# Introduction to Gameboy hardware and software development

Jean-Michel FRIEDT, April 2, 2001

## 1 Hardware aspects

Connecting an EEPROM just for executing a program is straight forward as the Gameboy offers, with TTL levels, a 8 bit wide data bus, a 16 bit wide address bus and the two usual control signals RD# and WR#.

The address range allocated to the ROM is from 0000 to 7FFF: it is thus possible to the address line 15 as a Chip Select signal for the ROM (CS#). The signal for reading RD# of the Gameboy is connected to the pin OE# of the ROM (so that it defines the value on the data bus when being read from).

In order to allow, in addition to executing a program from ROM, to control digital output lines as well as read digital inputs from sensors connected to the Gameboy, one must add an address decoder (represented by the 74138 on the schematic in figure 2). Indeed, the communication with the world requires adding components – a latch for outputs (74574) and a 3-state bus-divider (74245) for inputs – which must be selectable independently from the ROM.

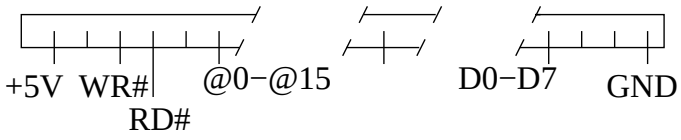


Figure 1: Gameboys connector, as seen from **top** of the console, the screen facing in the opposite direction from the viewer. The pins to the two ends of the connectir (+5V and GND) are correctly indicated, and the address bus (16 bits) and data bus (8 bits) are one next to the other.

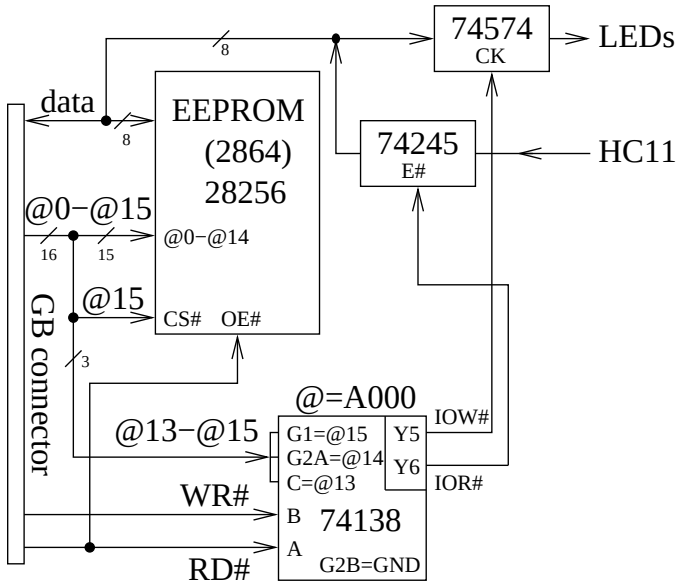


Figure 2: Schematic of a Gameboy cartridge

The address range allocated to external RAM (on the cartridge) starts at A000 (and finishes

at BFFF). We will use this address as a communication port between the Gameboy and the world. The ROM is not activated when this address is used as address bit 15 is then at a high level (as it is for all calls above address 8000 – remember that the ROM space extends up to 7FFF). Two possibilities are available: accessing to the world in input or output modes. During an output (writing), WR# is low and the Y5 output of the 74138 is triggered. During an input (reading), RD# is low and the Y6 output of the 74138 is triggered. One just has to connect these two signals, Y5 and Y6, to the trigger signals of the components used to isolate the Gameboy from the world (CK for the 74574, E# for the 74245).

## 2 Software aspects

At least two Gameboy development tools are available for Linux: `rgbds` is an assembler, while `gbdk` is a C compiler. These tools are more convenient than others Z80 compilers as they aumatically generate a header required at the beginning of the ROM for the program to be executed by the Gameboy (this header includes a checksum which must be correct, otherwise the ROM is refused when the Gameboy is switched on).

These two tools were used successfully for executing a program from ROM and to communicate with the world using the dedicated ports.

