8-Bit Wonderland

Executing custom code on the Nintendo Game Boy

Christian Ammann

November 13, 2015

Contents

1 License

2 Introduction

I wrote this paper for the PH-Neutral 2010 in Berlin. Now, it is 2015. Time for some clean up and a re-release on the Nullsecurity platform. Enjoy :-)

The Nintendo Game Boy was presented in 1989 for the first time and has sold more than 118 million times. It is still possible to buy the original Game Boy which is a cheap 8-bit mobile device. It has no protection mechanisms like modern consoles against the injection of homebrew code. Therefore, it is possible to customize this mobile device and use it for several 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. It can be divided into two major sections:

- Developing a homebrew Game Boy cartridge.
- Developing software for the Game Boy which is executed on the previously build cartridge.

It starts in section 3 with an introduction to the Game Boy and its hardware details. Afterwards, chapter 4 describes how to build a custom cartridge[2]. Custom code for the previously assembled cartridge can be written with the Game Boy Developers Kit (GBDK)[3][4][5][6] using ansi C. We use the GBDK in section 5 to develop the popular Pong game and demonstrate different programming aspects like video signals, avoiding expensive operations like multiplication, etc.

3 Hardware Specification

This section gives a brief introduction to Game Boy basics which are necessary for the development of a home brew cartridge. It starts with a description of the Game Boy hardware. Afterwards, we present its memory layout.

3.1 Features

Nintendo developed several Game Boy versions. This paper covers the *Game Boy Classic* from 1989. It has the following hardware features which are important for this tutorial:

- 8-Bit CPU: Similar to the Z80 processor
- Main RAM: 8K Byte
- Video RAM: 8K Byte
- Clock Speed: 4.194304 MHz
- Games come on special cartridges containing a ROM which holds program code and data. They are controlled with four operation buttons labeled A, B, SELECT, START and a directional pad.
- Battery powered.
- Four colour (greyscale) display.

The Game Boy Classic has also a serial connector called *Dual Link*. It is used to connect two (or more) Game Boys for multi player games. *Dual Link* is not covered by this paper and part of further work.

3.2 Memory Layout

The Game Boy CPU can access a 16 bit address space. It contains the game cartridge ROM, RAM, registers and 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...

According to the listening above, each game is stored on a 32KB external ROM and can access 24KB of RAM. Main RAM is used by games for general purpose aspects. Video RAM controls the display. To avoid shortcomings, Nintendo allows external RAM on a cartridge and and bank switching to extend the ROM size (see section 4.2 for more details).

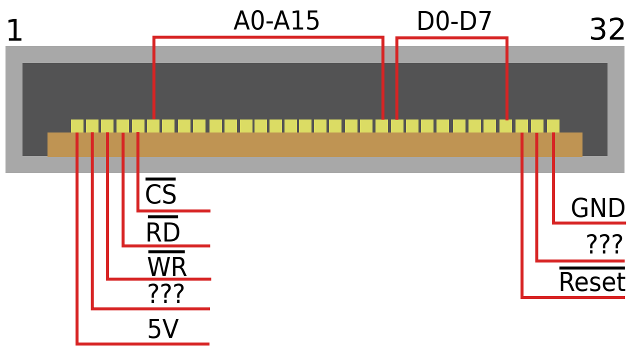

Figure 1: Bottom of a Game Boy Cartridge and its Pin

4 Cartridge Design

Game Boy software comes on special cartridges. Therefore, this chapter describes their layout in more detail and derives from it a simple way to build a custom cartridge.

4.1 Components

Usually, a cartridge consists of the following components:

- EPROM: Read-only memory which contains code and data.
- RAM: Nintendos Game Boy contains 16 Kilobytes of internal RAM. For several games, this is not enough. Therefore, cartridges can contain additional memory.
- Battery: RAM is not persistent. When a Game Boy is turned off, all data is erased or at least in an undefined state. The battery acts as cartridge RAM power supply and allows the persistant storage of save games, etc.
- Bankswitch- and memory controller: Handles bank switching to extend cartridge RAM and ROM size (see section 4.2 for more details).

