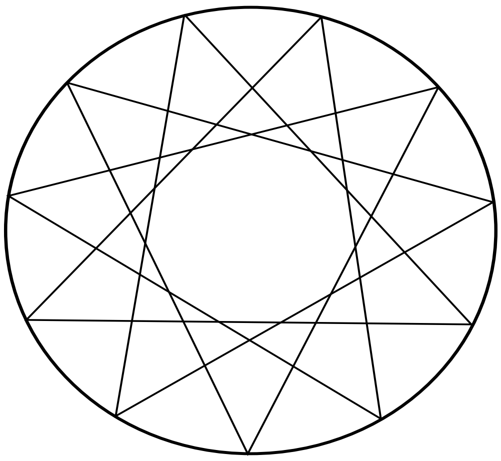
8-Bit Wonderland

Executing custom code on the Nintendo Game Boy

Version: 14.06.2010



Written by Belial
belial@apocalypsys.net

Contents

1 License

2 Introduction

The Nintendo Game Boy was presented in 1989 for the first time and has sold more than 118 million times. Now it is still possible to buy the original Game Boy which is a cheap 8-bit mobile device. Because it has no real protection mechanism like modern consoles against the injection of homebrew code it is possible to customize this mobile device and use it for different tasks. The circuit board layout, CPU, etc. are well documented[1] but it is difficult to collect all the distributed informations on the internet and there is a lack of tutorials too. The aim of this paper is to close this gap and it can be divided into two major sections:

- Developing a homebrew Game Boy cartridge
- Developing software for the Game Boy

It starts presenting an introduction to digital circuits and describes how the Game Boy works internally. After that it presents two ways to connect an EPROM to the Game Boy and describes how to build a custom cartridge[2].
When a custom cartridge has been build the Game Boy Developers Kit[3][4][5][6] makes it possible to write software for the Game Boy in ansi C and program the EPROM with it. As a first example a Pong game will be presented and used to illustrate the different coding aspects (video signals, avoiding expensive operations like multiplication, etc.)
The aim of this document is to describe the process of creating software for the Game Boy and a cartridge to execute it. Electrotechnical details are explained but a minimum knowledge of programming skills and how a computer works are still needed.

3 Game Boy Basics

Before we start developing a homebrew cartridge which can be connected to the Game Boy, the knowledge of some important basics is necessary. This section starts with a brief description of the Game Boy and its memory layout. It is followed by some basics about digital hardware which cover topics like bused and bank switching.

3.1 Features and Memory layout

Nintendo released the *Game Boy Classic* in 1989 as a mobile gaming device. Since then, it was followed by some successors like *Game Boy Pocket*, *Game Boy Advanced*, etc. This paper deals with the *Game Boy Classic* from 1989 which has the following features:

- CPU: 8-bit (similar to the Z80 processor)
- Main RAM: 8K Byte
- Video RAM: 8K Byte

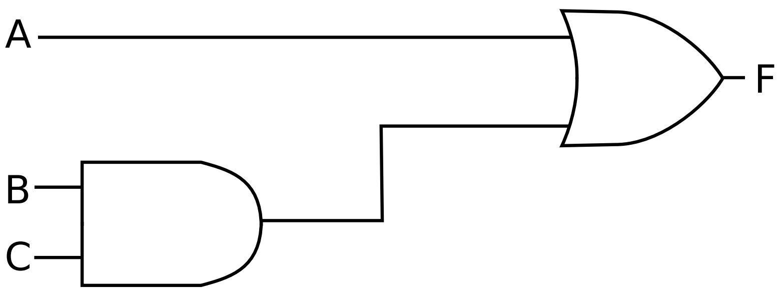


Figure 1: Logical abstraction Level

- Clock Speed: 4.194304 MHz
- Games come on special cartridges containig a ROM which holds program code and data.

The 16 Kilobytes of RAM (Video RAM + Main RAM) are mapped in a 64 Kilobyte address space which has the following layout:

- 0x0000 - 0x3FFF: external 16KB ROM
- 0x4000 - 0x7FFF: external switchable 16KB ROM bank
- 0x8000 - 0x9FFF: internal 8KB Video RAM
- 0xA000 - 0xBFFF: external switchable 8KB RAM bank
- 0xC000 - 0xDFFF: internal 8KB Main RAM
- 0xE000 - 0xFFFF: I/O ports, Interrupt Enable Register, Sprite Attributes, etc...

This means a game is stored on a 32KB external ROM and can access 16KB of internal RAM. To extend this, Nintendo has implemented two features: 1) external RAM can be installed on a cartridge and 2) bank switching (see section 3.4 for details).

3.2 Digital Circuits

This section describes how digitals circuits work. It covers how 1 and 0 are mapped to physical hardware and how they can be used to implement e.g. addition, substraction, etc. We use two abstraction layers, which help understanding the structure of a digital circuit. *Logic Level* and *Circuit Level*.

3.2.1 Logic Level

On this level, every component of a digital circuit consists of a network of boolean logic operators (AND, OR, XOR, INVERT, ...). Components are connected with each other. Each component or boolean logic operator has input and output connections. This is shown in figure 1. A component sends/receives either 1 or 0 on/with its output/input pins. In figure 1 the input

Input B	Input C	Output
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2: Logic table for digital AND Gate

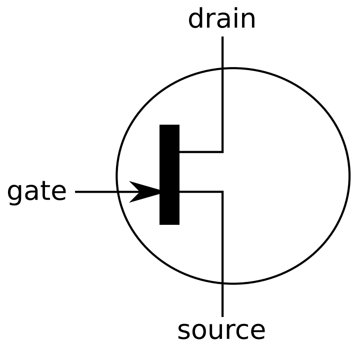


Figure 3: field effect transistor

pins B and C are connected with an AND gate. Its output is connected with the input of an OR gate. An AND gate works like the AND in boolean algebra. Its behavior can be described with a table showed in figure 2. This means, when both input pins receive a 1, it writes a 1 on the output. For a detailed description to boolean algebra and other operators like OR, XOR, etc. we refer to [7][8].
With a few AND and OR components a simple 1-bit adder can be made which is an important part in every CPU. To be able to add two 8-bit numbers, 8 1-bit adders can be concatenated to an 8-bit adder. Of course its also possible to create substractors, multiplier, etc...
This means, each CPU, Microcontroller or the components on your mainboard are just a complex network of millions of AND, NOR,... gates and FlipFlops. FlipFlops are also part of a digital circuit, can store 1 bit of data and acts as a kind of memory[9].