The assembly program is copied from the `galp` tutorial, which is an introduction to the `rgbds` assembler, and slightly modified at the end of the printing routine to access to the output port.

```
; jmfriedt, 5/02/01
INCLUDE "gbhw.inc"
        INCLUDE "ibmpc1.inc"
        SECTION "Org $100",HOME[$100]
        nop
        jp      begin

        ROM_HEADER      ROM_NOMBC, ROM_SIZE_32KBYTE, RAM_SIZE_2KBYTE

        INCLUDE "memory.asm"

TileData:
        chr_IBMPC1      1,8
begin:
        di
        ld      sp,$ffff

        call    StopLCD

        ld      a,$e4
        ld      [rBGP],a        ; Setup the default background palette

        ld      a,0
        ld      [rSCX],a
        ld      [rSCY],a

        ld      hl,TileData
        ld      de,$8000
        ld      bc,8*256        ; length (8 bytes per tile) x (256 tiles)
        call    mem_CopyMono    ; Copy tile data to memory

        ld      a,$20           ; Clear tile map memory
        ld      hl,$9800
        ld      bc,SCRN_VX_B * SCRN_VY_B
        call    mem_Set

        ld      hl,Title        ; Draw title
        ld      de,$9800+3+(SCRN_VY_B*7)
        ld      bc,13
        call    mem_Copy

; Now we turn on the LCD display to view the results!
        ld      a,LCDCF_ON|LCDCF_BG8000|LCDCF_BG9800|LCDCF_BGON|LCDCF_OBJ16|LCDCF_OBJOFF
        ld      [rLCDC],a       ; Turn screen on

ld a,$ff
wait:
dec a
ld [$a000],a ;   <- doit permettre d'ecrirer ff en RAM a000 <- LEDs
ld hl,$ffff
bll:
dec hl
jp NZ,bll
        jp      wait

Title:
        DB      "Hello World !"


; *** Turn off the LCD display ***

StopLCD:
        ld      a,[rLCDC]
        rlca                    ; Put the high bit of LCDC into the Carry flag
        ret     nc              ; Screen is off already. Exit.

; Loop until we are in VBlank

.wait:
        ld      a,[rLY]
        cp      145             ; Is display on scan line 145 yet?
        jr      nz,.wait        ; no, keep waiting

; Turn off the LCD

        ld      a,[rLCDC]
        res     7,a             ; Reset bit 7 of LCDC
        ld      [rLCDC],a

        ret

;* End of File *
```

The C program is easier to understand at first sight. `gbdk` offers a lot of functions, including graphics functions, and is thus easy to start with. However, the final binary code generated on such a simple example as the one presented here is 4 times larger (nearly 8 KB) than its equivalent written in assembly language.

```
/* Sample Program to demonstrate the drawing functions in GBDK     */
/* Jon Fuge jonny@q-continuum.demon.co.uk (modified by jmfriedt)   */
/* ../../bin/lcc -Wa-l -Wl-m -Wl-j -DUSE_SFR_FOR_REG -o jm.gb jm.c */


#include <gb/gb.h>
// #include <stdio.h>
#include <gb/drawing.h>
```

```
void main(void)
{UBYTE  *c,cpt,inp;int i;

 c=0xA000;cpt=0;                    // address of pointer <- write to @ 0xA000
 gotogxy(2,2);gprintf("hello world");
 gotogxy(2,4);gprintf("jmfriedt");
    /* Draw two circles, a line, and two boxes in different drawing modes */
```

```
color(LTGREY,WHITE,SOLID);
circle(140,20,15,M_FILL);
color(BLACK,WHITE,SOLID);
circle(140,20,10,M_NOFILL);
color(DKGREY,WHITE,XOR);
circle(120,40,30,M_FILL);
line(0,0,159,143);
```

```
color(BLACK,LTGREY,SOLID);
box(0,130,40,143,M_NOFILL);
box(50,130,90,143,M_FILL);
while (1) {cpt++;*c=cpt;for (i=0;i<6500;i++) {}
          inp=*c;gotogxy(2,6);gprintf("%d",inp);}
}                                          // use different vars for
                                           // counter and output pointer @
```

A useful tool, at least for the graphics display part, is an emulator. Two such programs are available under Linux: `vgb` (binary version only) and xgnuboy. The screenshots of the two latter programs are visible on figure 3.

