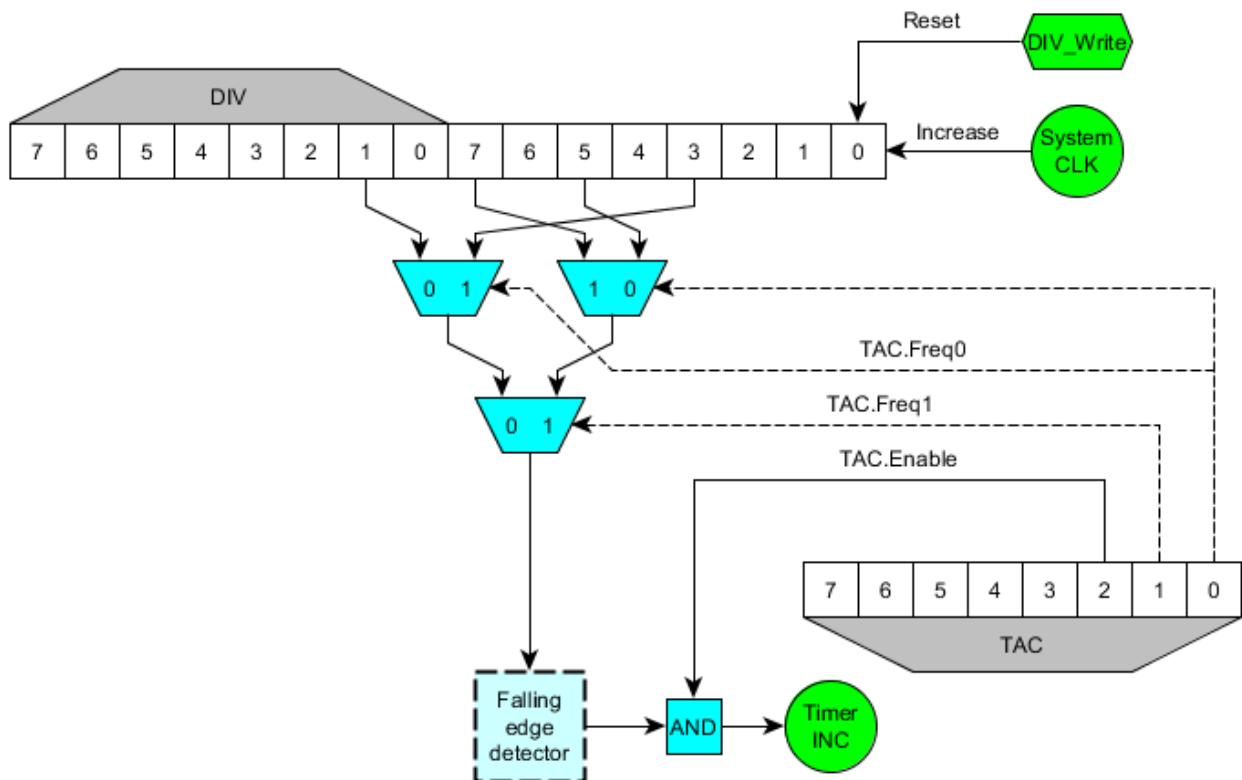
```

IF old_enable == 0 THEN
    glitch = 0 (*)
ELSE
    IF new_enable == 0 THEN
        glitch = (sys_clocks & (old_clocks/2)) != 0
    ELSE
        glitch = ((sys_clocks & (old_clocks/2)) != 0) && ((sys_clocks & (new_clocks/2)) == 0)
    END IF
END IF
END IF

```

For example, if the old TAC value was 6, old\_clocks is 64 and old\_enabled is 1. If the written TAC value is 0, new\_clocks is 1024 and new\_enabled is 0.

The sentence marked with a (\*) has a different behaviour in GBC (AGB and AGS seem to have strange behaviour even in the other statements). When enabling the timer and maintaining the same frequency it doesn't glitch. When disabling the timer it doesn't glitch either. When another change of value happens (so timer is enabled after the write), the behaviour depends on a race condition, so it cannot be predicted for every device. One possible explanation is that the AND gate corresponding to the enable bit is placed after the edge detector, or even that the edge detector is asynchronous, but I'm not sure about this, so the next schematic shouldn't be trusted:



Because all of this, timer initialization should be done carefully. A safe way to set up the timer is:

```

ld a,TACF_16KHZ|TACF_STOP
ld [rTAC],a ; Disable timer and load new frequency.
ld a,___TMA_VALUE___ ; ___TMA_VALUE___ is any byte value.
ld [rTMA],a ; Load modulo value in both
ld [rTIMA],a ; TMA and TIMA registers.
ld a,TACF_16KHZ|TACF_START
ld [rDIV],a ; Synchronize system counter.
ld [rTAC],a ; Enable timer.

```

If the timer interrupt is going to be used:

```

ld b,~IEF_TIMER
ld a,[rIE]
and a,b
ld [rIE],a ; Clear IE timer flag.

xor a,a ; a = 0
ld [rTIMA],a ; Set TIMA to 0
ld a,TACF_16KHZ|TACF_STOP
ld [rTAC],a ; Disable timer and load new frequency.
ld a,___TMA_VALUE___ ; ___TMA_VALUE___ is any byte value.
ld [rTMA],a ; Load modulo value in both
ld [rTIMA],a ; TMA and TIMA registers.

ld a,[rIF]
and a,b
ld [rIF],a ; Clear IF timer flag.
ld b,IEF_TIMER
ld a,[rIE]
or a,b
ld [rIE],a ; Set IE timer flag.

ld a,TACF_16KHZ|TACF_START
ld [rDIV],a ; Synchronize system counter.
ld [rTAC],a ; Enable timer.

```

## Timer Overflow Behaviour

When TIMA overflows, the value from TMA is loaded and IF timer flag is set to 1, but this doesn't happen immediately. Timer interrupt is delayed 1 cycle (4 clocks) from the TIMA overflow. It could be less clocks, but the CPU can't check that. The TMA reload to TIMA is also delayed. For one cycle, after overflowing TIMA, the value in TIMA is 00h, not TMA. This happens only when an overflow happens, not when the upper bit goes from 1 to 0, it can't be done manually writing to TIMA, the timer has to increment itself.

For example (SYS is the system internal counter divided by 4 for easier understanding, each increment of the graph is 1 cycle, not 1 clock):

Timer overflows:

				[A]	[B]		
SYS	FD	FE	FF	00	01	02	03
TIMA	FF	FF	FF	00	23	23	23
TMA	23	23	23	23	23	23	23
IF	E0	E0	E0	E0	E4	E4	E4

Timer doesn't overflow:

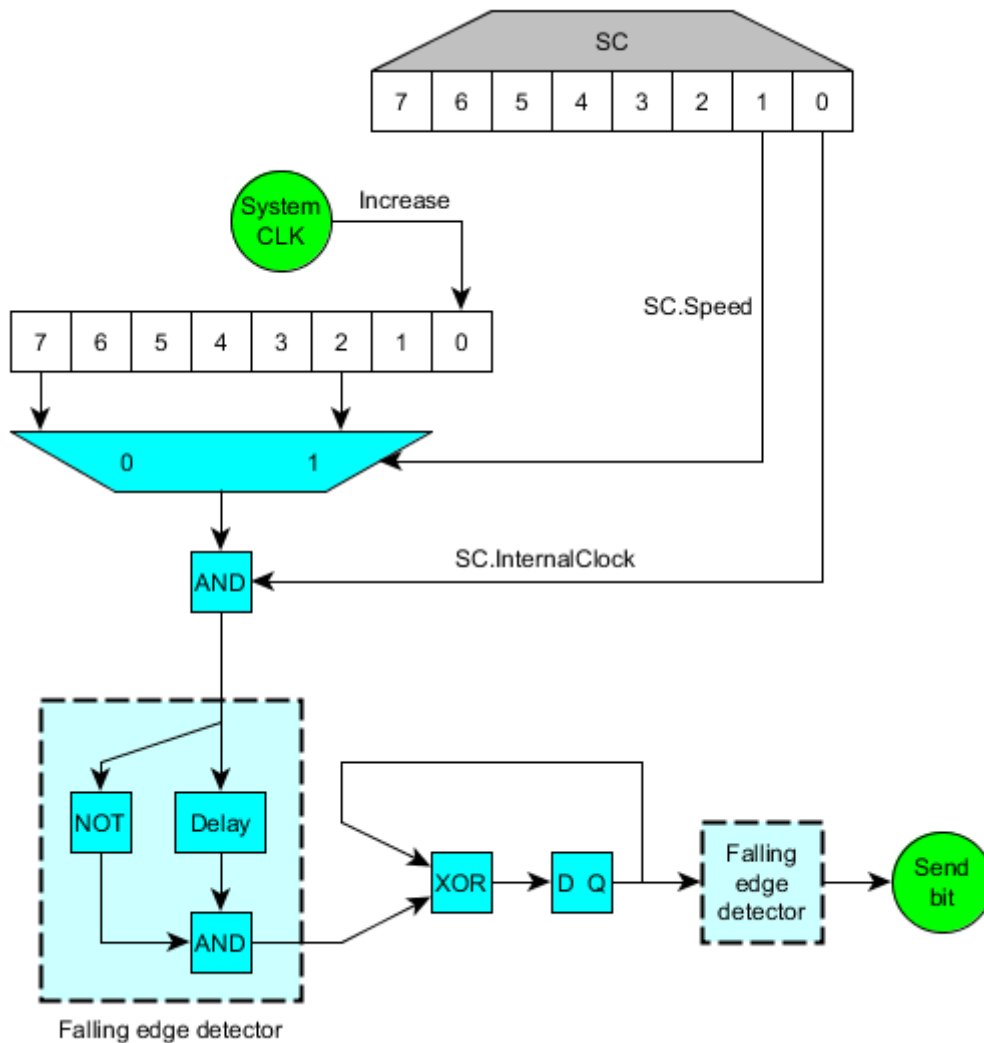
				[C]			
SYS	FD	FE	FF	00	01	02	03
TIMA	45	45	45	46	46	46	46
TMA	23	23	23	23	23	23	23
IF	E0	E0	E0	E0	E0	E0	E0

- During the strange cycle [A] you can prevent the IF flag from being set and prevent the TIMA from being reloaded from TMA by writing a value to TIMA. That new value will be the one that stays in the TIMA register after the instruction. Writing to DIV, TAC or other registers won't prevent the IF flag from being set or TIMA from being reloaded.
- If you write to TIMA during the cycle that TMA is being loaded to it [B], the write will be ignored and TMA value will be written to TIMA instead.
- If TMA is written the same cycle it is loaded to TIMA [B], TIMA is also loaded with that value.
- This is a guessed schematic to explain the priorities with registers TIMA and TMA:

TMA is a latch. As soon as it is written, the output shows that value. That explains that when TMA is written and TIMA is being incremented, the value written to TMA is also written to TIMA. It doesn't affect the IF flag, though.

## 6. Serial

The serial hardware works with an internal timer that can't be reset by any means. This is a diagram of the control part of the serial:



TODO: check, document... everything that goes here...

SB is R/W

Unused bits in SC are 1

masks:

DMG/GBC in DMG mode: 7E

GBC: 7C

## 7. Joypad

### FF00h – P1 – Joypad (R/W)

The buttons of the Game Boy are arranged in the form of a 2×4 matrix. This matrix has 2 input lines (outputs from the CPU) and 4 output lines (inputs from the CPU).

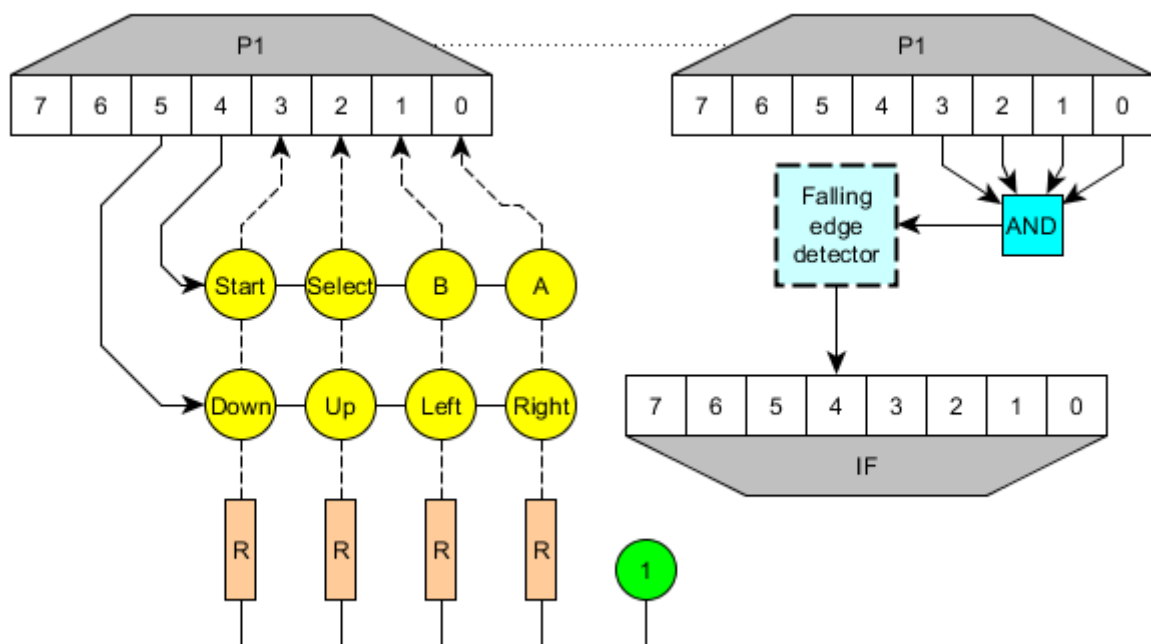
Most programs read from this port several times as a delay to allow the inputs to stabilize.

This register is also used to send command packets to the SNES in the SGB and SGB2 models.

Bit 5 - P15 Select Button Keys (0=Select)  
Bit 4 - P14 Select Direction Keys (0=Select)  
Bit 3 - P13 Input Down or Start (0=Pressed) (Read Only)  
Bit 2 - P12 Input Up or Select (0=Pressed) (Read Only)  
Bit 1 - P11 Input Left or Button B (0=Pressed) (Read Only)  
Bit 0 - P10 Input Right or Button A (0=Pressed) (Read Only)

Bits 6 and 7 always return 1.