3.2.2 Circuit Level

This section describes how the logical abstraction layer is implemented physically on a chip. To understand this we take a look at the circuit level. Every AND, OR, NAND, inverter, etc. component is a network of field effect transistors shown in figure 3. The important physical unit to control them is Volt. The 1 and 0 send through the connections of the different components are represented by a special voltage on circuit level. E.g. a voltage between 0 and 0,5 is interpreted as a logical 0 and a voltage between 4 and 5 Volt is interpreted as a logical 1. A field effect transistor can be understood as a simple switch or a resistor. There are two versions of field effect transistors.

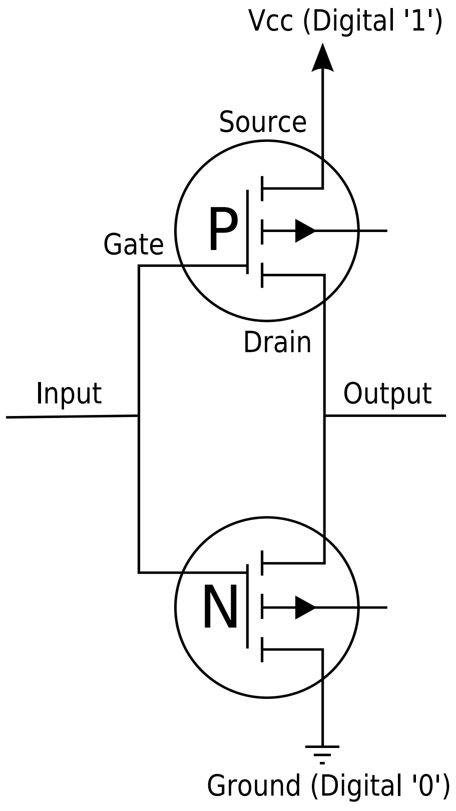


Figure 4: Digital Inverter on circuit level

- **pnp-transistor** works like an adjustable resistor or a switch between drain and source. If there is a low voltage between gate and source (logical 0), the resistor has a very high value (the switch is opened). If theres high voltage between gate and source (logical 1), the resistor has a low value (the switch is closed).
- **npn-transistor** works like an inverted pnp-transistor. A low voltage between gate and source (logical 0) leads to a low resistor value between drain and source (the switch is closed), a high voltage (logical 1) leads to a high resistor value (the switch is open).

In general you can understand a field effect transistor as a switch which state is controled by input voltage between gate and source. Now we will take a look how these transistors can be combined to a component on the logical abstraction level. Our example is the inverter. As you may imagine, an inverter simply inverts its entry signal (as shown in figure 6). The schemetics of such an inverter on circuit level are shown in figure 4 and on logic level in figure 5.
VCC and ground can be interpreted as + and - from a power supply (like a battery e.g.). If the inverter has a high voltage (logical 1) on its input, the pnp-transistor (marked in figure 4 with an N) would pull its output to ground (because it has a very low resist between gate and source). This means, you will have a very low voltage on the output (logical 1). If the inverter has a low voltage on its input (logical 0), the npn-transistor (marked in figure 4 with a P) has very low resist between drain and source. This leads to a high voltage (nearly Vcc) between output and ground (logical 1). Vcc and Ground are often not shown on logic level for simplification.

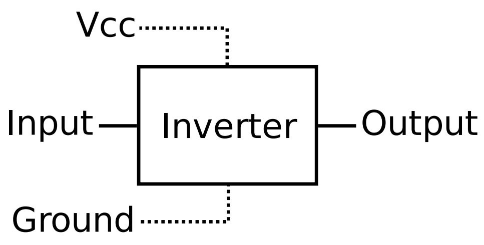


Figure 5: Digital Inverter on logic level

Input	Output
0	1
1	0

Figure 6: Logic table for a digital Inverter

3.3 Data- and Addressbus

Before we create a homebrew cartridge which contains our code, it is important to know how the Game Boy communicates with a cartridge. In the last sections we have seen that components send 1 and 0 over the wires, but for complexer components we use a higher abstraction level. This is done with a data- and an addressbus. A bus is a communication channel between two components. This does not need to be a Game Boy and its cartridge but can also be a CPU and RAM or two PCI-Cards e.g. There exist several different buses but their basics are always the same: There are wires for data, wires for addressing and some control signals (some buses like I2C use for address-, data- and control signals the same wire(s) but thats not important in this paper).
The main unit on a Game Boy cardridge is an EPROM. Code and data are stored on an EPROM. It is a small chip which has a certain size like 32 kilobytes. An EPROM is a *Read Only Memory* and only read operations are performed on it. Because cartridges can contain RAM too, the Game Boy has also the possibility to write data with its databus to the cartridge. This means, the Game Boy needs two control signals, to tell the cartridge that either a *read* or a *write* operation is performed.
As written above, a cartridge can contain additional RAM. This means RAM and EPROM share the same data- and addressbus. To to tell the cartridge whether RAM or EPROM are accessed, another signal is necessary: *chip select*. When operations on RAM are performed, *chip select* is set.
To read a byte from the EPROM, the Game Boy writes its address on the addressbus, e.g. 0x003F. Because this is a read operation, the *read* control signal is set to 1, *write* and *chip select* signal to 0. The EPROM on the cartridges receives these signals and writes the data, which is stored at address 0x003F, on the databus. On circuit level, these buses are simply electric connections from the cartridge to the Game Boy. A single wire transfers one bit: 1 or 0. This means, if a databus consists if 8 signals, the Game Boy can read values between 00000000 and 1111111 from the cartridge (0x00 - 0xFF) and is called an 8-bit databus.

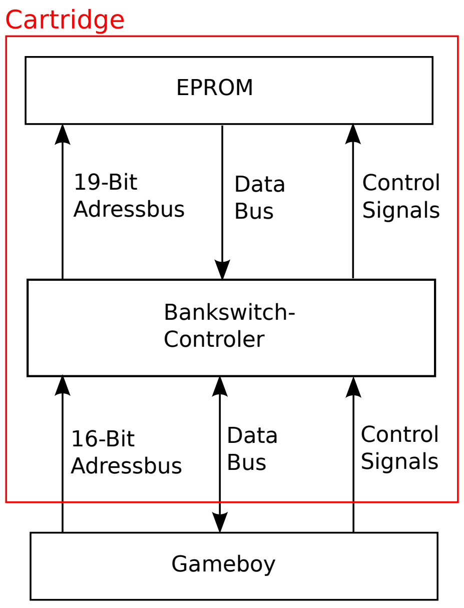


Figure 7: Game Boy connected to an EPROM with a Bankswitch Controller

3.4 Bankswitching

We have seen in section 3.1 that the Game Boy can address 64KB of memory. We will see in section 4.1 that the Game Boy is connected with a 16-bit addressbus to a cartridge.
The memory layout in section 3.1 shows, that only 32KB of external ROM are mapped into Game Boys memory (0x0000 - 0x3FFF and 0x4000 - 0x7FFF). Games need often more space on a ROM. This is where *Bank Switching* becomes interesting.
Each EPROM has a maximum size. Imagine an EPROM in which fit exactly 512KB of data. This means we need a 19 bit address bus to access all data. But what to do if our hardware can only map 32KB in its memory and has only a 16-bit address bus? In this case, a special component called *Bank Switch Controller* (BSC) is connected between the Game Boy and the EPROM.
This is shown in figure 7. It works like a state machine. 512KB of the EPROM space are divided into 16KB large section called *Banks*. (*Bank 0* : 0x0000 - 0x3FFF, *Bank 1* : 0x4000 - 0x7FFF, etc.) The *Bank Switch Controller* decides in dependence of its state which *banks* of the EPROM are accessed by the Game Boy. When the Game Boy is turned on, the BSC is in state s_0. Figure 8 shows the output of the BSC in dependence of