4.2 Bankswitching

Nintendos Game Boy provides a 16-Bit address bus to access a cartridge. 32KB are reserved for EPROM access. To avoid this limitation, a technique called *Bank Switching* can be used. Bank switching needs additional hardware. Due to this, a special component called *Bank Switch Controller* (BSC) is assembled on a cartridge and located between Game Boy and EPROM. It maps the Game Boy address bus to a larger

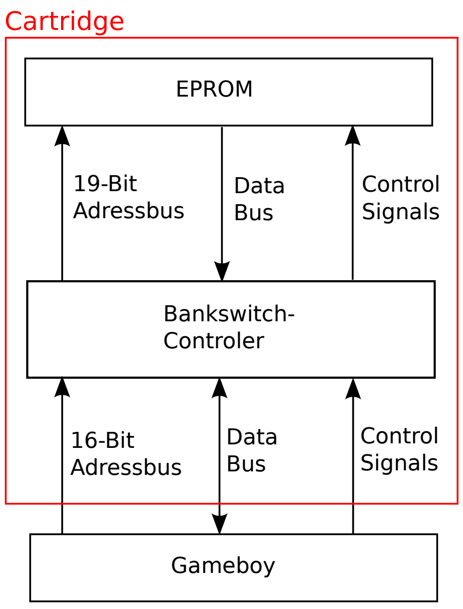

Figure 2: Game Boy connected to an EPROM with a Bankswitch Controller

EPROM address bus.
Figure 2 visualizes this approach. It shows a cartridge with a bank switch controller and its connection to the Game Boy. The EPROM capacity is 512 KB which requires a 19-Bit address bus. The BSC is implemented as a state machine. 512 KB are divided into 16 KB sections called *Banks*. (*Bank 0* : 0x0000 - 0x3FFF, *Bank 1* : 0x4000 - 0x7FFF, etc.) According to its state, the BSC selects a bank which can be accessed by the Game Boy. When the Game Boy is turned on, the BSC is in state s_0. The following table shows the BSC address bus output according to its state and Game Boy address bus input:

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
...	...	...

Address bus values between 0x0000 and 0x3FFF (the lower 16 KB) are just forwarded to the EPROM and access bank 0. Addresses between 0x4000 and 0x7FFF are modified according the the BSC state. This means: Memory between 0x0000 and 0x3FFF is static and always contains bank 0 of the EPROM. Region 0x4000-0x7FFF is dynamic.

It contains bank 0 in state s_0 , bank 1 in state s_1, bank 2 in state s_2, etc.
The BSC has an internal register to control its state. It is mapped to memory address 0x3000. When writing a 0x01 to 0x3000, it switches from state s_0 to s_1. Please keep in mind that different BSC exist for several ROM and RAM types (RAM can be extended with bank switching too).

4.3 Pins

Figure 1 shows the bottom of a cartridge and its connection pins. Besides address- and data bus (A0-A15 and D0-D7), a cartridge provides Vcc, Ground, the control signals WR, RD, CS, Reset and two undocumented pins. The following table describes each pin in more detail:

Pin	Name	Description
01	+5V	Vcc
02	?	–
03	WR	Cartridge ROM is read-only. If a cartridge contains extra RAM, this signal is set to 0 (active low) during write operations
04	RD	Read signal is set to 0 (active low) during read operation on RAM or ROM.
05	CS	Cartridge RAM and ROM share the same data and address bus. “Chip Select” signal can be used to activate RAM and deactivate ROM or vice versa.
06 - 21	A0 - A15	16-bit address bus
22 - 29	D0 - D7	8-bit data bus
30	Reset	Reset cartridge and its components (e.g. the BSC)
31	?	–
32	GND	Ground

4.4 Development of a Homebrew Cartridge

The previous sections describe a cartridge, its components and pin layout. We use these informations to build a custom, homebrew cartridge. Two approaches are possible:

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

This section describes both approaches in more detail.

4.4.1 EPROM

A simple cartridge contains just an EPROM. RAM and BSC are not mandatory. In this paper, we use an EPROM called 27C256 which has a total size of 32 KB (two banks). Therefore, it can be mapped into Game Boys memory without an additional BSC. The 27C257 operates in two modes:

- Normal Mode: Only read operations are possible.
- Programming Mode: Data can be written to the 27C256.

