

Game Boy Assembly Programming for the Modern Game Developer

Martin Ahrnbom

October 16, 2020

This book and its source code is released under CC BY-NC-SA 4.0. You can read more about it here: <https://creativecommons.org/licenses/by-nc-sa/4.0/>.

The book is available in source code here: <https://github.com/ahrnбом/gbapfomgd>.

The book is available for download for free here: <https://github.com/ahrnбом/gbapfomgd/releases>.

Contents

1	About this book	5
1.1	Who is it for?	5
1.2	Why write this?	5
1.3	Assembler vs C	7
2	Basics of Game Boy Assembly	9
2.1	Brief description of the hardware	9
2.2	Introduction to the CPU	10
2.3	The most common commands	14
2.4	Programming structure	18
2.5	Loops	20
2.6	Optimization	21
2.7	Compile time operations	22
2.8	Defining data	26
3	Making Game Boy games	29
3.1	About RGBDS	29
3.2	The GingerBread library	30
3.3	Memory mapping	30
3.4	Sections	31
3.5	The starting point	32
3.6	The header	32
3.7	Interrupts	33
3.8	RAM	34
3.9	Graphics: backgrounds	35
3.10	Graphics: sprites	38

3.11	Reading user input	39
3.12	Audio: Sound effects	40
4	More involved topics	45
4.1	Memory map	45
4.2	ROM banks, bank switching	46
4.3	Graphics: text and numbers	47
4.4	Audio: music	51
4.5	Save data	52
5	Super Game Boy Features	53
5.1	Setting up SGB functionality	54
5.2	Palettes	55
5.3	Border image	58
6	Game Boy Color features	61
7	Practical aspects	65
7.1	Debugging in BGB	65
7.2	Emulator versus real hardware	65
8	Final notes	67
9	Version history	69

Chapter 1

About this book

What is the purpose of this book? This book attempts to explain the entire process of making homebrew Game Boy games, from idea to finished ROM file. It explains how Game Boy's assembler programming works in a way that is hopefully understandable for modern programmers.

1.1 Who is it for?

This book assumes that the reader is familiar with game development on some other platform. It also assumes experience with some modern programming language. When comparing assembler code to "modern" code, the latter will be written in Python, so basic understanding of Python's syntax helps. Such examples will however be quite few.

1.2 Why write this?

There is plenty of documentation on how to make Game Boy games, including Nintendo's own programming manual (available here <https://archive.org/download/GameBoyProgManVer1.1/GameBoyProgManVer1.1.pdf>), and the Pan Docs (available here <https://gbdev.io/pandocs/>). Such sources however expect the reader to already be familiar with 80s style assembler programming. There are also guides that try to explain the basics of assembler programming, without going into sufficient detail to be allow the user to make full games.

This book tries to explain things detailed enough to get people started in making actual games, but without too many details to bore the reader. It is not intended as a complete documentation, and it only briefly mentions many things that I do not consider necessary to make games. For example, things that are implemented in the GingerBread library (which was made alongside this book and used many times) are usually not explained in further detail than how to use the functionality in the library. Once the reader is finished with this book, they should have the skills and understanding to look for more information on topics that might interest them. Because of this, I sincerely believe this book wastes the reader's time as little as possible.

There are two main reasons for writing this book.

The first is to help people who want to make Game Boy games. I had dreams of making my own Game Boy games when I was six years old, and 20 years later I made that dream a reality. By writing this, I hope I can help others do the same for themselves.

The second reason is for preservation purposes. The art and craft of programming assembler is becoming increasingly rare as it is not often required for modern programming tasks. By keeping this knowledge alive, we can better understand and appreciate the games made for the Game Boy, by disassembling their code and understanding it directly, or by making games of our own and comparing with commercial releases from that era.

The Game Boy stands out among other retro consoles as an attractive platform to develop for. Many people have Game Boys that still work, thanks to their build quality. It is possible to obtain relatively cheap reproduction cartridges that can be reflashed, allowing for a distribution of the game that works on real hardware. And the Game Boy is an iconic and nostalgic piece of gaming history.

I do not know every detail about Game Boy development. I have made a complete Game Boy game, and this book is based on the experience and knowledge I gathered when making it. It is my opinion that the currently existing literature doesn't explain the topic well enough (either leaving out far too much, or by including far too many unnecessary details and assuming too much previous knowledge) that I believe that this book should be useful to many, even if there might be some errors, or some important things left out. If you have any suggestions for improvements, please contact me at mail@teamlampoil.se.

There is a project called "Awesome Gameboy Development", available at <https://github.com/gbdev/awesome-gbdev> which tries to gather as much

resources as possible related to Game Boy development, emulators etc. If you are looking for something that you cannot find in this book, chances are you can find it there.

1.3 Assembler vs C

When making Game Boy games today, a developer has two main options: either one can write the game in ASM, or in C. While C might seem easier, in that the GBDK compiler takes care of a lot of "gotchas" that you have to worry about yourself in ASM, and in C you'll much more quickly get a simple working ROM to start with. But as soon as one starts going beyond that, the weakness of C becomes obvious, especially from a pedagogical standpoint. The C language is designed from the ground up to hide processor implementation details, to allow a single piece of code to work on multiple processors. You still need extensive knowledge of the Game Boy CPU's limitations, but coding in C does nothing to inform the programmer of those limitations. Sooner or later, one will write a line of code that would work perfectly on any modern computers, but it simply cannot run on the Game Boy's limited CPU (or, at the very least, not fast enough), and the compiler won't give any warnings or errors. This is extremely counterintuitive, but unavoidable when coding in C. The C language is actively trying to hide information that you need, making it your enemy instead of your ally (at least from a pedagogical perspective). Furthermore, emulators support ASM debugging, but not C, and it is impossible or difficult to use existing (compiled) games' code as references without knowledge in ASM.

Even if one has decided to write a game in C, some knowledge of ASM is still very helpful. Perhaps this book can be of use in such cases as well.

Another argument for using ASM is that it's more authentic in the sense that it's the way games were made back in the day. Knowing ASM helps preserving a part of gaming history that would otherwise risk getting lost in time.

The downside of using ASM is mainly that the counterintuitive syntax tends to result in less readable code.

In this book, the Rednex Game Boy Development System (RGBDS) is used for all ASM syntax and examples. It is available as open source at this link and works on most modern computers: <https://github.com/rednex/rgbds> It creates Game Boy ROM files that can be played either on an emulator

or on real Game Boy hardware (given the right equipment).

Chapter 2

Basics of Game Boy Assembly

2.1 Brief description of the hardware

The Game Boy CPU is an 8-bit Sharp LR35902. It is quite similar to the z80 processor which was common in the 1980s, but it's not identical. The z80 has some more advanced instructions, so if you see z80 ASM code you want to use for your game, it might need some modifications. The Game Boy CPU runs at around 4 MHz (or 8 MHz on the Gameboy Color's high speed mode). This is incredibly slow by today's standards, and the CPU has such a limited amount of supported operations that it's quite easy to know most of them by heart, which makes it relatively easy to program for, at least in ASM.

The Game Boy has 8 kB of RAM, and additional RAM may be available inside certain game carts. There's also 8 kB of dedicated VRAM for storing graphics.