1. its state
2. its input

We see, input addresses between 0x000 and 3FFF are just forwarded to the EPROM. When the *Game Boy* writes addresses between 0x4000 and 0x7FFF on the address bus, the *Bank Switch Controller* changes them in dependence of its state. This means: Game Boys memory between 0x0000 and 0x3FFF is static and contains always *Bank 0* of the EPROM. Region 0x4000-0x7FFF is dynamic. It contains in s_0 *Bank 0*, in s_1 *Bank 1*, in s_2 *Bank 2*, etc.
To control the BSC's state, it contains registers. A register is a small memory region (e.g. 8-bit),

BSC State	GB $\rightarrow$ BSC	BSC $\rightarrow$ EPROM
s_0	0x0000 - 0x3FFF	0x00000 - 0x03FFF
s_0	0x4000 - 0x7FFF	0x00000 - 0x03FFF
s_1	0x0000 - 0x3FFF	0x00000 - 0x03FFF
s_1	0x4000 - 0x7FFF	0x04000 - 0x07FFF
s_2	0x0000 - 0x3FFF	0x00000 - 0x03FFF
s_2	0x4000 - 0x7FFF	0x08000 - 0x0AFFF
s_3	0x0000 - 0x3FFF	0x00000 - 0x03FFF
s_3	0x4000 - 0x7FFF	0x0B000 - 0x0EFFF
...	...	...

Figure 8: *BSC Input* and *BSC Output* in dependence of *BSC State*

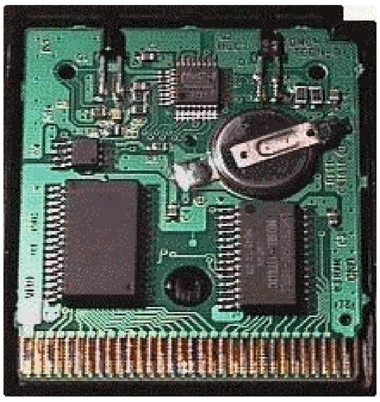


Figure 9: Game Boy Cartridge Circuit Board

which can be implemented with FlipFlops. When the Game Boy accesses a special memory region, the controller won't access the EPROM but its internal register instead. This means, when a program writes at address 0x3000, it will write in the BSC's state register. The value in the register sets the state of the BSC. When writing a 0x01 to it, it will switch from state s_0 to s_1 and so on.
Game Boy has not only the possibility to extend ROM space via bank switching but also to extend on-cartridge RAM with bank switching. The only difference is that RAM banks have a size of 8KB and are mapped in memory region 0xA000 - 0xBFFF and 0xC000 - 0xDFFF.

4 Game Boy Cartridge

4.1 Components

We know now the necessary basics and will take a look at a real cartridge. Every cartridge consists of a circuit board and a plastic case surrounding the circuit board. Figure 9 shows the circuit board of an opened Game Boy cartridge. It has the following components:

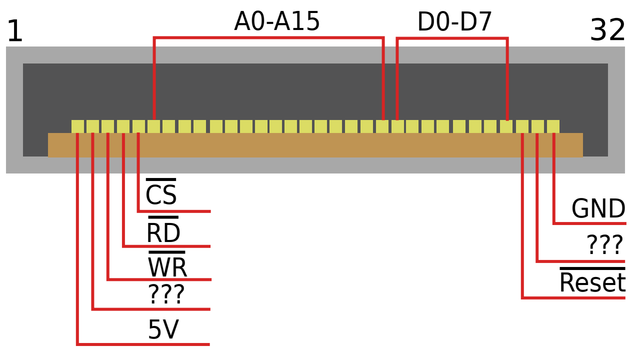


Figure 10: Bottom of a Game Boy Cartridge and its Pin

- EPROM: As explained above, the game itself is stored on the EPROM.
- RAM: The Game Boy contains 16 Kilobytes of RAM, but this is not enough for some games. Because of this, cartridges can contain extra memory. Games use this extra memory for storing gamestates, etc...
- Battery: RAM is not persistent memory like a HD. When it loses contact with power supply, all data is erased. This happens e.g. when the Game Boy is turned off. To store gamestates, RAM needs an extra powersupply.
- Bankswitch- and memory controller: As explained above, for bankswitching an extra controller unit named BSC is necessary.

4.2 Pins

You can see a schematic in figure 10 which shows the bottom of a cartridge and its connection pins. Stroked pins are inverted and called *active low*, which means they are active when 0 is set instead of 1. The semantics of cartridge pins can be seen in figure 4.2. Besides the address- and databus (A0-A15 and D0-D7), a cartridge has the pins Vcc, Ground (see section 3.2.2), the control signals WR, RD, CS, Reset and two undocumented pins. The *reset* pin was not mentioned before. It can be used to reset components on the cartridge like the BSC (bringing it back to state s_0). Documentation from other researchers says about the unknown pins that one is a clock signal, the other deals with audio. Both are not relevant for this work, so i won't discuss them here.

4.3 Development of a Homebrew Cartridge

We start now with the the creation of our own Game Boy cartridge. There exist two ways to do so.

- Modify an existing cartridge
- Create your own cartridge from scratch

When we speak about *creating a cartridge*, the internal cartridge circuit board is meant. For a very simple cartridge we don't need RAM and BSC. All we need is a 32KB EPROM (2 Banks can be completely mapped into Game Boys memory without a BSC).