You can write data to the 27C256 in programming mode using a EPROM writer. Writing devices devices for hobbiest can be bought for about 30 euros on several internet market places. It is also possible to delete the content of a 27C256. This is done with ultra violet light rays. Bright sunlight can be used but this would take several years. Therefore, special erasure devices exist which create ultra violet lightrays and delete the content of a 27C256 within minutes. 27C256 EPROMs have the following pin layout:

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

VPP is necessary to activate the EPROMs programming mode. In normal mode, it can be connected to a 5V power supply (e.g. VCC). Chip select (CE) activates a standby mode which turns off the EPROM and consumes less energy (from 20 milli-ampere down to 100 micro-ampere). Output enable (OE) can be used to turn off the EPROMs data bus pins. This is necessary if several components share the same data bus (e.g. EPROM and RAM).

4.4.2 Homebrew Cartridge Layout

This chapter derives from the previous sections a layout for a home brew cartridge. To keep it simple, all we use is a single 27C256. The layout is shown in figure 3 and describes the connections between Game Boy and EPROM pins.
The Game Boys power supply on pin 1 is connected with VCC and VPP of the 27C256. Therefore, 27C256 is running in normal mode. Pin 32 delivers ground to EPROMs VSS input signal. Game Boys *read* signal at pin 4 is connected with CE of our EPROM. Due to this design, the 27C256 is only turned on when the Game Boy wants to read from it. Otherwise, it remains in standby mode to save power. Game Boy and 27C256 data bus