The screen has a resolution of 160x144, supporting 4 colors (darkest, dark, light, lightest) on the original Gameboy and the Game Boy Pocket and Light, or many more colors on the Super Game Boy and Game Boy Color (although the way these are colored is completely different, see Chapters 5 and 6).

The graphics are made out of background tiles and foreground sprites, letting the sprites move arbitrarily around over the background. The Game Boy actually renders a 256x256 pixel surface, and then a 160x144 section is

cropped from that and displayed on the screen. Moving this viewport around is used for creating scrolling effects.

There are significant limitations to the graphics, like the number of sprites that can be displayed at once on screen. Many games, especially early ones, have flickering sprites as a result of trying to display more than possible on screen.

The Game Boy has limited sound capabilities, but can still produce pleasant music and sounds if used correctly. There are four audio channels: two square wave channels, a white noise channel and a customizable wave shape channel. The latter can be used to produce sampled audio, but this is used sparsely because of the high CPU usage and storage requirements.

The Game Boy has eight inputs: four directions on the D-pad, and A, B, Start and Select buttons.

2.2 Introduction to the CPU

One of the most important core concepts in ASM programming is registers. A register is a small memory storage inside the CPU, and in the case of the Game Boy CPU, each register hold one 8-bit number; an integer in the range of 0-255. The operations that the CPU can perform work on the numerical data currently stored in those registers, and the results of the computations are also stored in registers afterwards.

There is one "main" register which is called A. The most "complicated" operations that the CPU supports, like addition and subtraction, work only on the A register. Other registers, like B, C, D, H and L (more on them and their names later) support only simpler operations like incrementing and decrementing (addition and subtraction by one) and nothing more complex than that. Therefore, those registers are used more for temporary storage of data, while the A register contains whatever data you are currently working with, in some sense.

One of the main differences in how you think about programming in ASM compared to modern languages is that in ASM, you need to care a lot about where data is, and less about worrying about what kind of data it is. This is easily shown by example. Let's consider the following python function, which does the simple job of subtracting 12 from an integer number.

```
def subtract_by_12(x):  
    return x-12
```

It can then be called like this:

```
num_anteaters = 62
num_anteaters = subtract_by_12(num_anteaters)
# now the number of anteaters is 50
```

Here, we don't care at all about where the data (`num_anteaters` and `x`) is stored, but we care about what kind of data it is (it needs to be some kind of number to support subtraction).

Before we can try to translate this extremely simple function to ASM, we need to think about position. The input value `x`, where is it? A common design when a function takes a single input value, is that the input is assumed to be on the A register (the "main" register). The output of the function is similarly also commonly placed on the A register whenever the function only outputs a single number. If we follow these conventions, the equivalent ASM code would be

```
; input and output on the A register
subtract_by_12:
    sub 12
    ret
```

Which can be called by

```
ld a, 62 ; store the number of anteaters in A
call subtract_by_12
; There are now 50 anteaters, as A now holds 50
```

As one might guess, `sub` means subtraction and `ret` means return. Notice how similar this is to python! The main difference is that there is no `x` anywhere, because the `sub` operation only works on the A register, so it is implied that we work on the A register. `sub` writes its output to the A register, so there is no need to specify a return value; the calling code can simply read the A register after calling this function to access the "return value". Because the code itself doesn't explicitly state where the input and output are, it's a good idea to write this as a comment at the top of a function (comments start with a semicolon `;` in ASM).

If we would, for whatever reason, want the input and output to be on some other place, like the B register, we would need a slightly more complicated function. In modern code, you would expect to be able to just replace A with B in all places but that doesn't work here, because the `sub` function only works on the A register. So you **CANNOT** do something like

```
subtract_b_by_12:
    sub b, 12 ; doesn't work, the CPU has no such command
    ret
```

Instead you have to do something like

```
subtract_b_by_12:
    ; Input and output on B. Destroys any existing data on A.
    ld a, b
    sub 12
    ld b, a
    ret
```

Which is called like

```
ld b, 62 ; Store the number of anteaters in B
call subtract_b_by_12
; Now the number of anteaters is 50, because B holds 50
```

What this function does is to copy (or load, which is what `ld` stands for) the value in the B register to the A register, then perform the subtraction, and then copy the value back to B. The reason for this is, as mentioned, that "complex" operations like subtraction only work on the A register. As you see, we have to think a lot about where data is, as different operations work on different positions of data. We do not, however, have to worry about the kind of data we're dealing with, since the CPU really only supports 8-bit integers, so it's implicitly assumed for all commands used in this piece of code that that's what we're working with. The CPU also support 16-bit integers for some operations, but data types never really get more complicated than that.

Because we are using the A register in the code above, any pre-existing data there will be overwritten. This is also good to mention in a comment, to make sure the caller is aware of this. If such behaviour is unwanted, it can be fixed (see Chapter 2.3, regarding the `push` and `pop` commands).

What if we want the number we're working with to be stored in RAM instead? ASM doesn't have automatic memory management like Python, so we need to know where in RAM the number should be stored, via a memory address. A memory address is a 16-bit number, and each possible number refers to a single slot in memory, and each such slot holds a single 8-bit number.

Each register holds a single 8-bit integer, so to store a memory address which is 16-bit, we need two registers. When two registers are used as a pair,

their values are treated like a single 16-bit number and the Game Boy CPU has some basic operations that work with 16-bit numbers this way, like for reading from memory from a given address stored in two registers. It should be stressed that the Game Boy's CPU doesn't really operate on 16-bit numbers, as only very simple operations work on them (that's why it's considered an 8-bit CPU) but things like storing a memory address in two registers does work.

For operations using 16-bit numbers, two registers are used as a pair. Only certain pairs are allowed: BC, DE, HL and AF (the F register is special, see Chapter 2.4). One such operation that works with 16-bit numbers is copying (loading) data from a memory address (to access that stored in RAM or on the game ROM). If we want to make the same function again to subtract 12 from a number, but this time we want the input number to be defined by a memory address, we again need to think about where this memory address would be. A common practice is to provide a single 16-bit input on the H and L registers, so let's go with that. The output should be written to the same memory address so we don't have to specify an output register.

```
subtract_by_12:
    ; Input and output on RAM, by address in HL
    ld a, [hl] ; reads memory from address specified
    ; by the numbers in HL registers

    sub 12
    ld [hl], a ; write output to RAM
    ret
```

Which is called like so, assuming that NUM_ANTEATERS is a constant 16-bit number which is a memory address on RAM where we store the number of anteaters:

```
; There are 62 anteaters,
; so the number 62 is stored at address NUM_ANTEATERS in RAM
ld hl, NUM_ANTEATERS ; Stores the memory address
; in the H and L registers

call subtract_by_12
; The number of anteaters in RAM is now 50
```

What this does is to load data from the memory address defined in the HL registers and store it in A. Then 12 is subtracted, and the value is “returned”,

by writing it to the same memory address.

To summarize:

- In ASM, we worry more about where data is, and less about what kind of data it is
- There's a kind of memory called registers, that hold numbers that the CPU operates on
- The most complex operations work only with the A register, leading to moving data back and forth
- 16-bit numbers are supported by working with pairs of registers. Can be used for storing memory addresses.