Pin number	Name	Description
01	+5V	Vcc
02	?	–
03	WR	If a cartridge contains extra RAM, this signal is set to 0 (its active low) if a write operation is performed
04	RD	This Read signal is set to 0 (its active low) if the Game Boy performs a read operation on RAM or ROM
05	CS	This signal is called “Chip Enable”. It can be used to turn possible RAM on cartridge on or off if a Read/Write operation is performed on it. Its like RD and WR active low.
06 - 21	A0 - A15	16-bit addressbus
22 - 29	D0 - D7	8-bit databus
30	Reset	Reset components on the cartridge like BSC state
31	?	–
32	GND	Ground

Figure 11: Game Boy Cartridge Pins Description

4.3.1 EPROM

We have chosen to use an EPROM named 27C256 which is shown in figure 12. The 27C257 has a size of 32KB and operates in two modes:

- Normal Mode: When assembled on a cartridge it operates in normal mode. Only read operations are possible
- Programming Mode: Data can be written on the 27C256.

You can write on it in programming mode with a special “writing device”. There exist professional writing devices which are very expensive. But you can also buy cheap ones which can’t write on every EPROM and are more for “hobby usage”. I bought mine for 30 euros on a well known internet marketplace.
It is possible to delete the content of a 27C256 too. This is done with ultra violet lightrays (thats why the EPROM in figure 12 has this small glass window). Bright sunlight can erase the content of a 27C256 but this would take several years. This is very slow and because of this exist special devices which create ultra violet lightrays and erase a 27C256 within minutes.
The 27C256 pin layout is shown in figure 13. VPP is needed to set the EPROM in writing mode (which is done by the EPROM writing device talked about above). In normal mode, it is set to +5 Volt and be connected together with VCC to the power supply. CE can be interpreted as an on/off switch for the device. If set to 1 (remember its inverted), the 27C256 goes into standby mode. In standby mode, the EPROM consumes less energy (from 20 milli-ampere down to 100 micro-ampere). OE is used to turn on/off the output on the databus. Its importent when several components share the same databus for output (like EPROM and RAM) that only one components output is enabled. This won’t happen on our homebrew cartridge because we have only one 27C256 on it.

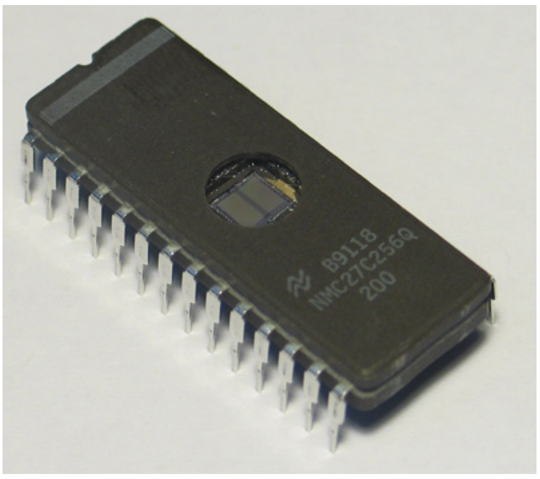


Figure 12: 27C256 EPROM

Pin	Name	Description
01	OE	Inverted Output Enable
07	VCC	+5V Power Supply
08	VPP	Needed for programming the EPROM
21	VSS	Ground
27	CE	Inverted Chip Enable
2-6, 9-17, 28	Address Bus	–
18-20, 22-26	Data Bus	–