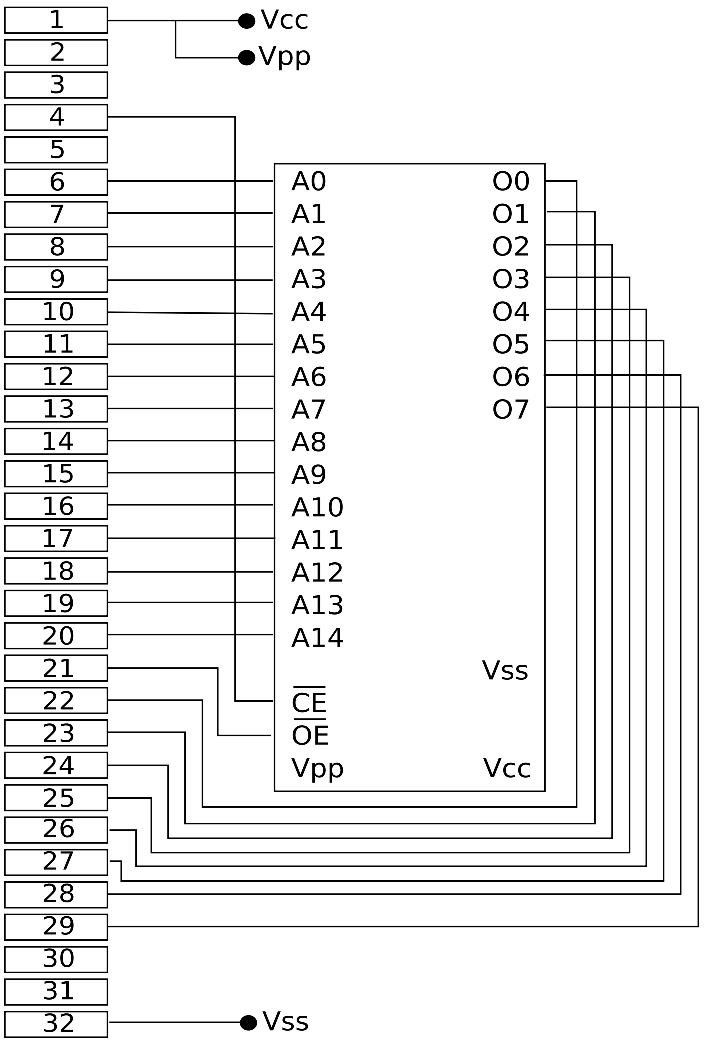

Figure 3: Homebrew Cartridge Layout

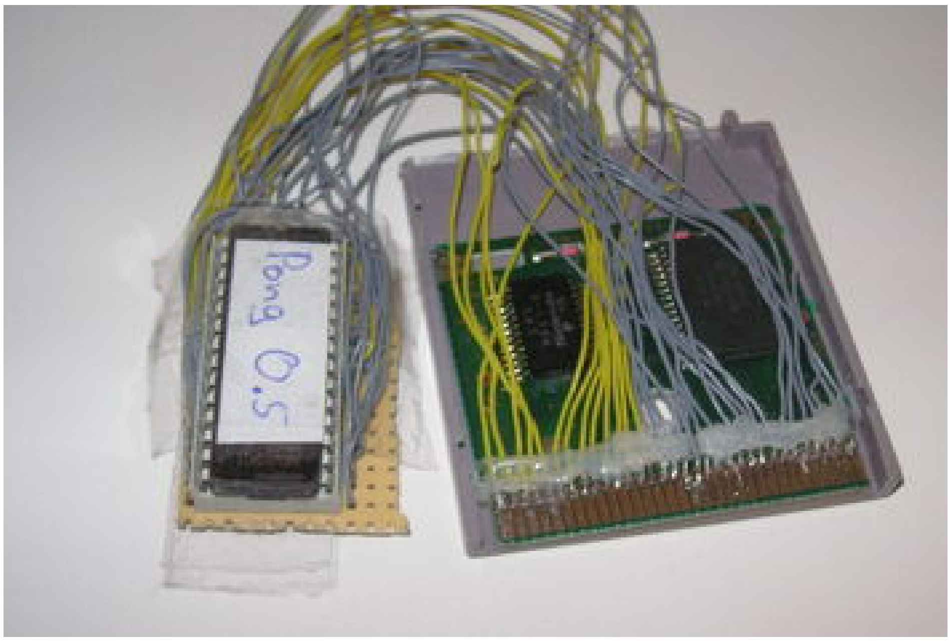

Figure 4: Modified custom Game Boy Cartridge connected to a 27C256

have the same size (8 bit). Therefore, they can be directly connected with eachother. Unfortunatly, this can not be done with address bus pins because the Game Boy has a larger address bus (16 bit) size than the 27C256 (15 bit). We solve this problem the following way:

- The lower 15 address bus pins are directly connected with eachother.
- Game Boys highest pin (16) is connected with EPROMs OE input signal. Due to this, cartridge EPROM is turned off if a game accidently tries to access an invalid memory region larger than 32KB.

4.4.3 Assembling the Cartridge

The previous section describes a cartridge layout. This chapter turns abstract schematics into real hardware. Our first approach does not need a custom PCB. Instead, we modify an existing cartridge (poor paperboy, may it rest in peace;). Figure 4 shows the result. The cartridge pins are connected with an EPROM which contains a homebrew Pong clone (see section 5.5 for details). A saw was used to cut the connections to the originally used EPROM. Furthermore, we assembled at each pin a wire fixed with liquid glue which is connected to the corresponding 27C256 signals (see section 4.4.2 for layout details). 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 custom cir-

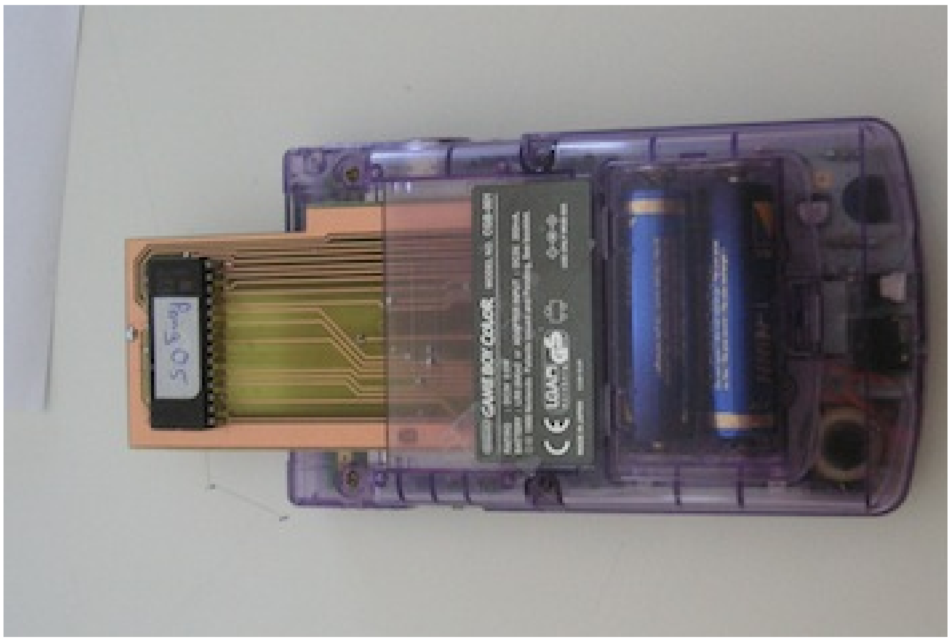

Figure 5: Assembled Homebrew Cartridge

cuit board. We have created a layout for such a cartridge with the Eagle CAD tool[10]. Results are shown in figure 5 and 6.