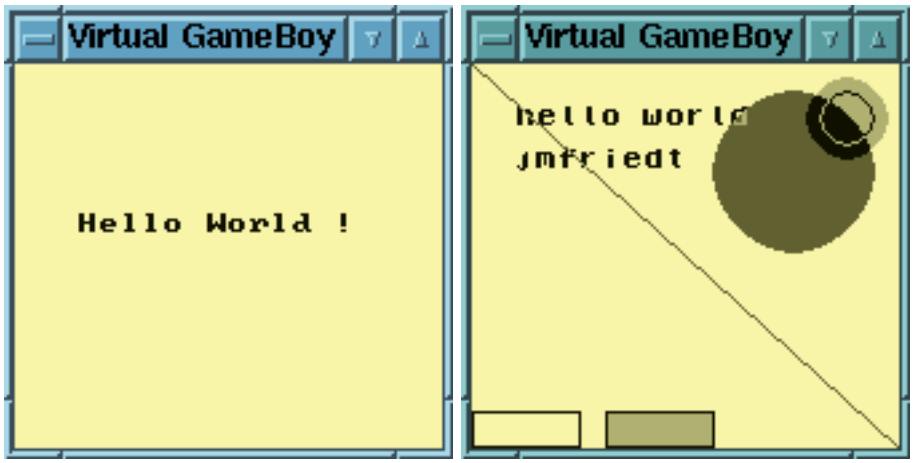


Figure 3: Sortie d'écran de l'émulateur `vgb`

# 3 Communication with a microcontroller

Development on the Gameboy itself is a painful process: plug EEPROM into 68HC11 programmer, program EEPROM, plug EEPROM on Gameboy extension card, test EEPROM, check what part of the program fails, program EEPROM again ... It thus seemed much easier to connect a more classical microcontroller (which include RS232 communication, A/D converters, several I/O ports ...) to the Gameboy, and have a simple and standard program on the Gameboy side just for displaying data while the complex operations are being done on the microcontroller. Because the microcontroller (in our case the 68HC11F1) has a bootstrap mode (*i.e.* can be programmed from its serial port), development is easy and quick.

We thus added to the Gameboy extension card a 74245 tri-state bus transceiver so that the Gameboy can not only send data and orders to the microcontroller (as demonstrated by out ability to power on and off LEDs connected to the 74574) but also read data from the microcontroller. This additional component's power consumption seems to go above the limit power the Gameboy can supply, and external power supply (+5V) was required for it.

A simple protocol was developed for one-way communication between the microcontroller and the Gameboy: every time the Gameboy seeks a datum from the microcontroller, it reads from the 74245 the value stored on the output (A) port of the microcontroller. As the Chip Select signal of the 74245 (as defined at the output of the address decoder on the Gameboy extension card) is also connected the the 68HC11 IRQ# line, the microcontroller knows a datum has just been read by the Gameboy: it starts a new conversion on its analog to digital converter and stored the new result on port A (this process must take less time than the delay between two Gameboy data requests, which in our case is about half a second) so that the new value is ready when the Gameboy will next request it. Obviouly, using one of the digital input ports on the 68HC11 side, more complex commands could be sent from the Gameboy to the 68HC11 by connecting it to the

74574 on the extension card (for example, using output-only port B of the 68HC11 for sending data to the Gameboy and using port A in read-mode for getting commands).

An example of such a setup is given bellow: notice that the curves read on the serial port of the 68HC11 and displayed on the Gameboy are similar. The 68HC11 was reading the voltage on the middle pin of a potentiometer connected to one of its A/D converters, and sending the result simultaneously to the Gameboy and on the serial port (for reading by a PC).