Figure 13: 27C256 Pins Description

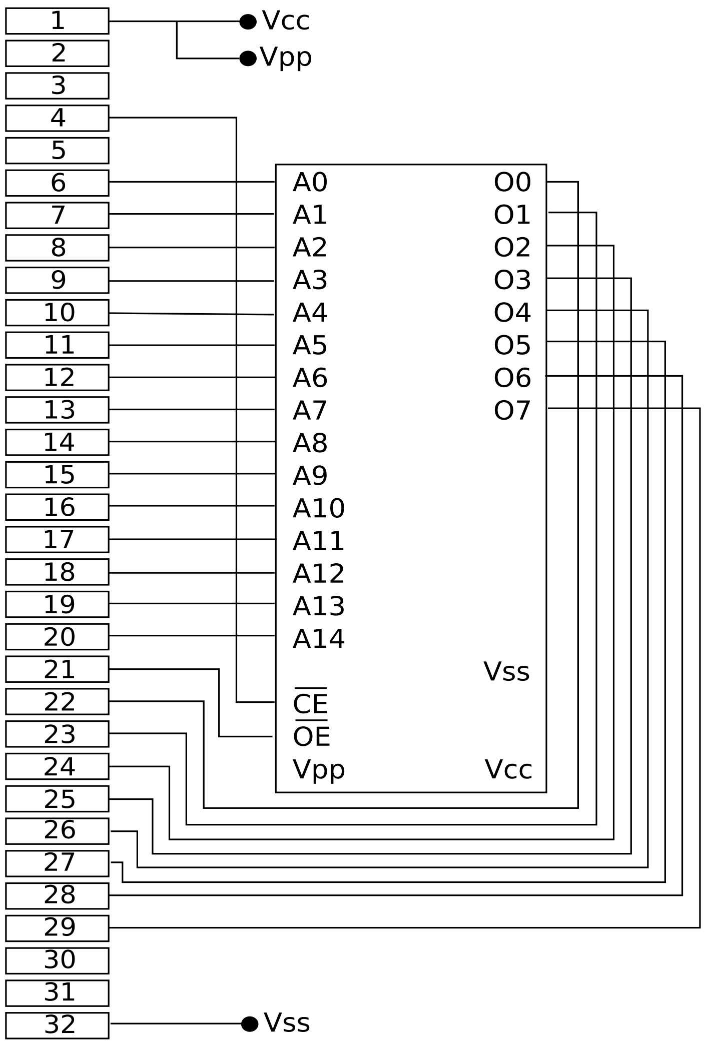


Figure 14: Homebrew Cartridge Layout

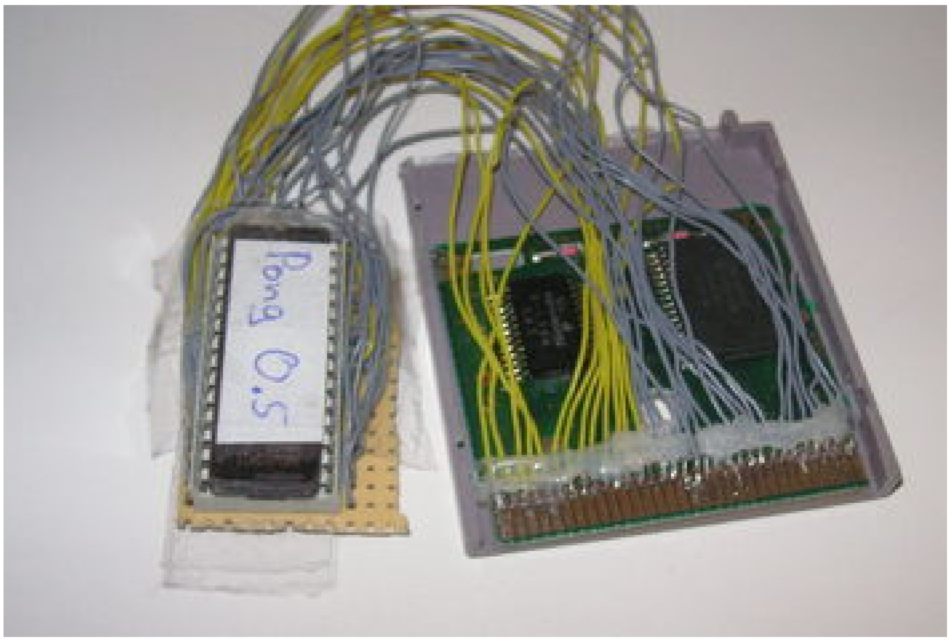


Figure 15: Modified custom Game Boy Cartridge connected to a 27C256

4.3.2 Layout

We know now enough about our hardware to specify a layout for the cartridge. You can see the complete layout for a simple homebrew cartridge in figure 14. It shows the Game Boy pins connected with a 27C256.
The Game Boys powersupply on pin 1 is connected with VCC and VPP of the 27C256. Game Boys ground (pin 32) gets a connection with the EPROMs VSS pin. The Game Boy *read* signal at pin 4 is connected with the CE input of the EPROM. Through this design, the 27C256 is only turned on when the Game Boy wants to read from it, the rest of the time it remains in standby mode to save power. The databus of the Game Boy is directly connected with the EPROMs data output. The same is done with the Game Boy and EPROM address bus pins, with one little difference:
The Game Boy has a larger address bus (16 bit) than the EPROM (15 bit). We know that the highest Game Boy adressbus bit will always be 0 (if set to 1 we would try to access memory >32KB). Because of this, it is connected with the OE signal of the 27C256.

4.3.3 Assembling the Cartridge

Our first attempt to create a homebrew cartridge is modifying an existing cartridge (paperboy, may it rest in peace;). Figure 15 shows a photo. The cartridge pins are connected with an EPROM which contains a homebrew Pong clone. This Pong Clone is described in section 5.5.
We destroyed with a small saw all connections from the cartridge pins to to the original cartridge content and assembled at each pin a wire (fixed with liquid glue). During the next step, a socket for the 27C256 is assembled on a circuit board and connected with the wires. We use the connection layout which was described in section 4.3.2. This first prototype is a very cheap way to create a simple cartridge.
A second way creating a homebrew cartridge is to design and create a circuit board instead of

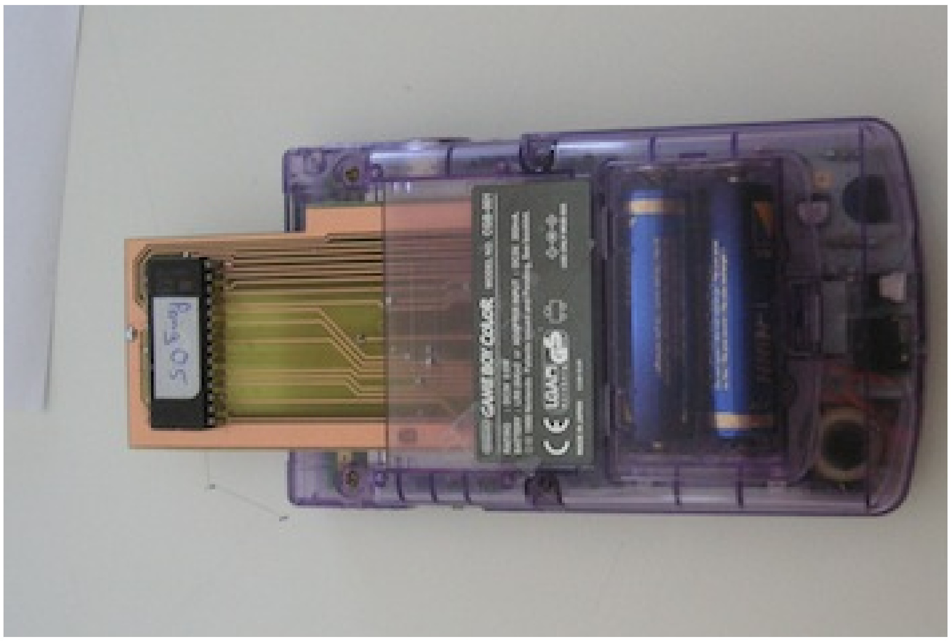


Figure 16: Back of homebrew cartridge

modifying an existing cartridge. It is important that the connection pins of this circuit board have the same size like on a original cartridge to fit in the Game Boy cartridge slot. We have created a layout for such a cartridge with the Eagle CAD tool[10] and you are allowed to use and modify it. Photographs are shown in figures 16, 17 and 18.

5 Game Boy Software

We have described in the last chapter, how to develope a homebrew Game Boy cartridge. This chapter describes how to write software, which can be written on the EPROM and executed by the Game Boy. It is structured the following way:

- Subsections 5.1 - 5.4 describe the necessary basics: ROM layout, accessing a video device and the development kit
- Section 5.5 is our first example application: a Pong clone

5.1 ROM Layout

Like a windows portable executable, the data on a Game Boy cartridge consists of two sections:

- a header, which contains information about the ROM: supported Game Boy model, whether it contains a bankswitch controller, etc.
- the code itself

[1] describes the layout of the header mentioned above in detail. We will give in this section just a brief overview. The header begins with a sequence of pointers (interrupts, etc.). Adress 0x0100 is the execution entry point of a Game Boy cartridge. It has a size of 4 bytes and is in