The inputs to the CPU are pulled up by four resistors. They are only '0' when a CPU output is selected (set to '0') and a button is pressed. This is a schematic of the circuit:



### Reading the Joypad

This is a commonly used routine to read the joypad:

```
ld  a,$10
ld  [rP1],a      ; select P14
ld  a,[rP1]
ld  a,[rP1]      ; Wait a few cycles.
```



```

cpl                ; Complement A.
and  a,$0F         ; Get only first 4 bits.
swap a            ; Swap it.
ld   b,a
ld   a,$20
ld   [rP1],a      ; Select P15.
ld   a,[rP1]
ld   a,[rP1]
ld   a,[rP1]
ld   a,[rP1]
ld   a,[rP1]
ld   a,[rP1]
ld   a,[rP1]      ; Wait a few MORE cycles.
cpl
and  a,$0F
or   a,b          ; Put A and B together.
ld   [JOYPAD_STATE],a ; Save joypad state.
ld   a,$30        ; Deselect P14 and P15.
ld   [rP1],a

```

The resulting byte is formed like this:

```

Bit 7 - Start  (1=Pressed)
Bit 6 - Select (1=Pressed)
Bit 5 - B      (1=Pressed)
Bit 4 - A      (1=Pressed)
Bit 3 - Down   (1=Pressed)
Bit 2 - Up     (1=Pressed)
Bit 1 - Left   (1=Pressed)
Bit 0 - Right  (1=Pressed)

```

## Joypad Interrupt

This interrupt is triggered when there is a transition from '1' to '0' in one of the P1 input lines. For that to happen, one or two of the output lines have to be selected (with a '0' in that line). It can only detect a key press if there are no other buttons pressed.

This interrupt is present in all Game Boy models. GBA SP seems to be the only model not affected by key bouncing because of the different buttons used in it, or maybe because it uses low-pass filters.

## Software-Triggered Joypad Interrupt

This interrupt can be triggered by software. The interrupt itself is driven by an internal signal. When that signal goes from '1' to '0', the IF flag is set to '1'. The signal is obtained by performing a logical AND of the four input bits of P1.

Normally, the program would set the output bits to '0'. The joypad inputs are pulled up, so until the user presses, the signals are '1' and the internal signal is '1'. When the user presses, at least one of the inputs go to '0' and the internal signal is set to '0', so the IF flag is set.

If the user is pressing a button but both output lines are set to '1', the input lines are set to '1' too. If the program sets the output lines to '0', one of the input bits will go to '0', setting the IF flag too.

## 8. Video controller

### The LCD

The Game Boy LCD is 160×144 pixels. It can display 4 gray shades in DMG/MGB/SGB and 15-bit depth colors in CGB/AGB/AGS. It can show a background and a window (another background over the first one), and up to 40 sprites (10 per line) of 8×8 or 8×16 (everyone the same size). The DMG has 1 palette of 4 gray shades for backgrounds and 2 palettes of 3 gray shades for sprites (color 0 is transparent). The GBC has 8 palettes of 4 colors for backgrounds and 8 palettes of 3 colors for sprites (color 0 is transparent). A vertical refresh happens every 70224 clocks (140448 in GBC double speed mode): 59,7275 Hz

LCD timings are a bit different depending on the hardware. DMG and MGB (and SGB?) behave the same way and CGB, AGB, AGS in another way (even in DMG compatibility mode).

### FF40h – LCDC – LCD Control (R/W)

This register is used to enable or disable various elements (sprites, background, the LCD itself) and to configure them.

Bit 7 - LCD Power	(0=Off, 1=On)
Bit 6 - Window Tile Map	(0=9800h-9BFFh, 1=9C00h-9FFFh)
Bit 5 - Window Enable	(0=Disabled, 1=Enabled)
Bit 4 - BG & Window Tileset	(0=8800h-97FFh, 1=8000h-8FFFh)
Bit 3 - BG Tile Map	(0=9800h-9BFFh, 1=9C00h-9FFFh)
Bit 2 - Sprite Size	(0=8×8, 1=8×16)
Bit 1 - Sprites Enabled	(0=Disabled, 1=Enabled)
Bit 0 - BG Enabled (in DMG)	(0=Disabled, 1=Enabled)

Turning off the LCD (by writing a 0 to bit 7) should only be done during V-Blank. Doing it outside V-Blank period may damage the hardware.

Bit 0 of this register has different meanings depending on the Game Boy model.

- DMG/MGB/SGB: BG Enable. Window and sprites are unaffected by this bit.

- CGB/AGB/AGS:

- CGB mode: BG and Window Master Priority. If this bit is '0', sprites are always on top of the background and the window, regardless of BG attribute map and sprite priority flags.

- DMG compatibility mode: BG and Window Enable. If this bit is '0', both window and background are disabled (bit 5 is ignored). This is a possible compatibility problem.

A disabled background is white (color 0?).

TODO: Test everything.

## FF44h – LY – LCD Current Scanline (R)

This register indicates the current horizontal line that is being drawn. It can take any value from 0 to 153. When the LCD is off this register is fixed at 00h.

## FF45h – LYC – LY Compare (R/W)

This value is used to trigger an interrupt when LY has the same value as LYC. Check STAT register documentation below for more information.

## FF41h – STAT – LCD Status (R/W)

This register is used to check the status of the LCD and to configure the LCD interrupt. For more information about LCD interrupt read below.

Bit 6	- LY=LYC Check Enable	(1=Enable) (Read/Write)
Bit 5	- Mode 2 OAM Check Enable	(1=Enable) (Read/Write)
Bit 4	- Mode 1 V-Blank Check Enable	(1=Enable) (Read/Write)
Bit 3	- Mode 0 H-Blank Check Enable	(1=Enable) (Read/Write)
Bit 2	- LY=LYC Comparison Signal	(1:LYC=LY) (Read Only)
Bits 1-0	- Screen Mode	(Mode 0-3) (Read Only)
	0: H-Blank	
	1: V-Blank	
	2: Searching OAM-RAM	
	3: Transferring Data to LCD Driver.	

Bit 7 is unused and always returns '1'. Bits 0-2 return '0' when the LCD is off.

During mode 0 and mode 1 the CPU can access both VRAM and OAM. During mode 2 the CPU can only access VRAM, not OAM. During mode 3 OAM and VRAM can't be accessed. In GBC mode the CPU can't access Palette RAM (FF69h and FF6Bh) during mode 3.

A scanline normally takes 456 clocks (912 clocks in double speed mode) to complete. A scanline starts in mode 2, then goes to mode 3 and, when the LCD controller has finished drawing the line (the timings depend on lots of things) it goes to mode 0. During lines 144-153 the LCD controller is in mode 1. Line 153 takes only a few clocks to complete (the exact timings are below). The rest of the clocks of line 153 are spent in line 0 in mode 1!

Bit 2 is set to '1' even if Bit 6 is '0'. The timings are a bit strange, they are described below.

- VBL IRQ is not delayed, it's executed the same clock as PPU enters VBL screen mode.??????

-What happens when turning it on/off?

## VBL Interrupt

Vertical Blank interrupt is triggered when the LCD controller enters the VBL screen mode (mode 1, LY=144). This happens once per frame, so this interrupt is triggered 59.7 times per second. During

this period the VRAM and OAM can be accessed freely, so it's the best time to update graphics (for example, use the OAM DMA to update sprites for next frame, or update tiles to make animations).

This period lasts 4560 clocks in normal speed mode and 9120 clocks in double speed mode.

## STAT Interrupt

This interrupt can be configured with register STAT.

The STAT IRQ is triggered by an internal signal.

This signal is set to 1 if:

( (LY = LYC) AND (STAT.ENABLE\_LYC\_COMPARE = 1) ) OR

( (ScreenMode = 0) AND (STAT.ENABLE\_HBL = 1) ) OR

( (ScreenMode = 2) AND (STAT.ENABLE\_OAM = 1) ) OR

( (ScreenMode = 1) AND (STAT.ENABLE\_VBL || STAT.ENABLE\_OAM) ) -> Not only

VBL!??

-If LCD is off, the signal is 0.

-The STAT IRQ is triggered when this signal goes from 0 to 1 (rising edge). This explains some cases when the IRQ is not triggered if 2 or more events are enabled. For example, when going from HBL to OAM mode, if STAT.ENABLE\_HBL and STAT.ENABLE\_OAM are enabled, the signal will stay 1 in the transition, so no IRQ will be triggered.