2.3 The most common commands

The Game Boy CPU is very limited by today's standards, so there are quite few operations it can do. Therefore, it's quite easy to learn the most important ones by heart. Once you do that, it should become significantly easier to understand how to write ASM code, since that problem boils down to expressing your intent in terms of those operations.

Probably the most common command is `ld`, which stands for "load". It copies integer data from one place to another. It can copy between registers, for example `ld e, b` which copies data from the B register to the E register. It can load constant values, like `ld c, 17` which writes the number 17 to the C register. You could equivalently write `ld c, $11` or `ld c, %00010001` in hex and binary, respectively (hex numbers always start with a dollar sign, and binary numbers always start with a percent sign; `11`, `$11` and `%11` mean 11, 17 and 3 respectively).

`ld` supports 16-bit numbers as well, allowing you to load a number onto a pair of registers. As briefly mentioned, only certain pairs of registers can be used this way: AF, BC, DE and HL. For example, `ld bc, $12AB` writes the number 4779 to the two registers B and C. This means that B would hold `$12` (which is 18 in decimal) and C would hold `$AB` (which is 171 in decimal). Basically, the first 8 bits go into the first register and the last 8 bits go into the last register. You cannot however copy data from a pair of registers to another pair, like `ld bc, de` but the stack can be used for this purpose (see below).

In addition, `ld` can load from memory addresses using special syntax: `ld [de], a` would copy the data from A onto a memory position specified by the address stored on DE. This does not work for other registers; for example, `ld [de], b` does not work. The same goes when the memory address is stored in BC. If the memory address is, however, stored in HL, you have more options for loading the data: `ld [hl], b` and `ld b, [hl]` work, and you can replace `b` with any single register.

A hardcoded address can be used as well, like `ld [$ff83], a` which writes the data stored in A to memory position `$FF83`. Similarly, data can be read from memory positions, for example by `ld a, [$ff83]`. For reading and writing into hardcoded memory addresses, the data must be written to and read from the A register, respectively.

A few things you **CANNOT** do:

```
ld b, $1A2B ; a 16-bit number doesn't fit in a single register
ld ab, $1A2B ; AB is not a valid register pair
ld 5, a ; copying goes from right to left,
; cannot write to a constant
```

```
ld [hl], bc ; a memory address points to a single position
; holding a single 8-bit number (BC is 16-bit)
```

```
ld [h], 5 ; H register holds a 8-bit value,
; but addresses must be 16-bit
```

```
ld hl, bc ; copying 16-bit numbers from
; one register pair to another is not supported
```

```
ld [$1234], 255 ; Reading/writing to memory address only works
; to/from A, not some other value like 255
```

A final note on the `ld` operation: When reading and writing from memory addresses, it is common to read/write many numbers in a row. For such cases, special syntax allows you to increase or decrease the memory address after reading/writing, in the same CPU command, for example `ld [hl+], a` writes the content of A to whatever address HL is holding, and then increases HL by one (as a single 16-bit number), and `ld a, [hl-]` reads from the address specified in HL, stores the value in A, and finally decreases the value of HL by one.

Two other common commands are `inc` and `dec`, which increment (increase

by one) and decrement (decrease by one) the value of registers, respectively. These are very flexible, and work on all the commonly used registers (A, B, C, D, E, H and L) as well as 16-bit register pairs (BC, DE and HL). The syntax is intuitive: `inc b` increments the 8-bit number in B while `dec de` would decrement the 16-bit number in DE.

There's also `add` and `sub`, which adds and subtracts arbitrary amounts to/from the A register only (for example, `add b` means that the number in A is increased by B). In the case of `add`, 16-bit addition is supported but only to the HL register pair (for example `add hl, bc` increases the 16-bit number on HL by the 16-bit number stored in BC). 16-bit subtraction is not supported, and it is also not supported to add/subtract an 8-bit number to/from a 16-bit number.

The Game Boy CPU also supports bitwise operations (AND, OR, XOR), all of which only work on the A register. These are applied bit-by-bit, with arguments being either another register or an 8-bit constant. For example, if A holds `%10101010` and B holds `%11110000`, then `and b` would produce `%10100000` in A, or `b` would produce `%11110101` in A, and `xor b` would produce `%01011010` in A. Another example is `xor a` which always sets A to 0, regardless of previous values. This code is often preferred over the more readable `ld a, 0` for optimization reasons (see Chapter 2.6).

The `cpl` command computes the binary complement of A (flips all bits, so that `%11001010` becomes `%00110101`). This command only works on the A register.

The "swap" command swaps the top four bits with the bottom four bits of any register (so `%11001010` becomes `%10101100`).

There are operations for so-called rotations and shifts. The operations `rlc` and `rrc` rotates any register to the left and right respectively, which means that each bit is moved one step left/right, looping around to the other side. For example, if the B register holds `%11001010` before, then after a `rlc b` it will hold `%10010101`, but if we ran `rrc b` instead, it would hold `%01100101`. Note that special, faster commands exist for rotating the A register: `rlca` and `rrca`. These work exactly the same, except they only work on the A register, and execute twice as fast, and take up half the ROM space (see Chapter 2.6).

Shifts are similar to rotations, but do not "wrap around" and instead just pad with zeros. The commonly used commands are `sla` for shifting left, and `srl` for shifting right. For example, if B holds `%11000011` before, then running `sla b` would result in `%10000110` while `srl b` would result in `%01100001`. Note that the shifting operations (sometimes called bit-shifts) are equivalent to

multiplying/dividing numbers by two (rounding down, in the case of division).

Several CPU commands can be executed directly on memory addresses, without first loading the data into a register. For this to work, the memory address needs to be stored in the HL registers. Some examples include `swap [hl]`, `sla [hl]`, `sra [hl]`, `inc [hl]` and `dec [hl]`.

There exist a command called `cp` which is used to compare numbers, to see if they're equal or one is bigger than the other. More on this in Chapter 2.4.

The command `halt` puts the CPU to sleep for a while, to save battery life and can be used to keep the timing of the game. The game sleeps until the CPU reaches an interrupt (see Chapter 3.7). If only VBlank interrupts are used, this means that the CPU sleeps until a 60 Hz timer activates.

The command `nop` does nothing (no operation). Because of a hardware bug, the command `halt` should always be followed by a `nop`.

There exists a built-in data structure inside the Game Boy, called the stack. Unlike "the stack" in languages like C/C++, it is used like an actual stack where you can typically only push data onto the stack and pop the top value, with no easy way to access any data except at the very top. A "pointer" (actually two special registers called SP, for "stack pointer") keep track of where the top of the stack is. The stack stores 16-bit numbers and is therefore often used to store addresses (more on this in Chapter 2.4) but it can also be used to temporarily store arbitrary data. The operations for this are `push` and `pop`, and they use register pairs as their arguments. For example, if we want to implement the `subtract_b_by_12` function from Chapter 2.2 so that the original value in A is preserved, we can implement it like so:

```
subtract_b_by_12:
    ; Does not affect A. Input and output on B.
    push af ; places the 16-bit number from AF onto the stack
    ld a, b
    sub 12
    ld b, a
    pop af ; takes the same 16-bit number
    ; back from the stack onto AF again,
    ; thus A is restored to its initial value
    ret
```

Another use for the stack is to move 16-bit numbers between register pairs.

```
push bc
pop de
```

Is equivalent to

```
ld d, b
ld e, c
```

(The latter is preferred from an optimization standpoint, as it executes faster).