5 Software Development

This chapter describes the development of Game Boy software, which can be flashed on the previously assembled homebrew cartridge. It is structured the following way:

- Usually, Game Boy development tools generate a ROM file which contains code, data and is written on an EPROM. Section 5.1 describes its layout.
- When a Game Boy is turned on, a boot ROM takes control. The boot sequence is presented in section 5.2.
- The graphical capatibilities of a Game Boy are its most important feature. Therefore, section 5.3 gives an introduction to its data format and timing aspects.
- Section 5.4 presents the Game Boy Development Kit which contains a collection of C compiler, linker, etc.
- Finally, we present in section 5.5 our example application: a Pong clone.

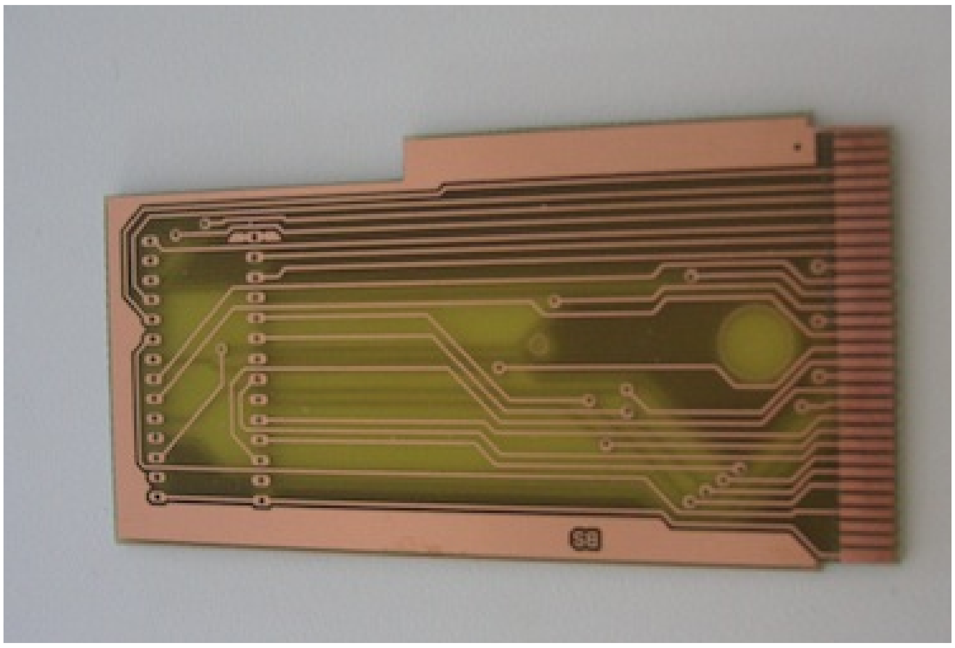
Figure 6: Homebrew Cartridge without EPROM

5.1 Binary Format

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 the header layout in more detail. Therefore, we just give a brief introduction. A ROM file header starts at 0x0000 with a list of pointers (interrupts, etc.). Address 0x0100 is the execution entry point of a Game Boy cartridge. It has a total size of 4 bytes and contains in most cases a jump instruction to the real entry adress. The execution entry point is followed by the Nintendo logo (it is scrolling down when the device is turned on). Afterwards, the ROM header contains 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 section is just a summary of [1]. The boot sequence consists of two steps:

1. A boot program from an internal ROM is executed.
2. Control is passed to the cartridge EPROM.

The internal boot ROM reads the Nintendo logo from the cartridge header and displays it on screen. To be more precisly, it is scrolling down followed by two a musical notes. Afterwards, the Nintendo logo is read a second time from the cartridge header and compared with an internal copy. Execution stops if the logo data on the cartridge differs from the logo data on the internal boot ROM. Otherwise, the internal ROM is disabled and execution passed to address 0x0100. Keep in mind: If you compile custom code, genuine logo data has to be placed in the header or your game will not start.

5.3 Video

This section describes how to access the Game Boy display. Usually, software writes not only video data but also synchronisation signals to a screen. Synchronisation signals are used by the hardware to detect the end of a scanline or a frame. Therefore, this sections starts with an introduction to synchronisation signals using NTSC as an example. Afterwards, we explain how to access the Game Boy video interface and display data on screen.