-It seems that this interrupt needs less time to execute in DMG than in CGB?

-DMG bug?

## LY, LYC, STAT and IF Timings. LY=LYC interrupt.

The next tables will show scanline timings (accurate to 4 clocks) but they don't show the change from mode 3 to mode 0 because it depends on the number of sprites drawn, this will be explained later. Mode 2 needs the same number of clocks to complete regardless of sprite size and the sprite enable bits of LCDC register and the number of sprites in the line. DMG has the same timings as MGB. CGB has the same timings as AGB and AGS.

Each scanline takes 456 clocks in normal speed mode and 912 clocks in double speed mode but there are some strange things regarding the LY register and the LY=LYC comparison that are explained below. The "LY to compare LYC" means that if LYC is equal to the value of the table, Bit 2 of STAT register is set to '1', if not it is set to '0'. Its timings are a bit different from the real LY value. A '-' in this field means that it seems that Bit 2 is always '0' in this cycle, regardless of the value of LYC.

In the IF Flag field, a '1' means that this clock the IF flag will read '1' if the LYC value is the one that corresponds to LY (and STAT Bit 6 is set). Of course, the value won't be changed to '0' unless LCD interrupt is handled or the IF register is written manually, a '0' only means that during that cycle the flag doesn't change to '1' (if STAT register isn't modified).

## Timings in DMG

### Line 0

Clocks	0	4	8	12	...	76	80	84	...	448	452
LY Register	0	0	0	0	...	0	0	0	...	0	0
STAT Mode	0	2	2	2	...	2	2	3	...	0	0
LY to compare LYC	0	0	0	0	...	0	0	0	...	0	0
IF Flag (LY=LYC)	0	0	0	0	...	0	0	0	...	0	0

### Lines 1-143

Clocks	0	4	8	12	...	76	80	84	...	448	452
LY Register	13	13	13	13	...	13	13	13	...	13	13
STAT Mode	0	2	2	2	...	2	2	3	...	0	0
LY to compare LYC	-	13	13	13	...	13	13	13	...	13	13
IF Flag (LY=LYC)	0	1	0	0	...	0	0	0	...	0	0

### Line 144

Clocks	0	4	8	12	...				...	448	452
LY Register	144	144	144	144	...				...	144	144
STAT Mode	0	1	1	1	...				...	1	1
LY to compare LYC	-	144	144	144	...				...	144	144
IF Flag (LY=LYC)	0	1	0	0	...				...	0	0

### Lines 145-152

Clocks	0	4	8	12	...				...	448	452
LY Register	146	146	146	146	...				...	146	146
STAT Mode	1	1	1	1	...				...	1	1
LY to compare LYC	-	146	146	146	...				...	146	146
IF Flag (LY=LYC)	0	1	0	0	...				...	0	0

### Line 153

Clocks	0	4	8	12	16	...			...	448	452
LY Register	153	0	0	0	0	...			...	0	0
STAT Mode	1	1	1	1	1	...			...	1	1
LY to compare LYC	-	153	-	0	0	...			...	0	0
IF Flag (LY=LYC)	0	1	0	1	0	...			...	0	0

## Timings in CGB in DMG mode

### Line 0

Clocks	0	4	8	12	...	76	80	84	...	448	452
LY Register	0	0	0	0	...	0	0	0	...	0	0
STAT Mode	1	2	2	2	...	2	2	3	...	0	0
LY to compare LYC	0	0	0	0	...	0	0	0	...	0	0
IF Flag (LY=LYC)	0	0	0	0	...	0	0	0	...	0	0

### Lines 1-143

Clocks	0	4	8	12	...	76	80	84	...	448	452
LY Register	13	13	13	13	...	13	13	13	...	13	13
STAT Mode	0	2	2	2	...	2	2	3	...	0	0
LY to compare LYC	12	13	13	13	...	13	13	13	...	13	13
IF Flag (LY=LYC)	0	1	0	0	...	0	0	0	...	0	0

### Line 144

Clocks	0	4	8	12	...				...	448	452
LY Register	144	144	144	144	...				...	144	144
STAT Mode	0	1	1	1	...				...	1	1
LY to compare LYC	143	144	144	144	...				...	144	144
IF Flag (LY=LYC)	0	1	0	0	...				...	0	0

### Lines 145-152

Clocks	0	4	8	12	...				...	448	452
LY Register	146	146	146	146	...				...	146	146
STAT Mode	1	1	1	1	...				...	1	1
LY to compare LYC	145	146	146	146	...				...	146	146
IF Flag (LY=LYC)	0	1	0	0	...				...	0	0

### Line 153

Clocks	0	4	8	12	16	...			...	448	452
LY Register	153	153	0	0	0	...			...	0	0
STAT Mode	1	1	1	1	1	...			...	1	1
LY to compare LYC	152	153	153	0	0	...			...	0	0

IF Flag (LY=LYC)	0	1	0	1	0	...			...	0	0
------------------	---	---	---	---	---	-----	--	--	-----	---	---

## Timings in CGB in CGB mode (single speed)

### Line 0

Clocks	0	4	8	12	...	76	80	84	...	448	452
LY Register	0	0	0	0	...	0	0	0	...	0	(*)
STAT Mode	2	2	2	2	...	2	3	3	...	0	0
LY to compare LYC	0	0	0	0	...	0	0	0	...	0	0
IF Flag (LY=LYC)	0	0	0	0	...	0	0	0	...	0	0

### Lines 1-143

Clocks	0	4	8	12	...	76	80	84	...	448	452
LY Register	13	13	13	13	...	13	13	13	...	13	(*)
STAT Mode	2	2	2	2	...	2	3	3	...	0	0
LY to compare LYC	13	13	13	13	...	13	13	13	...	13	13
IF Flag (LY=LYC)	1	0	0	0	...	0	0	0	...	0	0

### Line 144

Clocks	0	4	8	12	...				...	448	452
LY Register	144	144	144	144	...				...	144	(*)
STAT Mode	1	1	1	1	...				...	1	1
LY to compare LYC	144	144	144	144	...				...	144	144
IF Flag (LY=LYC)	1	0	0	0	...				...	0	0

### Lines 145-152

Clocks	0	4	8	12	...				...	448	452
LY Register	146	146	146	146	...				...	146	(*)
STAT Mode	1	1	1	1	...				...	1	1
LY to compare LYC	146	146	146	146	...				...	146	146
IF Flag (LY=LYC)	1	0	0	0	...				...	0	0

### Line 153

Clocks	0	4	8	12	...				...	448	452
LY Register	153	0	0	0	...				...	0	0

STAT Mode	1	1	1	1	...				...	1	1
LY to compare LYC	153	153	0	0	...				...	0	0
IF Flag (LY=LYC)	1	0	1	0	...				...	0	0

The value of (\*) depends on the current scanline (and it's consistent between GBC models). Values for lines 0-152 are:

LY	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh
(*)	00h	00h	02h	00h	04h	04h	06h	00h	08h	08h	0Ah	08h	0Ch	0Ch	0Eh	00h
LY	10h	11h	12h	13h	14h	15h	16h	17h	18h	19h	1Ah	1Bh	1Ch	1Dh	1Eh	1Fh
(*)	10h	10h	12h	10h	14h	14h	16h	10h	18h	18h	1Ah	18h	1Ch	1Ch	1Eh	00h
LY	20h	21h	22h	23h	24h	25h	26h	27h	28h	29h	2Ah	2Bh	2Ch	2Dh	2Eh	2Fh
(*)	20h	20h	22h	20h	24h	24h	26h	20h	28h	28h	2Ah	28h	2Ch	2Ch	2Eh	20h
LY	30h	31h	32h	33h	34h	35h	36h	37h	38h	39h	3Ah	3Bh	3Ch	3Dh	3Eh	3Fh
(*)	30h	30h	32h	30h	34h	34h	36h	30h	38h	38h	3Ah	38h	3Ch	3Ch	3Eh	00h
LY	40h	41h	42h	43h	44h	45h	46h	47h	48h	49h	4Ah	4Bh	4Ch	4Dh	4Eh	4Fh
(*)	40h	40h	42h	40h	44h	44h	46h	40h	48h	48h	4Ah	48h	4Ch	4Ch	4Eh	40h
LY	50h	51h	52h	53h	54h	55h	56h	57h	58h	59h	5Ah	5Bh	5Ch	5Dh	5Eh	5Fh
(*)	50h	50h	52h	50h	54h	54h	56h	50h	58h	58h	5Ah	58h	5Ch	5Ch	5Eh	40h
LY	60h	61h	62h	63h	64h	65h	66h	67h	68h	69h	6Ah	6Bh	6Ch	6Dh	6Eh	6Fh
(*)	60h	60h	62h	60h	64h	64h	66h	60h	68h	68h	6Ah	68h	6Ch	6Ch	6Eh	60h
LY	70h	71h	72h	73h	74h	75h	76h	77h	78h	79h	7Ah	7Bh	7Ch	7Dh	7Eh	7Fh
(*)	70h	70h	72h	70h	74h	74h	76h	70h	78h	78h	7Ah	78h	7Ch	7Ch	7Eh	00h
LY	80h	81h	82h	83h	84h	85h	86h	87h	88h	89h	8Ah	8Bh	8Ch	8Dh	8Eh	8Fh
(*)	80h	80h	82h	80h	84h	84h	86h	80h	88h	88h	8Ah	88h	8Ch	8Ch	8Eh	80h
LY	90h	91h	92h	93h	94h	95h	96h	97h	98h							
(*)	90h	90h	92h	90h	94h	94h	96h	90h	98h							

Line 153 (99h) is shorter than the rest, it works different, so it doesn't follow the pattern of that table. Take a look at the timings for line 153 below.

A pattern is repeated every 8 lines (except for lines in the form NFh, that follow the same pattern but it shows in the upper nibble). The following C code can be used to determine the correct value for (\*) depending on the current scanline:

```

unsigned char get_value_from_ly(unsigned char ly) // ly from 00h to 98h!
{
    const unsigned char pattern[9] = {0, 0, 2, 0, 4, 4, 6, 0, // Pattern
                                       8}; // To make it work with NFh lines
    if( (ly & 0x0F) == 0x0F )
        return ( pattern[(ly >> 4) & 0x0F] << 4 );
    else
        return ( pattern[ly & 7] | (ly & 0xF8) );
}

```



## Timings in CGB in CGB mode (double speed)

### Line 0

Clocks	0	4	8	12	...	160	164	168	...	904	908
LY Register	0	0	0	0	...	0	0	0	...	0	0
STAT Mode	1	2	2	2	...	2	3	3	...	0	0
LY to compare LYC	0	0	0	0	...	0	0	0	...	0	0
IF Flag (LY=LYC)	0	0	0	0	...	0	0	0	...	0	0

### Lines 1-143

Clocks	0	4	8	12	...	160	164	168	...	904	908
LY Register	13	13	13	13	...	13	13	13	...	13	13
STAT Mode	0	2	2	2	...	2	3	3	...	0	0
LY to compare LYC	12	13	13	13	...	13	13	13	...	13	13
IF Flag (LY=LYC)	0	1	0	0	...	0	0	0	...	0	0

### Line 144

Clocks	0	4	8	12	...				...	904	908
LY Register	144	144	144	144	...				...	144	144
STAT Mode	0	1	1	1	...				...	1	1
LY to compare LYC	143	144	144	144	...				...	144	144
IF Flag (LY=LYC)	0	1	0	0	...				...	0	0

### Lines 144-152

Clocks	0	4	8	12	...				...	904	908
LY Register	146	146	146	146	...				...	146	146
STAT Mode	1	1	1	1	...				...	1	1
LY to compare LYC	145	146	146	146	...				...	146	146
IF Flag (LY=LYC)	0	1	0	0	...				...	0	0

### Line 153

Clocks	0	4	8	12	14	16	20	24	...	904	908
LY Register	153	153	153	0	0	0	0	0	...	0	0
STAT Mode	1	1	1	1	1	1	1	1	...	1	1
LY to compare LYC	152	153	153	153	153	0	0	0	...	0	0
IF Flag (LY=LYC)	0	1	0	0	0	1	0	0	...	0	0

Timings when writing to LYC. How many cycles does it take to update BIT(2)? Can it be measured using software only?

Does STAT IF flag change to 1 automatically or does it have a delay in some cases?

Can the interrupt be disabled when writing to STAT or other registers just when changing the value of IF?

Does LYC >153 trigger stat interrupt?

In GBC single speed, can the (\*) cycle trigger LYC interrupt with the strange values?

DMG interrupt handling times are different than CGB. - check again!

**LY, STAT and IF Timings. VBL and STAT (mode 1) interrupts.**

TODO THIS

**LY, STAT and IF Timings. STAT interrupt (mode 2).**

TODO THIS

**LY, STAT and IF Timings. STAT interrupt (mode 0).**

TODO THIS

MODE 3 TIMINGS DEPENDING ON THE NUMBER OF SPRITES!!!

## **FF42h – SCY – BG Scroll Y (R/W)**

Specifies the Y coordinate of the background at the upper left pixel of the LCD.

## **FF43h – SCX – BG Scroll X (R/W)**

Specifies the X coordinate of the background at the upper left pixel of the LCD.

The background map is 32×32 tiles (256×256 pixels). Any value can be written to SCX and SCY (0-255). The background will wrap when exceeding the lower or right border.

## **FF4Ah – WY – Window Y Position (R/W)**

Specifies the Y coordinate of the upper left corner of the window.

## **FF4Bh – WX – Window X Position (R/W)**

Specifies the X coordinate (minus 7) of the upper left corner of the window.

The window is an alternate background that can be displayed over the normal background. This window is drawn by resetting the LCD background state machine and changing to the window map. This results in only being able to show the upper left part of the window map (and no wrapping, logically). Priorities between window and sprites are shared with normal background.

The behaviour of this registers is a bit strange

## Window Display Depending on WX and WY

The window is visible (if enabled) when WX=7-166, WY=0-143. A position of WX=7, WY=0 locates the window at upper left corner of the screen, covering normal background. Other values for WX will give strange results:

TODO

## Accessing Unavailable VRAM and OAM

When the PPU is accessing the video memory the CPU can't access it. The behaviour of OAM and VRAM is different depending on the screen mode (2 or 3). OAM can't be accessed in modes 2 or 3, and VRAM can't be accessed in mode 3.

### Reading from OAM in Mode 2

Reads will return FFh regardless of the hardware. In GBC it doesn't matter if GBC functions are enabled or disabled. It doesn't matter if sprites are enabled or disabled in LCDC.

### Reading from OAM in Mode 3

...

...

...

### Reading from VRAM in Mode 3

Todo...

### Reading from GBC Palette RAM in Mode 2

Todo...

### Reading from GBC Palette RAM in Mode 3

Todo...

## 9. DMA

The DMA is a special circuit designed to copy very fast. There are two DMAs in the Game Boy. The first one is the OAM DMA, available in all Game Boy models. The second one is the GDMA/HDMA, available only in CGB/AGB/AGS in GBC mode.