It is important that every `push` is followed by a `pop` at some point, to avoid stack overflows, and messing with the Game Boy's program flow (see Chapter 2.4). It should be noted that when values are pushed to the stack, they are copied and thus remain on the registers they come from.

2.4 Programming structure

As mentioned, it's possible to write functions in ASM, by writing a label, followed by a colon, and then the indented code. Such functions can be called using the command `call`, and a special command `ret` returns the execution to wherever it was when the `call` happened, to the line after the `call`. The Game Boy keeps track of where to return to by storing an address onto the stack, so every time you run `call`, the address of the next line to execute (a 16-bit number) is pushed to the stack, then the execution is moved to wherever the `call` command pointed, and when a `ret` command is reached, the stack is popped and execution is moved to the location stored on the stack. Because of this, you **CANNOT** do something like the following:

```
push bc ; save this data for later
call some_function
; do more stuff
...

some_function:
    pop bc ; retrieve the saved data
    ; do stuff...
    ret
```

Because the `pop bc` will actually pop the execution address to BC, and when the code reaches the `ret` it will interpret whatever data was stored initially as an address and try to execute code there, which will result in a crash or unexpected behaviour.

Another way to control program flow, is through the commands `jp` and `jr`. These are often called `goto` in modern programming terminology, and are

used to jump from one place in code to another. Unlike `call` and `ret`, the stack is not used to keep track of where you come from.

When you jump using the commands `jp` and `jr`, the position you jump to should be specified by a label, just like with `call`. ASM does not require that every label is connected to one or more `ret` commands (like a function in modern programming languages would be) so you can create labels anywhere you want in your code, and jump to that location. For example, the code

```
Start:
    ld a, 3
    jr End
Middle:
    add 2
End:
    add 5
```

Will result in A holding the number 8, because the middle is skipped. If the line `jr End` was removed, then A would hold 10, as code execution simply moves from one label to the next unless you tell it otherwise. The labels are simply names given to the next line in code.

The difference between `jp` and `jr` is that the latter is a relative jump, meaning that you cannot jump further than 127 positions away (since how far you jump is internally stored as an 8-bit signed integer). From an optimization standpoint, `jr` is preferred whenever possible.

All of the operations `call`, `ret`, `jp` and `jr` support conditional arguments. This allows something equivalent to simple "if-statements". The syntax is as follows (in the case of `jp`):

```
jp z, some_position_in_code
jp nz, some_position_in_code
jp c, some_position_in_code
jp nc, some_position_in_code
```

The `z`, `nz`, `c` and `nc` decide the condition to be fulfilled for the jump command to work, and those conditions are based on the state of the special F register (the one that is paired with the A register in 16-bit commands). The F register does not allow you to write data to it directly, instead it stores some information about previous commands' results. Each bit in the F register has special meaning and these bits are called "flags". The two important bits are called `z` and `c` (the latter should not be confused with the C register!). The `z`-flag, which stands for "zero", will be set (meaning it holds a value of 1) if

the result of the previous computation was exactly zero, and unset (meaning it holds a value of 0) if the last operation had some non-zero result. The c-flag, which stands for "carry", will be set if the last operation had an "overflow", meaning that the result was larger than 255 or less than 0 (in which case the number simply rolls over). For example, if A holds 200, and you run `add 100`, the result in A will be 45 and the c-flag will be set, because the result ($200+100=300$, which is larger than 255, and since the 8-bit register can only hold numbers between 0 and 255, when it reaches 256 it will instead have 0, and continue adding the remaining 45 onto that). The z-flag will be unset, because the result was not exactly zero. Another example is if A had the value 10, and we execute `sub 15`, then the result in A would be 250 and the z-flag would be unset and the c-flag would be set. If A had the value 10, and we run `sub 10`, then the result would be 0 in A, and the z-flag would be set, and the c-flag would be unset. The `cp` command works the same as `sub`, except that it doesn't actually save the result in A, allowing it to be used to compare numbers: the z-flag will be set if the numbers are identical, while the c-flag will be set if the argument is greater than the number in A. For example, the following code calls `do_something` if and only if A holds the number 2:

```
cp 2
call z, do_something
```

If you want to call `do_something` if and only if A holds any number except for 2, do

```
cp 2
call nz, do_something
```

Where `nz` means "not z-flag", that is that the z-flag is unset. Similarly, `nc` can be used for checking if the c-flag is unset.

2.5 Loops

The Game Boy doesn't have any real built-in operations for loops, but they're fairly easy to program yourself. If the number of iterations is known in advance, this number should be stored somewhere, typically in the B register, and every iteration the number is decremented, and the z-flag is checked to see if the number reached zero, in which case the loop should end.

For example, let's say we want to have a function that multiplies a number by 11. Let's assume, for simplicity, that the input number is small enough so

that the end result will fit in an 8-bit number and not have to worry about overflows. A simple loop implementation, which simply adds the number to itself eleven times, is as follows:

```
mult_by_eleven:
    ; Input and output on A

    push bc ; lets us use B and C registers
    ; without messing up existing data,
    ; as it is restored before we return

    ld b, 10 ; the number of times we loop
    ; Since input is already on A, we only
    ; need to add it 10 times
    ld c, a ; store the input in C

.loop:
    add c ; add input onto A
    dec b ; decrease loop counter
    jr nz, .loop ; if didn't reach zero, go back

    ; If we got here, the loop is finished
    pop bc ; restore whatever were on BC before
    ret
```

Notice how the label `.loop` starts with a dot. This indicates that the label is local, in the sense that it is only valid until the next line where a new global label appears (a label without a dot at the start). This allows every function to have its own sublabel called `loop` so that you don't have to come up with unique names for every time you want to make a loop, for example.

Another thing to note is that if we remove the line `ld b, 10` then we have made a function for multiplying arbitrary numbers, with input on A and B. Such a function could actually be useful, although it should be noted that the implementation isn't always efficient (see Chapter 2.6).

2.6 Optimization

This section will not fully explore all aspects of Game Boy code optimization, which is a complicated topic, but instead give a basic understanding of what aspects code can be optimized in, and how it differs from modern program-

ming. On the Game Boy, the primary bottleneck is rarely CPU execution time or RAM usage. Instead, a common issue is that code takes up too much space on the ROM. All ASM code is divided into sections, and these are stored in divisions of the ROM called banks. Since each bank has a finite size, there's also a limit to how large the sections can be. If sections get too big, they might need to be split up into smaller sections and/or be placed in different banks. It is not quite trivial to call or jump to code from one bank inside code in another bank (how to do this will be covered in Chapter 4.2) so keeping the code small in size saves some work. In addition, the hardware cartridges that one could write the game onto also has a finite size, further motivating keeping the code as small as possible. This is in strong contrast to modern game development, where the final size of the resulting binary as a result of changes in the code itself (as opposed to graphical and audio resources) is rarely a concern.