5.3.1 Synchronisation

NTSC is an analog television 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 [11] was developed in 1941 when no modern LCD display existed. TVs created a video frame with a cathode ray tube. A single electron beam scans a phosphor screen from left to right, line by line and finally returns to the top. The synchronisation signals are necessary to control the TV's electron beam. Two synchronisation signals exist:

- 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 move 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 moved from the lower-right of the display to the upper-left to create the next video frame. It takes some time and is called *vertical synchronisation phase*.
The Game Boy does not contain a cathode ray tube and you do not have to create synchronisation signals. But there is a *vertical synchronisation phase* between two frames. This is important for software development: During a *vertical synchronisation phase*, the Game Boy does not read video RAM to update its screen. Therefore, software should only access video RAM in a *vertical synchronisation phase*. Otherwise, the result can be unexpected behaviour. This leads to the following structure of a simple game:

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

5.3.2 Tiles

The previous section focusses on video synchronisation. Besides timing aspects, the data itself is necessary to display sprites on a screen. Therefore, we give in this chapter an introduction to the Game Boy graphics format. It can be split up into two categories:

- Sprites
- Background

Both consist of one or more *tiles*. A *tile* is a 8x8 image and has a size of 16 bytes. Figure 7 shows an example tile. Each line consists of 8 pixels and each pixel is represented by a greyscale value. The Game Boy classic supports four different greyscale values. Figure 8 visualizes the first line of the previous tile example and its bit-encoding in memory. The bits 00, 01, 10 and 11 are used to model the grey scale value of a pixel. Bit tupels are marked with red squares. According to this, two bits for one pixel are not linear in memory. Instead, the second bit has an offset of 8.
16 byte character C array can be used to model a tile. Graphical tools exist which automate this task. Basically, a user paints *tiles* in a pixel raster and exports them as C character arrays. We have used the *Game Boy Tile Designer* for this paper (GBTD) which can be found at `http://www.devrs.com/gb/hmgd/gbtd.html`.
As mentioned above, game boy graphics consist of backgrounds and sprites. Both are

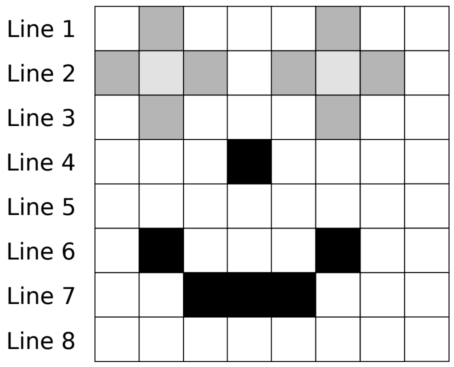

Figure 7: 8x8 Game Boy Tile

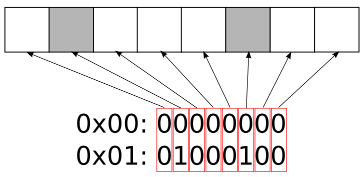

Figure 8: Tile Encoding

represented by *tiles* which are written into video RAM and displayed on screen. Background images can exceed screen size: The video RAM has a 256x256 pixel buffer for background tiles. The Game Boy can display only 160x144 pixels at once but contains *scrolling registers*. The corresponding registers control the background region which is displayed on screen.

5.3.3 VRAM

When a tile is loaded into VRAM, it is displayed on screen after vertical synchronisation phase. The Game Boy VRAM has the following layout:

- 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 as background on screen. The tile numbers refer to the background TDT.
- Object Attribute Table (OAT): Contains the tile numbers which are displayed as sprites on 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.

5.4 Game Boy Developers Kit

Game Boy software can be written using assembler or a high level language. In this paper, we use C. The *GameBoy Developers Kit* (GBDK) [3] provides a C compiler, linker, preprocessor, etc. for the developement of Game Boy ROM files. Before using it, some basic coding guidelines should be understood which are described in subsection 5.4.1. A developer can access the Game Boy hardware using several functions provided by the GBDK. Subsection 5.4.2 presents the corresponding API calls.