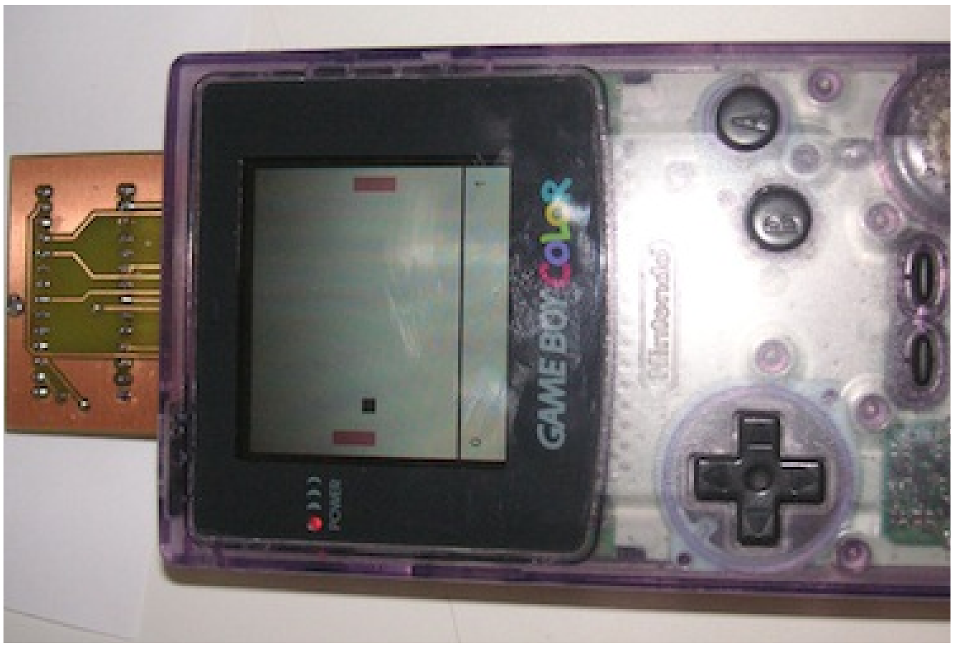


Figure 17: Front of a homebrew cartridge 1

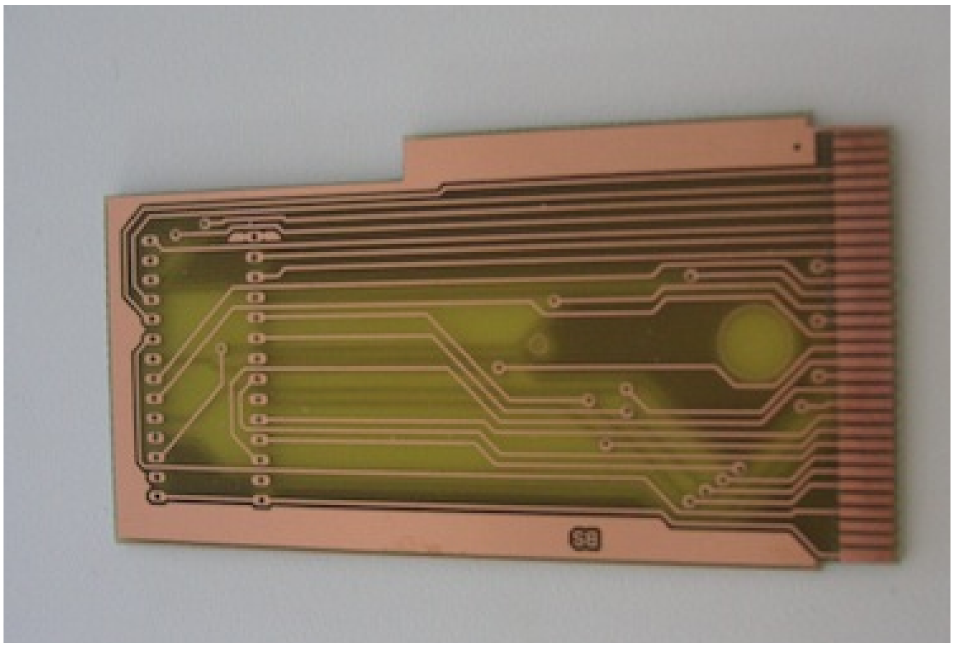


Figure 18: Front of a homebrew cartridge 2

most cases a jump instruction to the real entry adress. The execution entry point is followed by the data of the scrolling Nintendo logo which is shown after the device has been turned on. The logo is follwed by informations about the cartridge type (ROM size, cartridge RAM size, bankswitch controller type, etc.). A complement check and a checksum are stored at the end of the header (checksum is ignored by Game Boy Classic and only important for newer Game Boy revisions).

5.2 Boot Sequence

Again, this chapter is just a summary of [1]. When the Game Boy is turned on, two things happen:

1. Boot program from an internal boot ROM is executed.
2. Control is passed to ROM on the Game Boy cartridge.

The internal boot ROM starts reading the Nintendo logo from the cartridge and displays it on screen (its scrolled down). The two musical notes are played and the Nintendo logo from the cartridge is read again. It is compared with a copy of the logo data on the internal boot ROM. Execution stops if the logo data on the cartridge differs from the logo data on the internal boot ROM, otherwise it continues. All bytes from the adresses 0x0134 to 0x014d and 25 are added (remember, 0x0104d was the complement check value). The internal boot ROM is disabled if the last significant byte of this addition is 0 and adress 0x0100 on the cartridge is executed, otherwise execution stops. It is important to remember this when modifying an existing cartridge or create a cartridge header from scratch.

5.3 Video

This chapter covers video timings and video data. For a bedder understanding of video timing, we will introduce NTSC. After introducing NTSC and applying it to the Game Boy, we will explain how to write data on the Game Boy display.

5.3.1 Synchronisation

NTSC is an analog televion system used in north america (european countries use PAL which differs in video frame height, width and color encoding). It contains:

- Video frames
- Video synchronisation signals
- Audio data

NTSC uses a certain color space to encode pixel data in an (interlaced) video frame. Video and audio data can be transmitted on carrier frequency. NTSC TV signals use a resolution of 720x486 and 30 video frames are transmitted per second (which means NTSC has frequency of 30 Hz).
But that is not important for accessing the Game Boy display and not discussed in detail. We refer instead to the NTSC specification [11]. This chapter focusses on video synchronisation signals because knowledge about them helps understanding how to access the display on a gaming device (not only Game Boy, but also Atari 2600 e.g.).
NTSC was developed in 1941 when no modern LCD display existed. TV's created a video

frame with a cathode ray tube. A single electron beam scans a phosphor screen from left to right and then returns to the top. The synchronisation signals of the NTSC signal are necessary to control the TV's electron beam. There exist two synchronisation signals:

- HSYNC
- VSYNC

HSYNC signals a TV that the end of a scanline is reached (a video frame consists of several lines, so called scanlines). The TV needs some time to position the electron beam from the end of the old scanline to the beginning of the new scanline. This time window is called *horizontal synchronisation phase*.
VSYNC signals a TV, that the end of a video frame is reached and the electron beam has to be adjusted from the lower-right of the display to the upper-left to create the next video frame. This re-adjustment takes some time and is called *vertical synchronisation phase*.
The Game Boy does not contain a cathode ray tube, but it exits also a *vertical synchronisation phase*. This is important for Game Boy software development, because the developer has to take care about not accessing the video RAM while the Game Boy is reading from it and writing the next video frame on the display. Instead, Game Boy software should use the following scheme:

1. Game logic
2. Wait for vertical synchronisation phase
3. Update Video RAM
4. Goto 1

5.3.2 Tiles

We described in the last section the timings of video signals. We will describe in this section how video data on the Game Boy is structured. The Game Boy displays two different kind of graphics:

- Sprites
- Background

Both consist of several so called *tiles*. A *Tile* is a 8x8 image and has a size of 16 bytes. Figure 19 shows an example tile. Every line consists of 8 pixels and each pixel has one of four different greyscale values. Every line is mapped with two bytes into memory. Figure 19 shows an example tile. Figure 20 shows line 1 of this tile and its bit-encoding in memory (presuming it is mapped into memory at adress 0x00). The four greyscale values are represented by the bit encodings 00, 01, 10 and 11. All bit tuples in figure 20, which are used for pixel encoding, are marked with a red square. Arrows are pointing to the corresponding pixels of tile line 1.
A single *tile* can be implemented by hand in C with a 16 byte character array. This is very time consuming and not advisable for larger projects with many *tiles*. Therefore there exists tools which allow you to paint *tiles* in a pixel raster and export them as C character arrays. We have used for this project the *Game Boy Tile Designer* (GBTD) which can be found at `http://www.devrs.com/gb/hmgd/gbtd.html` (send us a mail if the link does not work for you).
We have seperated at the beginning of this section Game Boy graphic elements into background

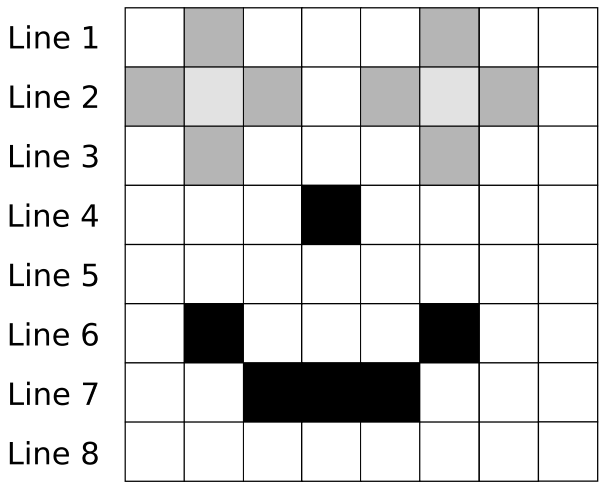


Figure 19: 8x8 Game Boy Tile

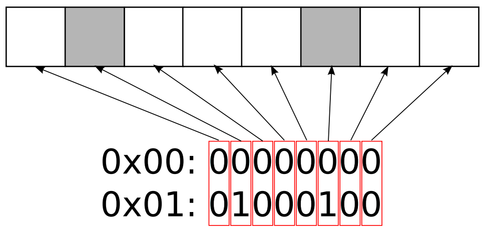


Figure 20: Tile Encoding

and sprites. Both are just *tiles*, written into the video RAM and displayed on the screen at certain coordinates. The background (actually there is more than one memory region for the background but this is not important in this work, we refer instead again to [1] for detailled information) has one peculiarity: The video RAM has a 256x256 pixel buffer for the background. The Game Boy can display only 160x144 pixels at once but contains *scrolling registers*. Those registers control which pixels in the background buffer become visible and which are shifted out.

5.4 Gameboy Developers Kit

We have described in the last sections the software and hardware aspects of Game Boy development. We have not described yet how to access them as a programmer: E.g. how to wait for vertical synchronisation, how to access the video RAM or how to generate the header of a ROM (see section 5.1).
One possibility to do so is to use an assembler. An assembler gives us a high optimisation potential and creates (in most cases) faster and smaller code than a high level language (HLL) compiler (e.g. some games on the Atari 2600 console can only be implemented with an assembler because of the limited hardware). On the other hand it has also some disadvantages compared with a HLL:

- Assembler code is harder to maintain and to extend.
- Assembler code leads to an higher amount of programming errors.
- It takes more time to create assembler source code because of its complexity.
- Knowledge of the target CPU architecture is necessary (Opcodes, Interrupts, Registers, Memory Layout, etc.).

Assembler is the bedder choice for complexer and larger games. Our case study in section 5.5 is less complex and for demonstration purpose only. Therefore, we will use a HLL in this work. The *GameBoy Developers Kit* (GBDK) [3] provides a C compiler, linker, preprocessor, etc. for the generation of Game Boy ROM files. We will present in section 5.5 an example application [12] which contains a *Makefile*. This *Makefile* can also be taken as an example for the GBDK workflow and the compiler flags. A description of the compiler flags, *Makefiles*, etc. is also found in [5].

5.4.1 Coding Guidelines

While writing C code with the GBDK for the Game Boy, you have to remember some important guidelines (desribed in detail at [3]):

- Initialized global variables are located on the ROM and therefore read-only. Uninitialized global variables are located in RAM and writable.
- The CPU in the Game Boy is an 8-bit CPU and optimized for 8-bit arithmetics. You should use 8-bit unsigned variables whenever possible. The compiler can of course handle larger datatypes too, but arithmetics with larger datatypes will take much more time.
- Avoid multiplication, division, modulo and floating point variables. The Game Boys CPU has no native support for them. Instead, the compiler will emulate these operations with software which is very costly and inefficient.

- The operators $!=$ and $==$ are more efficient than $<=$, $<$, etc.

These guidelines are restrictive. Nevertheless, it may be sometimes necessary to use floating point arithmetrics or complexer mathematical functions (e.g. sin(x), tan(x), etc.). A possible solution to implement e.g. $sin(x)$ is the approximation with a taylor polynom which will include several multiplications and a division with floating point variables. This is not advisable, because we have already seen that these operations are very costly on the Game Boy.
A faster (but more inaccurate) solution is the creation of a lookup table.

Listing 1: Example for Sinus Lookup Table Generation

```java
import java.lang.Math;
class Lookup{
    static float prec = 0.1f;
    static float treshold = 2f * (float) Math.PI;
    static int linebreak = 5;

    public static void main(String[] argz){
        System.out.println("float lookup[] = {");
        int linecount=0;
        for(float i=0;i<=treshold;i+=prec){
            //print next sin(i) and add "," if necessary
            System.out.print(Math.sin(i));
            if(i+prec<=treshold) System.out.print(", ");
            //add newline if necessary
            linecount++;
            if(linecount%linebreak==0) System.out.println("");
        }
        System.out.println("};");
    }
}
```

Listing 1 is a small *Java* example which creates a C *float* array of sinus values. It is controlled with three static variables which determine the resolution of the lookup table, its range and the total amount of elements per line in the source code (for bedder readability).
Multiplication can also be implemented with lookup tables. To do so, we make use of the logarithm function [13] and its computation rule:
$log_a(x * y) = log_a(x) + log_a(y)$
which leads to:
$x * y = log_a^{-1}(log_a(x) + log_a(y))$
A multiplication is transformed into an addition, which leads to a speedup in execution time. The computation of $log_a(x)$ and $log_a^{-1}(x)$ can be realized with lookup tables discussed above. These lookup tables are not limited to the computation of a multiplication. A division e.g. can be expressed with $\frac{x}{y} = log_a^{-1}(log_a(x) - log_a(y))$ and so on. See [13] for more details about the logarithm function.