Often, code that takes up less space also executes quickly. One example is the `jr` operation which is both more compact and executes faster than `jp`. More examples will be seen in the next chapter.

2.7 Compile time operations

RGBDS supports symbols and macros, which allow the programmer to avoid repeating code, and in the best case can make code more readable. The simplest symbol is probably the `EQU` command, which allows the creation of compile-time constants. This is useful for, among other things, not having to remember memory addresses. For example, in this example back in Chapter 2.2:

```
ld hl, NUM_ANTEATERS ; Stores the memory address in HL registers
call subtract_by_12
```

In order for the compiler to know which address `NUM_ANTEATERS` is, it needs to be defined somewhere above that line. This is done by writing something like

```
NUM_ANTEATERS EQU $FF13
```

Where `$FF13` is a position in the Game Boy's RAM. It is your job to make sure you don't use the same memory location for anything else (unless you know what you're doing), but at least you only have to define the position once, and can just refer to it as `NUM_ANTEATERS` anywhere else in the code. It

also means that if you later decide to change the position, you only have to change a single line.

EQU works very similarly to EQU but it works with strings (text). See Chapter 4.3.

A similar symbol is SET, which works exactly the same as EQU except that multiple values can be set to it, at different times. It can be thought of as a compile-time variable. For example, the code

```
xor a ; makes A hold 0
X SET 5
add X
X SET 7
add X
X SET 3
add X
```

would compile to

```
xor a
add 5
add 7
add 3
```

It is possible to use the symbols IF, ELSE and ENDC to create compile-time if-statements. They can look like so:

```
X SET 5
IF X > 3
    add 5
ELSE
    add 3
ENDC
```

which would compile to just add 5.

It is also possible to make repetitions, using the REPT and ENDR symbols. For example,

```
REPT 10
    sub c
ENDR
```

compiles to

```
sub c
```

```

sub c
sub c
sub c
sub c
sub c
sub c
sub c
sub c
sub c

```

If multiple lines of code are inside the loop, this can however be bad from an optimization standpoint, as the compiled code can become large. A loop as defined in Chapter 2.5 can be significantly more compact.

Note that compile-time symbols do not have access to runtime variables. So you cannot, for example do `REPT c` to try to loop as many times as the integer stored in the C register, because the value is not known at compile-time. Loops, as defined in Chapter 2.5, can do such things, however.

The most powerful compile-time command is the macro. It provides a function-like way of writing code, for example:

```

AddTwoNumbers: MACRO
    ld a, \1
    add \2
ENDM

```

When this is written into the code, no actual code will be placed there by the compiler. It does however let you write, anywhere after the macro definition, something like

```
AddTwoNumbers 5,7
```

which would then compile to

```

ld a, 5
add 7

```

This can be extremely useful, for when you do want to repeat code across multiple places, perhaps with minor changes. Macros allow you to define such code only once.

Some caution should be taken when using macros; they might look like they provide a more "modern" programming style, as macros allow for example input arguments, but it is important to understand what the macros actually do, as it can be bad from an optimization standpoint, in some cases.

Every time a macro is called, it generates new code, which takes up space in the ROM. If a macro is called many times with the same arguments, it is probably better to create a run-time function instead, as that would only exist once in the code, saving space.

A special symbol \@ can be used, for when labels are used inside a macro. One example would be if one were to write a macro which contains a run-time loop, like so:

```
; Multiply two numbers X and Y
; By adding X onto X, Y-1 times
MultiplyNumbers: MACRO
    push bc

    ld a, \1 ; Store X in A
    ld b, (\2-1) ; Store Y-1 in B
    ld c, a

.loop\@:
    add c ; Add X
    dec b
    jr nz, .loop\@

    pop bc
ENDM
```

Here, the \@ will generate unique label names every time the macro is used; if we only called the local label .loop there might be a conflict, for example if the macro is used twice in a row like so:

```
MultiplyNumbers 5,4
MultiplyNumbers 6,3
```

Which would compile to code containing two .loop labels, which would lead to a compiler error.

This chapter only provides a basic understanding of the more commonly used symbols and macros. For more detailed information, check the RGBDS manual available here: <https://rednex.github.io/rgbds/> (information about macros and symbols can be found under "RGBASM language description").

2.8 Defining data

When you write code, it will be compiled into simple numbers to be stored in the ROM file. It is possible to generate arbitrary numbers into the ROM file, which can be used for many things that are not necessarily game code to be executed on the CPU. Since the Game Boy CPU is so limited, it is often a good idea to precompute any complicated calculations, if possible, and store the results in a table in the ROM, where the Game Boy can simply read the results instead of trying to compute them. This can be useful for things like trigonometric functions (to have an object move like a sine-wave across the screen, for example). Computing a sine-wave in real-time on the Game Boy CPU would probably be a lot slower than just having a pre-computed one which can be read from the ROM.

To define such a table, it is a good idea to create a label at the top, so that the table can be referred to in code. There exist compiler commands for defining such data: `db` (to define data as bytes, 8-bit numbers) and `dw` (to define so-called “words”, meaning 16-bit numbers). For example

```
SomeTable:
    db 5,3,7
    dw 1337
    db "Hello"
```

will place the following 8-bit numbers in the ROM at that position: 5,3,7,5,57,72,101,108,108,111, because 1337, as a 16-bit number, is written as %0000010100111001 which is divided into %00000101 (which is 5) and %00111001 (which is 57), and strings are simply stored as their ASCII-numbers (‘H’ is 72 in ASCII, and so on). There is no real distinction between data defined in different lines, the data is simply placed after the data from the previous line.

If we would want to use this table in our game, we could load the values like so:

```
ld hl, SomeTable
ld b, 10
.loop:
    ld a, [hl+]
    ; do something here... The first time,
    ; we will get 5 in A,
    ; the second 3 in A, and so on

    dec b
```

```
jr nz, .loop
```

Such tables are used to store graphics, audio, etc. in our ROM files. Note that the data is stored alongside the code, which means that, if we are not careful, the CPU might try to execute the data as if it were code, which will crash or lead to unexpected behaviour. This is easily avoided by making sure that the command above the data table is something like a jump or return, so that code execution never moves into the table itself. For example,

```
ld b, 6 ; some code here
jr MoreCode ; without this line,
; the CPU will think that the table
; data is code to run, and likely crash

; Here we define some table
SomeTable:
    db 1,2,3,4,5

MoreCode:
    ; Here we want to use the table or whatever
    ld a, [SomeTable]
```

If you find yourself in a situation where an emulator complains that you are executing "invalid opcodes", it means that it is trying to execute some data as if it were code, but it isn't. One way this could happen is if there is no jump command before a data table. Another possibility is a bank switching error (see Chapter 4.2).

Chapter 3

Making Game Boy games

The previous chapter focused on the Game Boy CPU and how to control it. To make a game, you need to control other pieces of hardware as well, that make graphics, audio and interpret user input. That will be the subject of this chapter, as well as going through some practical details of how to work with the RGBDS compiler. By the end of it, you should be able to compile a Game Boy ROM that can be loaded into an emulator or real hardware, and start making an actual game.

A good Game Boy emulator for game development is BGB, which is available here: <http://bgb.bircd.org/>. It has good features for debugging, and accurately emulates most of the Game Boy's quirks.