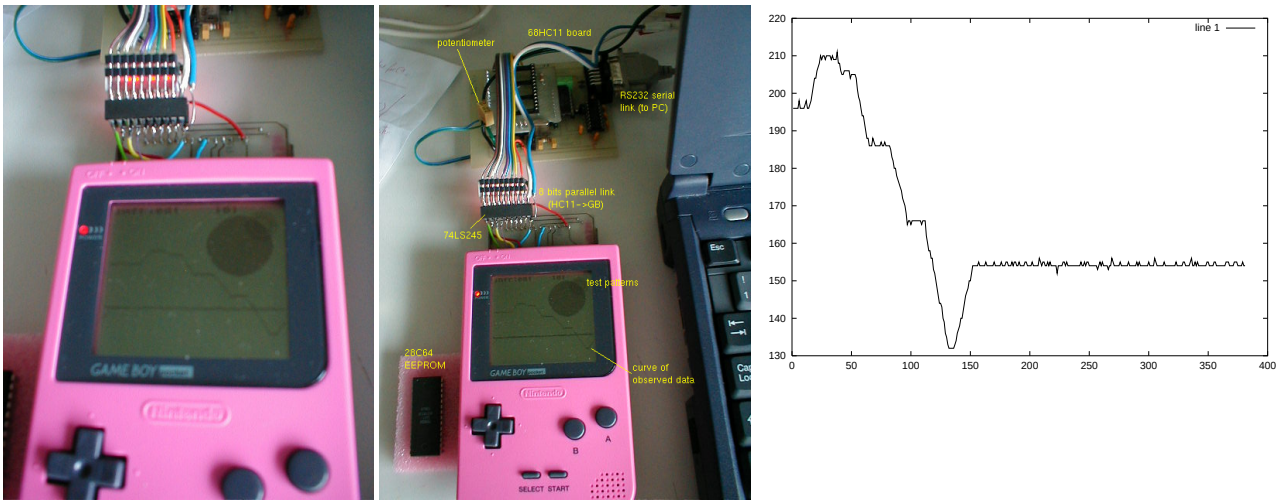


Figure 4: Connecting the Gameboy to a 68HC11(F1) microcontroller: notice that the curve to the right (as read on the HC11's serial port by a PC) and the curve displayed on the Gameboy screen (left: same data transfered from the microcontroller to the Gameboy) are similar.

Using a microcontroller as an extension of the Gameboy greatly increases the capabilities of this project, as all the usual functionalities of microcontrollers (timer, A/D converters, serial communication protocols, interrupts ...) are available to the Gameboy which otherwise does not allow access to these low level hardware functionalities.

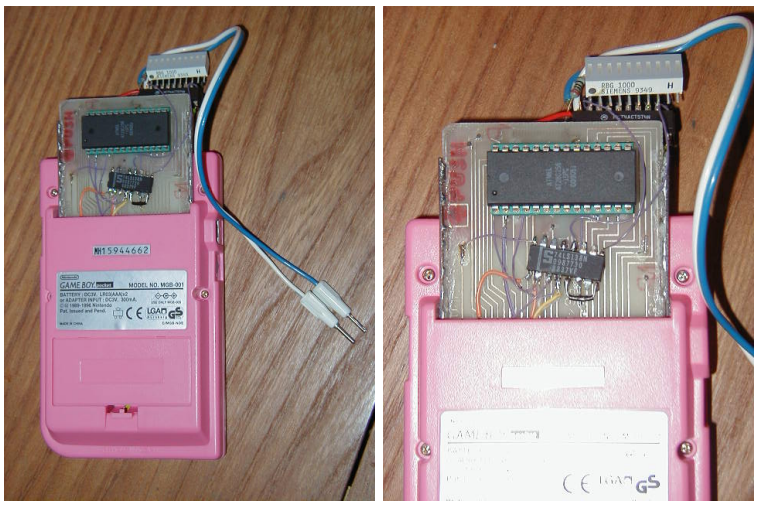

Figure 5: Rear views of the Gameboy extension card: notice the LEDs array for testing the output port, the address decoder (added to the card using wires) and the 28C256 EEPROM holding the program to be executed (also works fine with a 28C64 EEPROM).

A more efficient communication protocol could be developed using an intermediate FIFO mem-

ory (AM7202A for example). Indeed, in that case, the microcontroler (in our case a 68HC11F1 or a TMPZ84) writes data to be displayed on the screen of the Gameboy to the FIFO. The Gameboy waits for the data until half the memory is full (HF# signal enabled). At this moment, the Gameboy denies the access to the memory to the microcontroler, while it quickly reads the content of the FIFO until it is emptied (pin EF# activated meaning the FIFO is empty). The Gameboy then releases access to the FIFO to the microcontroler, plots the data it just read and waits for a new time series.

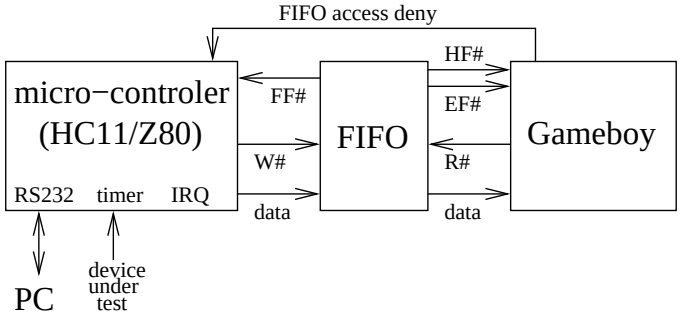


Figure 6: Schematic for the communication between a microcontroler and the Gameboy