### FF46h – DMA – OAM DMA Transfer (R/W)

This register starts the OAM DMA transfer. The written value specifies the upper byte of the source address (XX00h). The destination address is always OAM. If this register is written when a copy is active, it will be cancelled and a new copy will start. This copy needs  $160 \times 4 + 4$  clocks to complete in both double speed and single speeds modes. The copy starts after the 4 setup clocks, and a new byte is copied every 4 clocks.

**TODO: Test if the first 4 clocks are Read(0), and then every 4 clocks Read(n+1) and Write(n) are performed at the same time. It would make sense.**

Any written value (00h – FFh) will trigger the copy, but the copied data and the disabled memory areas depend on the source area and the Game Boy model.

During this copy interrupts can still be triggered and the CPU can jump to interrupt vectors. This shouldn't be a problem with VBL and LCD interrupts if DMA is used during V-Blank only, but the timer, serial and joypad interrupts can still be a problem.

Programs usually have a small routine at HRAM that is called when the program needs to perform a DMA copy. Most programs execute this routine during V-Blank as H-Blank is too short to perform a full DMA copy.

```
ld  [rDMA],a  ;Start DMA transfer. The A register has the source address.
ld  a,$28     ;Delay...
.wait:
dec a
jr  nz,.wait
ret
```

- If HALT or STOP mode are entered during a DMA copy, the copy will still complete correctly. The same happens with GBC performing a speed switch. The copy is paused during HALT and STOP modes, and it's also paused during the speed switch.

**- Start DMA while HDMA? GDMA/HDMA while DMA?**

## **FF51h – HDMA1 – GBC Mode – HDMA Source, High (W)**

Specifies the higher byte of the source address. Always returns FFh when read.

## **FF52h – HDMA2 – GBC Mode – HDMA Source, Low (W)**

Specifies the lower byte of the source address. Lower 4 bits are ignored, addresses are always aligned to 10h (16 bytes). Always returns FFh when read.

## **FF53h – HDMA3 – GBC Mode – HDMA Destination, High (W)**

Specifies the higher byte of the destination address. Destination is always in VRAM (8000h – 9FFFh), the 3 upper bits are ignored. Always returns FFh when read.

## **FF54h – HDMA4 – GBC Mode – HDMA Destination, Low (W)**

Specifies the lower byte of the destination address. Lower 4 bits are ignored, addresses are always aligned to 10h (16 bytes). Always returns FFh when read.

## **FF55h – HDMA5 – GBC Mode – HDMA Length/Mode/Start (R/W)**

This register specifies the length and mode of the transfer. It starts the copy when it is written. Returns FFh in DMG and GBC in DMG mode.

Bit 7 - Transfer mode (0=GDMA, 1=HDMA)  
Bits 6-0 - Blocks (Size = (Blocks+1)×16 bytes)

When a copy is finished, if another copy is started without changing the source/destination addresses, it will continue from the last addresses of the previous copy. For example, writing 01h twice to this register will copy the same data as when writing 03h.

A GDMA/HDMA copy cannot be interrupted by interrupts. The CPU will handle them after finishing copying. This is not a problem for HDMA (probably), but it may be a problem if the program is copying a large memory block and an important interrupt is triggered.

This DMA shouldn't be used in old cartridges, they aren't guaranteed to support reading latencies fast enough to allow using this DMA. It doesn't seem to matter what MBC the game uses. Pokémon Crystal (MBC3) uses HDMA, while most GBC games use MBC5.

## **GDMA – General Purpose DMA**

All the data is transferred at once, and the CPU is halted until the copy is finished. This will try to copy even if VRAM is being used by the LCD controller (so the writes will fail). This should be used during V-Blank or when the screen is off (or for small blocks of data during H-Blank).

Timings:

- Single speed mode:  $(4 + 32 \times \text{blocks})$  clocks
- Double speed mode:  $(4 + 64 \times \text{blocks})$  clocks

The preparation time (4 clocks) is the same in single and double speed mode, but the actual transfer

needs the same time in both modes, so in double speed mode it needs twice the clocks than in single speed mode.

When the transfer is finished this register will return FFh.

## HDMA – H-Blank DMA

This mode will transfer one block (16 bytes) during each H-Blank. No data is transferred during V-Blank (LY = 143 – 153), but the transfer will continue at LY = 0. During the copy periods the CPU is stopped, the same as GDMA.

Source and destination addresses may be changed during the copy.

After writing a value to HDMA5 that starts the HDMA copy, the upper bit (that indicates HDMA mode when set to '1') will be cleared. Reading from HDMA5 register will return the remaining length (divided by 16, minus 1). A value of FFh indicates that the transfer has completed.

- If HDMA5 is written during a HDMA copy, the behaviour depends on the written bit 7.

- New bit 7 is 0: Stop copy. HDMA5 new value is ( 80h OR written\_value ).
- New bit 7 is 1: Restart copy. New size is the value of written\_value bits 0-6.

This means that HDMA can't switch to GDMA with only one write. It must be stopped first.

- If the CPU is in HALT or STOP modes, the HDMA copy won't happen. It also won't happen during a speed switch.

- If a HDMA transfer is started when the screen is off, one block is copied. The transfer is continued the first H-Blank period after switching the screen on.

- HDMA will only copy one block when the screen is off, right when starting the copy. It will continue after the screen is turned on. If the HDMA started when the screen was on, when the screen is switched off it will copy one block after the switch.

- If ROM/VRAM banks are changed the copy will continue with the new banks.

- When a HDMA transfer is started during HBL it will start right away. There is no problem of copying while VRAM is inaccessible, mode 2 is long enough to allow the HDMA copy the 16 bytes.

Timings (copy time per HBL):

- Single speed mode: (4 + 32) clocks
- Double speed mode: (4 + 64) clocks

## GDMA/HDMA Allowed Source Addresses

The copying circuit can't read from any memory. Source address can be 0000h-7FFFh (ROM), A000h-BFFFh (SRAM) or C000h-DFFFh (WRAM).

- Selecting an address in the range E000h-FFFFh will read addresses from A000h-BFFFh (SRAM).
- Selecting an address in the range 8000h-9FFFh (VRAM) will read incorrect data.

The incorrect data read when copying from VRAM depends on the mode (GDMA/HDMA) and the hardware.

- CGB/AGB: The DMA circuit copies two unknown bytes and the rest of the bytes are FFh.
- AGS: The same, but only one byte.

In GDMA the corrupted bytes are copied once for the whole copy. In HDMA the corrupted bytes are copied for each block.

TODO: In GDMA investigate the incorrect data pattern depending on the status of the LCD controller and the hardware: screen on/off, mode 0,1,2,3.

...

.....

## **10. Audio Processing Unit**



# 11. The Game Boy Cartridge

## The Cartridge Header

The area at 0100h-014Fh of the ROM is reserved for some special information.

### 0100h – 0103h – Start Vector

When the boot ROM is exited the game execution starts at 0100h. Usually, games have a “nop” here followed by a “jp 150h”.

### 0104h – 0133h – Nintendo Logo

This 48 bytes represent the Nintendo logo that is shown when the Game Boy is powered on (the ones that appear corrupted if the cartridge is not inserted correctly, for example). They must be set to specific values or the boot ROM won't jump to the game start vector (0100h). Some models check more values than others. The DMG and MGB check the 48 bytes, the SGB doesn't check any byte (the SNES does it). The GBC only checks the first 24 bytes.

The correct values are:

```
CE ED 66 66 CC 0D 00 0B 03 73 00 83 00 0C 00 0D
00 08 11 1F 88 89 00 0E DC CC 6E E6 DD DD D9 99
BB BB 67 63 6E 0E EC CC DD DC 99 9F BB B9 33 3E
```

### 0134h – 0143h – Game Title

Title of the game in upper case ASCII. The unused bytes after the title are filled with 00h. The last few bytes are used for other things in GBC games. This is described below.