3.1 About RGBDS

RGBDS is a Game Boy development system, and probably the compiler most commonly used today for making Game Boy games. It contains an assembler (which converts assembly code to binary), a linker (which converts one or more binaries into a Game Boy ROM file) as well as a "fixer" which fixes some common mistakes in the ROM file's header (like the checksum). You can find RGBDS and up-to-date install instructions for several operating systems are available here: <https://github.com/rednex/rgbds>

3.2 The GingerBread library

The GingerBread library is an attempt to make a kind of standard library for developing Game Boy games using RGBDS assembler. It is written by the author of this book, to make game development a bit easier, and allows this book to skip some boring technical details. By relying on this library, this book can focus on understanding the important things and getting games up and running quicker. The library is open source (with the extremely permissive Unlicense license) and available here: <https://github.com/ahrnbom/gingerbread>.

Many examples in the chapter will make use of constants and functions defined in this library, and this will be pointed out every time, to make sure the distinction between RGBDS standard functionality and GingerBread is clear.

3.3 Memory mapping

There is no official set of functions to control the various hardware inside the Game Boy, something like an API. Instead, memory mapping is used to directly control the hardware. When reading and writing to memory addresses, using the `ld` command, those addresses can point to plenty of different things, not just positions in RAM. By reading and writing special numbers to and from special addresses, we create graphics, interpret user input and produce sounds.

An easy example is the memory address `$FF47`, which controls the palette of background tiles. Since `$FF47` might be difficult to remember, it is usually given a name, like so:

```
BGPAL equ $FF47
```

Now, we can use the symbol `BGPAL` to refer to this memory address, whenever we want to change the background palette. The background tile palette basically allows us to swap the four colours that the Game Boy displays, allowing some neat effects like fading the screen to black or white "smoothly" by changing the colours multiple times, each time making the screen either darker or brighter. By writing a single 8-bit number to this address, we specify all four colours at once, each being specified by two bits. The default palette is `%11100100`, which means that the colours that is usually the darkest (the leftmost two bits) should actually be the darkest (`%11`), while the dark-ish

colour (the 3rd and 4th bits, from the left) should be dark-ish (%10) and so on, with %00 being the brightest colour. For example, doing

```
ld a, %00011011
ld [BGPAL], a
```

will invert the colours on the background tiles, so that parts that are usually bright are now dark and vice versa, while

```
ld a, %11110000
ld [BGPAL], a
```

will make it so that the background tiles only use two colours, either black or white (so that the parts that were before dark-ish are now completely dark, and the bright-ish parts are now at the brightest), like a "high contrast" mode.

Note that the address \$FF47 is defined in the GingerBread library as BG_PALETTE, not BGPAL.

3.4 Sections

Any ASM code you write for RGBDS needs to be placed in some section. A section is defined by a line that can look like this:

```
SECTION "Some_stuff",ROM0
```

or

```
SECTION "Some_other_stuff",ROMX,BANK[1]
```

The main difference between the two is where in the ROM the code will appear, the ROM0 is another name for bank 0, which is a special part of the game ROM which is always accessible, where the "main code" usually lies, to some extent, while the second line stores the data in Bank 1. The non-zero banks can be changed, and only one non-zero bank is accessible at the time. Each bank can only contain a limited number of bytes of compiled code or game data. This system allows the game to be quite large (up to something like 8 MB) even though only a small part of the 16-bit address space is reserved for pointing at game code. The downside is that the developer needs to take care of bank switching manually, to make sure the right data is available when needed. This will be covered in more detail in Chapter 4.2, as it is a slightly more advanced topic. For now, we will assume that all of our code fits in the 0 and 1 banks.

3.5 The starting point

GingerBread contains very basic code for starting up the game. The library expects that the game defines a label called `begin` which is where the game's logic can start. Some things are necessary to do there, like calling `StartLCD` and `EnableAudio` (both GingerBread functions) but it is probably a good idea to load some graphics first (see Chapter 3.9).

3.6 The header

A special part of the game ROM is called the header. This part contains information about the game, that the Game Boy needs to know about before it even begins running any code. It is used to specify what kind of cart the game runs on (and if you write the game onto a real physical cart, that cart must support the cart type specified in the ROM file), if any GBC/SGB features are used, etc. If any changes are made to the ROM header, it's a good idea to run the game ROM in BGB (which will complain if there are any header errors) and then look at Right Click → Other → Cart Info, which will (hopefully) confirm how a Game Boy would interpret the header.

GingerBread will provide your game with a sensible default game header. There are some options which can be used to adjust the header settings. These options are changed by defining compile-time constants prior to importing the GingerBread library. The options are as follows:

GAME_NAME: Sets the name of the game. This is mostly for cosmetic reasons when running in emulators that may display it. It should be up to 15 characters, in capital letters. Since **GAME_NAME** is a string (text) the **EQU** command should be used for defining it, while the other options are specified with **EQ** which works with numerical values.

SGB_SUPPORT: Makes the game support Super Game Boy functionality. By setting this to 1, GingerBread functions related to the Super Game Boy become available (see Chapter 5) but this also makes the GingerBread library take up more space on Bank 0.

GBC_SUPPORT: Makes the game support Game Boy Color functionality. By setting this to 1, GingerBread functions related to the Game Boy Color become available (see Chapter 6) but this also makes the GingerBread library take up slightly more space on Bank 0.

ROM_SIZE: Specify the size of the ROM file, where 0 means 32 kiB, 1 means

64 kiB, 2 means 128 kiB and so on up until 8, which means 8 MiB. If the RGBDS compiler complains about there not being enough space for the ROM file, increase this number. But also make sure than any physical carts used contain sufficient space for the game.

RAM_SIZE: Specify the size of save RAM on the cartridge, where 0 means no save RAM, 1 means 2 kiB, 2 means 8 kiB, 3 means 32 kiB, 4 means 128 kiB and 5 means 64 kiB. See Chapter 3.6. Make sure any physical carts used for the game contain sufficient amounts of save RAM.

An example of setting up a game called SNAKE with Super Game Boy support, but no Game Boy Color support, with a 128 kiB ROM and 2 kiB of save RAM is as follows:

```
GAME_NAME equs "SNAKE"
SGB_SUPPORT equ 1
;GBC_SUPPORT equ 1 ; Note that this line is commented out,
; setting it to 0 does not disable GBC support
ROM_SIZE equ 2
RAM_SIZE equ 1
INCLUDE "gingerbread.asm"
```

These lines should be near the top of the .asm file for your game. Make sure GingerBread is only included once.

3.7 Interrupts

One way the Game Boy handles timing is via something called interrupts. An interrupt means that, at some special time, the CPU gets interrupted and stops doing whatever it was doing before, and jumps to a special address in the game's code. At the same time, it pushes its the previous position in code onto the stack, so that it's possible to return there after dealing with the interrupt.

Interrupts can be enabled and disabled via the special CPU commands `ei` (for "enable interrupts") and `di` (for "disable interrupts"). After running a `di` command, there will be no interruptions in the CPU's execution until it runs an `ei` command, or the special command `reti` which is a combined `ret` and `ei` command. While one interrupt is being processed, any other interrupts are disabled, so an interrupt handling code which does nothing is simply a `reti` command; it returns to wherever code was executing before and enabled interrupts again, in a single call.