5.4.2 APIs

The GBDK APIs are described in detail in [5]. We will give in this section just a brief overview, to give you an idea how the GBDK APIs work. Before we take a look at the APIs themself, we have to introduce some regions in the VRAM:

- Tile Data Table (TDT): Contains the tile data (16 bytes per tile, see 5.3.2). Background and sprites have their own TDT.
- Background Tile Map (BTM): Contains the tile numbers which are displayed on the Game Boy screen. The tile numbers refer to the background TDT.
- Object Attribute Table (OAT): Contains information about the sprites which are displayed on the Game Boy screen. It is organized in so called blocks. Each blocks has a size of 4 bytes and consists of a tile number (refers to the sprite TDT), X and Y coordinate on screen and one byte for its attributes.

We will now present the most important GBDK APIs which are used in our Pong application in section 5.5:

```
void enable_interrupts();
```

There exist functions to add, enable and disable interrupts. It is basically a good idea to disable all interrupts before adding a new one. One should also disable all interrupts and the display (with the DISPLAY_OFF macro) at the beginning of a game when loading tile data into VRAM.

```
void set_bkg_data(UBYTE first_tile,
                  UBYTE nb_tiles,
                  unsigned char *data);
```

Copies tile data (*nb_tiles* is the amount of tiles to be copied) from the memory location *data* into the background TDT (*first_tile* is the destination index in TDT).

```
void set_bkg_tiles(UBYTE x, UBYTE y,
                   UBYTE w, UBYTE h,
                   unsigned char *tiles);
```

Copies a rectangular area (*w* and *h* are width and height) of tile index numbers (which are referring to the corresponing tiles in the background TDT) from *tiles* into the BTM to the coordinates *x*y*.

```
void set_sprite_data(UBYTE first_tile,
                     UBYTE nb_tiles,
                     unsigned char *data);
```

Copies tile data to the tile TDT analog to *set_bkg_tiles()*.

```
set_sprite_tile(UBYTE nb,
                UBYTE tile);
```

Sets the tile number of a specific sprite in the OAT.

5.5 Case Study: Development of a Pong Clone

We have written a Pong clone with the GBDK for learning and demonstration purpose[12]. The file archive contains two C source code files which can be compiled with the GBDK and run on a Game Boy:

- *thumby.c* is a little demonstration program, which loads a background pattern and a black rectangular sprite. The sprite can be moved with the directional pad.
- *thumby2.c* is the Pong clone itself. A screenshot of it, running on a homebrew cartridge, can be seen in figure 17.

The Pong field is divided into two sections. One contains the two paddles and the ball, the other displays the score. The left paddle is controlled by the human player with the directional pad, the right one is controlled by the CPU. A player wins the game when his score reaches 10 points. When the game begins or a player scores, the game is set to a suspended state. This means, its possible to move the paddle but the ball is frozen in the middle of the field. The game will continue when the START button is pressed
The Pong source code has the following structure:

1. Disable display and interrupts.
2. Load sprites from ROM into VRAM TDT.
3. Enable display and interrupts.
4. Sleep until vertical synchronisation phase.
5. If directional pad was pressed: Change left paddle coordinates.
6. If the game is not suspended: Change ball coordinates.
7. If the game is not suspended: CPU moves the right paddle.
8. If the game is not suspended: Collision detection with ball, paddles and the field boarders. Suspend the game if a player has scored.
9. Update sprite positions on the display.
10. Goto 4.

The collision detection works without trigonometric functions (we have seen in the last sections that it is a good idea to avoid floating point arithmetrics). This is possible because ball and paddle are rectangulars. Therefore its possible to check wether the ball and paddles overlap (collide) with substractions and some *if-else* blocks.

6 Conclusion and further Work

We have given in this work an overview about the Game Boys technical details. Furthermore we have introduced the basics of digital circuits and used them to develope a layout for a homebrew cartridge. We have presented two ways to implement a real homebrew cartridge: modifying an existing cartridge and creating a new one from scratch. The GBDK was used to develope a Pong clone as a proof of concept for a homebrew cartridge.
Nevertheless, some topics were not mentioned and challenging for further work:

- The presented homebrew cartridge did not contain external RAM, a bankswitch controller, etc.
- For more complex projects, it would be necessary to make use of Game Boy assembler programming.
- Modification of existing ROMs to e.g. hide encrypted data in it.
- Place a programmable logic on the homebrew cartridge (e.g. a GAL) and check wether its possible to replace the scrolling Nintendo logo when the Game Boy boots.

7 Acknowledgement

I would like to express my gratitude (in alphabetical order) to Blarz, Kroko, Rembrandt and Xaxes for supporting this work.

List of Figures

References

[1] Pan, GABY, Marat Fayzullin, Pascal Felber, Paul Robson, Martin Korth, kOOPa, and Bowser. Game boy cpu manual, 1999.
`http://belial.blarzwurst.de/gb/GBCPUman.pdf`.

[2] Reiner zieglers page - home made cartridges.
`http://www.reinerziegler.de/readplus.htm`.

[3] Michael Hope. Game boy developers kit.
`http://gbdk.sourceforge.net`.

[4] Jason. CGBdk - how to use cgb features with gbdk, 1999.
`http://belial.blarzwurst.de/gb/cgbdk.txt`.

[5] Michael Hope and Pascal Felber. GBDK libraries documentation, 1998.
`http://belial.blarzwurst.de/gb/gbdk-doc.pdf`.

[6] Manfred Linzner and Jason. Gbdok v1.0, 1999.
`http://belial.blarzwurst.de/gb/gbdok.txt`.

[7] Stephen D. Brown. *Fundamentals of Digital Logic with VHDL Design (McGraw-Hill Series in Electrical and Computer Engineering)*. McGraw-Hill, Inc., New York, NY, USA, 2005.

[8] Paul Halmos. *Lectures on Boolean Algebras*. D. Van Nostrand, Princeton, 1963.

[9] Manfred Seifart and Helmut Beikirch. *Digitale Schaltungen*. Verlag Technik, 1997. in german.

[10] Game boy homebrew cartridge eagle layout.
`http://belial.blarzwurst.de/gbpaper/gb-cartridge-layout.zip`.

[11] International Telecommunication Union. Rec. ITU-R BT.601-4 - encoding parameters of digital television for studios. Technical report, 1994.

[12] Belial. Game boy pong clone.
`http://belial.blarzwurst.de/gbpaper/gb-pong.zip`.

[13] I.N. Bronstein and K.A. Semendjajew. *Taschenbuch der Mathematik, 25*. 1991.