### 013Fh – 0142h – Manufacturer Code

This is a 4 character uppercase manufacturer code.

### 0143h – GBC Flag

This flag is used to enable GBC functions in CGB/AGB/AGS. The two used values that will make the GBC enter GBC mode are C0h and 80h. Both have the same effect, but C0h means that the game is GBC only and 80h means that the game can also run on older monochrome Game Boy models.

Values with bit 7 set and bit 2 or bit 3 set will switch the GBC into a strange non-GBC-mode.

TODO: CHECK THIS USING THE GBC BOOT ROM. Any value with bit 7 set seems to have the same effect, but check bits 2 and 3.

### 0144h – 0145h – New Licensee Code

A two character ASCII code that indicates the company or publisher of the game. Only used in games released after the SGB, older games use the value at 014Bh.

## 0146h – SGB Flag

Specifies if the game supports SGB functions or not. This is checked by the SNES, not the SGB boot ROM. A value of 00h indicates that there are no SGB functions (DMG or CGB only game). A value of 03h indicates that the game supports SGB functions. Any other value will have the same effect as a 00h.

## 0147h – Cartridge Type

Specifies the Memory Bank Controller (MBC) if any, and if there is additional hardware in the cartridge.

00h - ROM Only	14h - Unused
01h - MBC1	15h - Unused
02h - MBC1 + RAM	16h - Unused
03h - MBC1 + RAM + Battery	17h - Unused
04h - Unused	18h - Unused
05h - MBC2	19h - MBC5
06h - MBC2 + RAM + Battery	1Ah - MBC5 + RAM
07h - Unused	1Bh - MBC5 + RAM + Battery
08h - ROM + RAM	1Ch - MBC5 + Rumble
09h - ROM + RAM + Battery	1Dh - MBC5 + RAM + Rumble
0Ah - Unused	1Eh - MBC5 + RAM + Battery + Rumble
0Bh - MMM01	1Fh - Unused
0Ch - MMM01 + RAM	20h - MBC6 + RAM + Battery
0Dh - MMM01 + RAM + Battery	21h - Unused
0Eh - Unused	22h - MBC7 + RAM + Bat. + Accelerometer
0Fh - MBC3 + Timer + Battery	... - Unused
10h - MBC3 + RAM + Timer + Battery	FCh - POCKET CAMERA
11h - MBC3	FDh - BANDAI TAMA5
12h - MBC3 + RAM	FEh - HuC3
13h - MBC3 + RAM + Battery	FFh - HuC1 + RAM + Battery

## 0148h – ROM Size

Specifies the ROM size, calculated as “32KB << NN”. The biggest known GBC ROM is 8MB. Valid values are:

00h - 32KB - 2 banks (No MBC needed)	05h - 1MB - 64 banks
01h - 64KB - 4 banks	06h - 2MB - 128 banks
02h - 128KB - 8 banks	07h - 4MB - 256 banks
03h - 256KB - 16 banks	08h - 8MB - 512 banks
04h - 512KB - 32 banks	

## 0149h – RAM Size

Specifies the RAM size (if any):

00h - None  
01h - 2KB  
02h - 8KB - 1 bank  
03h - 32KB - 4 banks of 8KB  
04h - 128KB - 16 banks of 8KB  
05h - 64KB - 8 banks of 8 KB. Used by "Pokémon Crystal (Japan)".

In MBC2 cartridges this value is 00h, but the MBC chip has a RAM of  $512 \times 4$  bit.

### **014Ah – Destination Code**

A value of 00h means that the game is supposed to be sold in Japan, a 01h means anywhere else.

### **014Bh – Old Licensee Code**

Specifies the company or publisher of the game. A value of 33h indicates that the New Licensee Code at 0144h-0145h is used instead. SGB functions require that this byte is 33h!

### **014Ch – ROM Version**

Some games have more than one version, this byte indicates that. This value is usually 00h as most games only have one version.

### **014Dh – Header Checksum**

Contains a checksum of the cartridge header bytes 0134h-014Ch. It is calculated like this:

```
unsigned char sum = 0;
int i = 0x0134;
while(i <= 0x014C)
    sum = sum - Memory(i++) - 1;
```

The game won't run if this value is incorrect!

### **014Eh – 014Fh – Global Checksum**

This is a 16 bit checksum (upper byte in 014Eh) of the whole ROM. This is calculated adding all the values of the cartridge (except this two bytes). This is not verified in the Game Boy.

## **Memory Bank Controllers**

MBCs are chips used to avoid the limitation of the 16 bit address bus of the Game Boy. These chips are in the cartridge. They are used to select the ROM and RAM bank mapped to the Game Boy, and they are also used to enable or disable the RAM (if any) and to control the external hardware included in the cartridge (if any). The used MBC is specified in byte 0147h of the cartridge header.

Addresses at 0000h-7FFFh and A000h-BFFF can be used to read from cartridge hardware and to write to it. ROM addresses are used to switch ROM and RAM banks and enable or disable RAM, and RAM addresses are used to control extra hardware.

Disabled RAM will ignore any writes, and it will return FFh when read. If there is no RAM, any

reads from this area will also return FFh. RAM should be disabled when it's not being used to protect the data. RAM is disabled on reset.

If there is no ROM at 0000h-7FFFh (for example, by turning the Game Boy on without an inserted cartridge) this area will return FFh when read.

## **None (32KB ROM)**

This games don't need a MBC. All the ROM can be mapped to 0000h-7FFFh directly. A RAM chip up to 8KB may be connected to A000h-BFFFh, but a tiny circuit would be required to enable and disable it.

## **MBC1 (2MB ROM. 32KB RAM. DMG, SGB)**

This is the first MBC chip, used in most DMG and SGB games. This cartridge has a special register that selects the upper bits of the ROM bank number or the RAM bank number, but can't specify both at the same time.

### **0000h-3FFFh – ROM Bank 0 (Read)**

This area contains the first 16KB of the ROM.

### **4000h-7FFFh – Switchable ROM bank (Read)**

This area may contain any 16KB ROM bank. Bank numbers 00h, 20h, 40h, 60h cannot be used and the following bank (01h, 21h, 41h, 61h) will be selected instead!

### **A000h-BFFFh – Switchable RAM bank (Read/Write)**

This area is used to read and write from external RAM (if any). External RAM is often battery buffered so its values are stored even if the Game Boy is turned off or the cartridge is removed.

### **0000h-1FFFh – RAM Enable (Write)**

Writing to this address range any value with 0Ah in the lower 4 bits enables RAM, and any other value disables it. Usually, 00h is used to disable RAM and 0Ah is used to enable it.

### **2000h-3FFFh – ROM Bank (Write)**

This address range is used to select the lower 5 bits of ROM bank number (01h-1Fh). Writing 00h will be translated to 01h. This will make banks 00h, 20h, 40h and 60h impossible to map, and banks 01h, 21h, 41h and 61h will be selected instead.

### **4000h-5FFFh – RAM Bank/Upper bits of ROM Bank (Write)**

This is a 2 bit register that can select RAM banks 0 – 3 or can specify the upper 2 bits of ROM bank, depending on the cartridge mode (see below).

### **6000h-7FFFh – ROM/RAM Mode (Write)**

This is a 1 bit register that selects if the two bits in register at 4000h-5FFFh are used to select the two upper bits of the ROM bank number or the RAM bank number.

Mode 0 will enable ROM Banking Mode (enabled by default). Only RAM bank 0 can be accessed in this mode, even if the mapped bank before the mode change wasn't bank 0.