The GingerBread library listens to VBlank interrupts (every time the entire screen finishes drawing) and uses this to update the positions of sprites. If other interrupts are desired, the GingerBread library needs to be modified, under "boot process and interrupts" to both jump somewhere when that interrupt happens (similar to the `jp DMACODE_START` that happens on VBlank) and also the `GingerBreadBegin` routine needs to be modified so that more interrupts are fed into `rIE`, so that the Game Boy knows that those other interrupts should be listened to. Make sure that the code you jump to ends with `reti` or otherwise re-enable interrupts.

3.8 RAM

The main place to store your own run time data is in the Game Boy's RAM. It occupies the addresses between `$C000` and `$E000` and can thus store 8 kB of data. There can also be RAM inside of cartridges, stored in `$A000` up to `$BFFF` (although the entire space might not be usable, depending on how much RAM there is in the cart). There is no real data allocation system when coding in ASM using RGBDS, and it will be up to you to ensure that you read and write only to addresses within this space. To help with that, you can give specific positions in RAM names. If you want to store multiple values, simply let the name point to the first one. The rest can then be accessed via addition. An example is the following:

```
Position EQU $C200 ; 2 values
SpeedVector EQU Position+2 ; 2 values
CharacterHP EQU SpeedVector+2 ; 1 value
CharacterMP EQU CharacterHP+1 ; 1 value
```

Then, to write the vector (5, 3) to the `SpeedVector`, we can do something like

```
ld a, 5
ld [SpeedVector], a
ld a, 3
ld [SpeedVector+1], a
```

This syntax is easy to understand but somewhat error-prone, as the word "Position" appears twice and without the comments, it's not super obvious how many values we want (especially for the last named place in the list).

Therefore, there exists a special syntax for dividing RAM into sections. For example,

```
SECTION "Some_variables_in_RAM", WRAM0[$C200]
Position: DS 2
SpeedVector: DS 2
CharacterHP: DS 1
CharacterMP: DS 1
```

works the same way as above. Just note that a new section has to be created afterwards for game code, as no code can be placed in RAM sections. The operation DS simply reserves space, with the number afterwards specifying the number of bytes to allocate.

Note that GingerBread reserves some space in RAM, for sprite graphics (see Chapter 3.10) and to accomodate for GBT-Player (see Chapter 4.4), so usable RAM for your game starts at \$C200 (as defined as USER_RAM_START in GingerBread).

3.9 Graphics: backgrounds

The graphics on the Game Boy are divided into three separate layers, which can all display different visuals, and they are placed over each other. The first layer is the background layer, which is always drawn the furthest "back" and can use all four "colors" on the Game Boy. This layer is usually used to draw the game world. The second layer is used for drawing sprites, which are movable objects like characters, enemies, projectiles, power ups etc. This layer has one "color" reserved for transparency so that the background can be visible behind objects. The third layer is called the "window" and works a lot like the background layer, but does not necessarily cover the entire screen and is drawn above the background layer (the drawing order of the sprite layer and window layer can be controlled). The window layer is usually used for things that should have a static position on screen, regardless of the scrolling of the background and sprites, like a health meter, menus and score counts, because it does not support scrolling. The rest of this section will focus on the background layer.

If you draw a 160×144 image using an image editing program on your computer, and make sure to only use four colors, like four shades of gray, green or some other color (similar to what you'd find on a Game Boy screen). The image should be saved as a lossless .PNG. Then, a tool called Game boy Tile Data

Generator (available at <http://www.chrisantonellis.com/gameboy/gbtdg/>, source code here <https://github.com/chrisantonellis/gbtdg>) lets you select an image and convert it to assembly code for RGBDS. At the time of writing, this tool is not hosted, but if the source code is downloaded then the `index.html` file can be opened in a browser locally and the tool still works.

Such assembly code mainly contains DB commands for defining the binary code of the graphics. It's divided into two main parts: the tile data that describes every tile used in the image, and the map data that explains how the tiles should be placed on screen. Because of this division, reusing the same tiles across multiple places results in significant savings in the amount of storage needed for the visuals. Because there's a limit to how many different tiles that the Game Boy can keep in its VRAM, it can even be necessary to reuse tiles as much as possible. A classic example of a similar strategy is in *Super Mario Bros.* on the NES, where the bushes and clouds are actually the same image (but colourized differently).

To be able to use the graphics in your ASM code the `.inc` file needs to be included, by a command like

```
INCLUDE "your_image.inc"
```

To display the image, you need to first copy the image tile data onto the tile section of VRAM, and then copy the map data to the map data section of VRAM. There are GingerBread functions for this. For example, the function `mCopyVRAM` can be used for copying data from the game's ROM onto VRAM. Note that standard copying functions, simply using `ld` many times, will generally not work when writing to VRAM because of timing issues; it's only possible to write to VRAM at specific times when the GPU is not busy. VRAM-safe functions will wait for these moments to write their data. If you makes game and the graphics end up garbled (but you can still see some resemblance of what it was supposed to look like) this is quite likely the problem. This only applies when the GPU is turned on; it's possible to programmatically turn it off temporarily which might be a good idea if large amounts of data is to be copied to VRAM.

An example of using `mCopyVRAM` from gingerbread is

```
ld hl, example_tile_data
ld de, TILEDATA_START
ld bc, example_tile_data_size
call mCopyVRAM
```

For this function, the 16-bit value in HL should be the address to where the data should be copied from, and DE should contain the address to where the data should be copied to (assumed to be somewhere in VRAM, the constant `TILEDATA_START` is defined in `GingerBread` and points to the start of the VRAM part where tiles should be defined), while BC contains the length of the data to be copied, as a 16-bit integer.

Map data is copied similarly, but some care must be taken when using the `GBTDG` tool if the image is smaller than the 256×256 rendering surface (32×32 tiles). The `GBTDG` writes the map data in order, from left to right, line by line, and doesn't mark where one line ends and another begins. If all the map data is copied straight to the map data storage in VRAM, parts of the image will be placed outside of view and the graphics will end up garbled (but all tiles will look correct, they'll just be in the wrong place). To solve this, one can right a simple macro like

```
; Macro for copying a rectangular region into VRAM
; Changes ALL registers
; Arguments:
; 1 - Height (number of rows)
; 2 - Width (number of columns)
; 3 - Source to copy from
; 4 - Destination to copy to (assumed to be on VRAM)
CopyRegionToVRAM: MACRO
```

```
I SET 0
REPT \1

    ld bc, \2
    ld hl, \3+(I*\2)
    ld de, \4+(I*32)

    call mCopyVRAM
```

```
I SET I+1
ENDR
ENDM
```

The macro simply copies map data line by line. It can be used like

```
CopyRegionToVRAM 18, 20, example_map_data,
    BACKGROUND_MAPDATA_START
```

where `BACKGROUND_MAPDATA_START` is the start of the map data storage in VRAM (\$9800), a constant defined in `GingerBread`.