5.4.1 Coding Guidelines

Using the GBDK, a developer writes C code and calls APIs to access the hardware. Some general coding guide lines should be kept in mind [3]:

- Initialized global variables are located on the ROM and therefore read-only. Uninitialized global variables are located in RAM and writable.
- The Game Boy CPU is optimized for 8-bit arithmetics. Whenever possible, a developer should use 8-bit unsigned variables. The compiler can handle larger datatypes too, but arithmetic operations will take much more time.
- Avoid multiplications, divisions, modulo and floating point operations. The Game Boys CPU has no native support for them. Instead, the compiler uses slow software emulation.
- The operators $! =$ and $==$ are more efficient than $<=$, $<$, etc.

As mentioned above, floating point arithmetics or complexer mathematical functions (e.g. sin(x), tan(x), etc.) are slow and should be avoided. If a developer still needs them, the generation of lookup tables can be a faster alternative:

Listing 1: Example for Sinus Lookup Table Generation

```
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 *Java* example which creates a C *float* array. It contains $sin(x)$ values and can be included in a Game Boy C project. A game receives sinus values with a simple array read operation. The disadvantage is an increased ROM size and reduced accuracy (the generated array has a granularity of 0.1).
It is also possible to speed up multiplications with lookup tables. To do so, we propse the following algorithm [13]:
$log_a(x * y) = log_a(x) + log_a(y)$
which leads to:
$x * y = log_a^{-1}(log_a(x) + log_a(y))$
A multiplication is mapped to an addition, $log_a(x)$ and $log_a^{-1}(x)$. The advantage of this approach: The Game Boy Z80 CPU has native (and therefore fast) support for additions. As described above, the computation of $log_a(x)$ and $log_a^{-1}(x)$ can be implemented with lookup tables. A developer can use these lookup tables to perform divisions too ($\frac{x}{y} = log_a^{-1}(log_a(x) - log_a(y))$). See [13] for more details about the logarithm function.

5.4.2 APIs

The GBDK contains APIs implemented as macros and function calls which control Game Boy behaviour. A detailled list can be found at [5]. This section gives a brief overview focussing on APIs used in the Pong case study:

```
void enable_interrupts();
void disable_interrupts();
```

The GBDK has functions to add, enable and disable interrupts. It is basically a good idea to disable all interrupts before adding a new one. Furthermore, a developer should disable interrupts at the beginning of a game when loading tile data into VRAM.

```
DISPLAY_OFF;
DISPLAY_ON;
SHOW_BKG;
SHOW_SPRITES;
```

The listing above shows macros to control the display. We use them in our pong clone to disable the display before the main loop while writing data into VRAM. VRAM access consists basically of two steps:

- Load tile data from ROM into background or sprite TDT.
- Update BTM and OAT to specify which tiles are painted on screen at certain coordinates.

The following two listings demonstrate background TDT and the corresponding BTM functionality:

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

The method above copies tile data from the memory location *data* into the background TDT. *first_tile* refers to the TDT destination index. *nb_tiles* represents the amount of tiles to be copied.

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

Set_bkg_tiles() copies a rectangular tile area into the BTM which is shown on displace after the vertical synchronisation phase. *W* and *h* represent width and height of the tile area. The background area coordinates on screen are specified by *x* and *y*. *Tiles* is a pointer to a tile index number array. Its size is derived from $w * h$.
Besides background tiles, the Game Boy supports also sprite data. Therefore, the following listings demonstrate sprite VRAM access.

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

The method *set_sprite_data()* copies tile data from the memory location *data* into the sprite TDT. *first_tile* refers to the TDT destination index. *nb_tiles* represents the amount of tiles to be copied. *first_tile* refers to the TDT destination index. *nb_tiles* represents the amount of tiles to be copied.

```
set_sprite_tile(UBYTE oat_index,
                UBYTE tdt_index);
move_sprite(UBYTE oat_index ,UBYTE x, UBYTE y);
```

The OAT consists of several blocks (see section 5.3.3 for details). The method *set_sprite_tile()* writes the tile index number *tdt_index* into the block *oat_index*. Afterwards, *move_sprite()* sets the x and y coordinates on screen.

5.5 Pong Clone Case Study

We have written a Pong clone using the GBDK for demonstration purpose. The project [12] contains two C source code files which can be compiled with the GBDK and executed on a Game Boy with the previously assembled cartridge:

- *thumby.c* is a 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.

The Pong playfield is divided into two sections. The first one contains the two paddles and the ball, the second one displays the game score. The left paddle is controlled by a 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. The game begins 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. Update ball coordinates.
7. CPU moves the right paddle.
8. 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 we just check wether the ball and paddles coordinates overlap (collide) using substractions and *if-else* blocks.

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.