Mode 1 will enable RAM Banking Mode, and only ROM banks 01h-1Fh will be able to be accessed. If other ROM bank is selected, ROM bank will be changed to the corresponding in 01h-1Fh by clearing the upper 2 bits.

## **MBC2 (256KB ROM. 512 × 4 bits RAM. DMG, SGB)**

This MBC can only support up to 16 ROM banks and has a RAM circuit inside the MBC2 chip itself. The MBC chip has only 4 data lines, that's the reason only 16 ROM banks can be selected and only 4 bits can be read/written at a time from/to RAM.

### **0000h-3FFFh – ROM Bank 0 (Read)**

This area contains the first 16KB of the ROM.

### **4000h-7FFFh – Switchable ROM bank (Read)**

This area may contain any 16KB ROM bank except for bank 0.

### **A000h-A1FFh – RAM (Read/Write)**

This area is used to read and write from MBC2 RAM. Only the 4 lower bits are used, the upper bits should be ignored when reading and will be ignored when writing. They probably return '1' when read.

### **0000h-1FFFh – RAM Enable (Write)**

Writing to this address range any value with 0Ah in the lower 4 bits enables RAM, and any other value disables it. Usually, 00h is used to disable RAM and 0Ah is used to enable it.

The least significant bit of the upper address byte must be '0' to enable/disable cart RAM. For example, the following addresses can be used to enable/disable cart RAM: 0000h-00FFh, 0200h-02FFh, 0400h-04FFh, ..., 1E00h-1EFFh. The suggested address range to use for MBC2 ram enable/disable is 0000h-00FFh.

### **2000h-3FFFh – ROM Bank (Write)**

The lower 4 bits written here will specify the ROM bank mapped to 4000h-7FFFh. The least significant bit of the upper address byte must be '1' to select a ROM bank. For example, the following addresses can be used to select a ROM bank: 2100h-21FFh, 2300h-23FFh, 2500h-25FFh, ..., 3F00h-3FFFh. The suggested address range to use for MBC2 ROM bank selection is 2100h-21FFh.

## **MBC3 (2MB ROM. 64KB RAM. RTC. DMG, SGB, CGB)**

This MBC can include a Real Time Clock. It uses an external 32.768 kHz Quartz Oscillator and an external battery to continue running while the Game Boy is off. The RTC registers are mapped to the same area as RAM, and can be selected by specifying special RAM bank numbers to 4000h-5FFFh.

### **0000h-3FFFh – ROM Bank 0 (Read)**

This area contains the first 16KB of the ROM.

### **4000h-7FFFh – Switchable ROM bank (Read)**

This area may contain any 16KB ROM bank, including banks 20h, 40h and 60h (but not bank 00h).

### **A000h-BFFFh – RAM Bank/RTC Register (Read/Write)**

When a RAM bank in the range 00h-07h is selected, that RAM bank in the cartridge will be mapped to this area. When RAM banks in the range 08h-0Ch are selected, a single RTC register will be mapped instead. When a register is mapped here, all the address range can be used to access it. Usually A000h is used for that.

### **0000h-1FFFh – RAM and RTC Registers Enable (Write)**

Writing to this address range any value with 0Ah in the lower 4 bits enables RAM and RTC registers, and any other value disables them. Usually, 00h is used to disable them and 0Ah is used to enable them.

### **2000h-3FFFh – ROM Bank (Write)**

The 7 lower bits of the value written here is the ROM bank that will be mapped to 4000h-3FFFh. Writing a 00h here will select bank 01h instead. Any other value will select the corresponding bank.

### **4000h-5FFFh – RAM Bank/RTC Register (Write)**

This will select the RAM bank or RTC register to map to A000h-BFFFh.

### **6000h-7FFFh – Latch Clock Data (Write)**

Writing 00h and then 01h latches the current time into the RTC registers, and that values won't change until they are latched again. This is used for reading only, writing affects the actual RTC value (not sure if latched registers are updated too). It's supposed that the registers are updated in the 00h to 01h transition.

Some games like “Pokémon Crystal” write 00h, then 01h and then 00h. Other games like “Cardcaptor Sakura - Itsumo Sakura-chan to Issho (Japan)” write 01h most times, and 00h to 01h very few times.

RTC Registers:

08h	-	RTC Seconds	0-59 (0-3Bh)
09h	-	RTC Minutes	0-59 (0-3Bh)
0Ah	-	RTC Hours	0-23 (0-17h)
0Bh	-	RTC Lower 8 bits of Day Counter (0-FFh)	
0Ch	-	RTC Upper 1 bit of Day Counter, Carry Bit, Halt Flag:	
		Bit 0	Most significant bit of Day Counter (Bit 8)
		Bit 6	Halt (0=Active, 1=Stop Timer)
		Bit 7	Day Counter Carry Bit (1=Counter Overflow)

The Halt Flag should be set before writing to the RTC Registers.

The Carry Bit is set when the day count overflows from 511 to 0. This Carry Bit remains set until the program resets it. This means that a game containing a MBC3 RTC should be used at least once every 511 days for the game to know how much time has passed.

When accessing the RTC Registers it is recommended to execute a 4 cycles delay (normal speed

mode) between the separate accesses due to a slow hardware.

## **MBC5 (8MB ROM. 128KB RAM. DMG, SGB, CGB)**

This MBC is the one used by most GBC games. Some MBC5 cartridges have an electric motor used for rumble.

### **0000h-3FFFh – ROM Bank 0 (Read)**

This area contains the first 16KB of the ROM.

### **4000h-7FFFh – Switchable ROM bank (Read)**

This area may contain any 16KB ROM bank (including bank 00h?).

### **A000h-BFFFh – RAM Bank (Read/Write)**

This contains the selected RAM bank.

### **0000h-1FFFh – RAM Enable (Write)**

Writing to this address range any value with 0Ah in the lower 4 bits enables RAM, any other value disables it.

### **2000h-2FFFh – ROM Bank (Low bits) (Write)**

The value written here is the lower 8 bits of the ROM bank number.

### **3000h-3FFFh – ROM Bank (High bits) (Write)**

The value written here is the higher 8 bits of the ROM bank number. There is only one game with more than 256 ROM banks (“Densha de Go! 2 (Japan)” with an 8MB ROM), so only bit 0 is used. In games that use 256 banks or less this register selects the lower 8 bits of the ROM bank too (?).

**TODO: CHECK THIS AGAIN**

### **4000h-5FFFh – RAM Bank/Enable Rumble (Write)**

This will select the RAM bank or RTC register to map to A000h-BFFFh. In cartridges with rumble, writing a '1' to bit 4 will enable the electric motor, writing a '0' will disable it. Games with rumble can't have more than 8 RAM banks. The Game Boy can do some sort of PWM by writing '1' and '0' to this bit quickly with different waiting periods. This way the motor can vibrate with more or less intensity.

## 12. Credits

- Pan Docs, by “Pan of Anthrox”, with contributions from Marat Fayzullin, Pascal Felber, Paul Robson, “kOOPa” and (a lot from) Martin Korth “nocash”.
- The people at the IRC channel #gbdev in EFnet. Specially beware, for BGB emulator, which was really useful when designing test ROMs before checking them in real hardware.
- Jonathan Gevanyahu “Lord Nightmare” for GBSOUND.txt.
- Carsten Sorensen for RGBDS and Anthony J. Bentley for maintaining it in Github.
- Sindre Aamås for Gambatte and its source code.
- GeeBee for the GB DEV FAQs.
- Otaku No Zoku for the Gameboy Crib Sheet.



## 13. Changelog

First version: Not finished yet!