Note that the map data that `GBDTG` creates is based on the assumption that the tile data will be placed starting from the beginning of the tile data region of VRAM, as the map data starts with 0, 1 and so on. If you load multiple images into VRAM at once, one of them will have to be loaded after another, for example

```
ld hl, example1_tile_data
ld de, TILEDATA_START
ld bc, example1_tile_data_size
call mCopyVRAM

ld hl, example2_tile_data
ld de, TILEDATA_START+example1_tile_data_size
ld bc, example2_tile_data_size
call mCopyVRAM
```

In such a case, the map data need to be modified by adding a constant to all the values in it (either at run-time or at compile-time, or perhaps before compilation, via some script).

3.10 Graphics: sprites

Sprites are graphical objects that appear on top of the background tiles. Unlike background tiles, they can move around freely (including positions that aren't divisible by 8). Their background color is transparent, allowing the background tiles to be seen behind them. The Game Boy can handle up to 40 sprites at once, and at most 10 sprites per horizontal line. Each sprite is either 8×8 (a single tile) or 8×16 (two tiles, one on top of the other), depending on which mode the GPU runs in. The latter mode is enabled by default in `GingerBread`, as it allows a larger number of sprite tiles to be on screen at once. The tile data used for sprites are shared with background tiles and window tiles. Because most games use objects that are larger than 8×16 , objects are often made up of multiple sprites placed next to each other. This does typically not result in any visual artefacts or seams, as the resolution of the screen is low, and sprites are always placed at integer pixel values, giving the impression of larger sprites moving across the screen.

To tell the Game Boy GPU which tiles to place where, it's necessary to write to VRAM using VRAM-safe writing operations. Because sprite data is often updated every frame, waiting before writing every single byte can end up taking too long. Therefore, a technique called DMA (Direct Memory Access) is used to quickly copy a larger amount of data during the short time windows when VRAM is writable. This is a bit technical and involved, so it won't be described in further detail in this book (a good summary is available here, for the interested reader: <https://exez.in/gameboy-dma>). This technique is implemented in GingerBread. By writing sprite data to RAM positions starting at the address defined as `SPRITES_START`, GingerBread makes sure to copy these values onto VRAM every frame.

The format of the data is quite simple. Each sprite is made up of 4 bytes: first the Y position, then the X position, then the tile number and finally an options byte. The positions are from 0-255, allowing the sprites to move freely across the 256×256 rendering surface. The tile number is a bit special when working with 8×16 sprites: all the tiles are divided into pairs of two, and only a pair can be used as a sprite, with the one on the lower address appearing above the other one. When selecting a tile number, it doesn't matter which one from the pair one chooses. To confirm the tile numbers, a VRAM viewer in an emulator can be helpful (see Chapter 7.1). Finally, the options byte contains 8 options, one bit each. The highest four bits correspond to rendering priority (allowing some sprites to be placed on top of others, in case they overlap), flipping along the y-axis and flipping along the x-axis, and finally choosing between two sprite palettes (on non-GBC systems). The remaining four bits are used only by the Game Boy Color, if the game is running in GBC mode (see Chapter 6).

3.11 Reading user input

Reading values from the D-pad and buttons is actually a bit involved, as the hardware is directly controlled from the Game Boy CPU. The GingerBread library has a function called `ReadKeys` which writes a byte to the A register, where each bit represents a single input.

The first (highest, leftmost) bit represents the Start button, followed by Select, B, A, down, up, left and right. For example, the value in A after running `ReadKeys` will be `%00010010` if the player is pressing both left on the D-pad and the A button.

For example, you can do

```
call ReadKeys
and KEY_A
cp 0
call nz, doSomethingIfAIsPressed
```

Which will call `doSomethingIfAIsPressed` if A is pressed, and not otherwise. The call will be made even if the user is also pressing some other button; it only checks whether the A button is pressed or not. Note that the `ReadKeys` command does not wait for user input, so you may want to run such code in a loop, to give the user more than one opportunity to perform the given action.

3.12 Audio: Sound effects

The Game Boy produces sounds from four audio channels: two square wave channels, of which the first supports both "sweep" and "envelope" effects, while the second only supports "envelope" effects, and one customizable wave channel, and finally a white noise channel, also supporting an "envelope" effect.

The "sweep" effect means that the frequency of the square wave changes while the sound is playing, while the "envelope" effect means the output volume changes. Both of these effects are commonly used for making a large variety of sound effects.

The `GingerBread` library supports playing basic sounds. The `"EnableAudio"` and `"DisableAudio"` functions turn audio on and off, with sensible default settings (maximum output volume, all channels active, and all channels output to both speakers in case stereo headphones are connected). If other settings are desired, you need to implement this yourself (the relevant addresses are \$FF24-\$FF26, defined in `GingerBread` as `SOUND_VOLUME`, `SOUND_OUTPUTS` and `SOUND_ONOFF`).

`GingerBread` has a function called `PlaySoundHL` which is designed to play sound effects. First, a sound effect needs to be specified as a data table, containing first a 16-bit "word" specifying which channel to be used, using one of the four `SOUND_CH1_START` through `SOUND_CH4_START`. This should then be followed by five bytes, containing the data to be written to that channel's addresses. For each channel, the meaning of each bit is a bit different, which will be described right after this example of a sound effect:


```

Sound_ball_bounce: ; give the sound a name in the code
DW SOUND_CH4_START ; specify which channel
; in this case channel 4 (noise)
DB %00000000 ; Data to be written to SOUND_CH4_START
DB %00000100 ; Data to be written to SOUND_CH4_LENGTH
DB %11110111 ; Data to be written to SOUND_CH4_ENVELOPE
DB %01010101 ; Data to be written to SOUND_CH4_POLY
DB %11000110 ; Data to be written to SOUND_CH4_OPTIONS

```

which can be played like so:

```

ld hl, Sound_ball_bounce
call PlaySoundHL

```

The comments to GingerBread's definition to the sound addresses explain what each bit does, so it is included below for reference. For more details on what the bits do, there's more detailed information at http://gbdev.gg8.se/wiki/articles/Sound_Controller and http://gbdev.gg8.se/wiki/articles/Gameboy_sound_hardware.

```

; Channel 1 (square with sweep and envelope effects)

; bit 7: unused,
; bits 6-4: sweep time,
; bit 3: sweep frequency increase/decrease,
; bits 2-0: number of sweep shifts
SOUND_CH1_START      EQU $FF10

; bits 7-6: wave duty,
; bits 5-0: length of sound data
SOUND_CH1_LENGTH     EQU $FF11

; bits 7-4: start value for envelope,
; bit 3: envelope decrease/increase,
; bits 2-0: number of envelope sweeps
SOUND_CH1_ENVELOPE   EQU $FF12

; bits 7-0: lower 8 bits of the sound frequency
SOUND_CH1_LOWFREQ    EQU $FF13

; bit 7: restart channel,
; bit 6: use length,
; bits 5-3: unused,

```

```

; bits 2-0: highest 3 bits of frequency
SOUND_CH1_HIGHFREQ EQU $FF14

; Channel 2 (square with envelope effect,
; with no sweep effect)
SOUND_CH2_START EQU $FF15 ; Not used

; bits 7-6: wave duty,
; bits 5-0: length of sound data
SOUND_CH2_LENGTH EQU $FF16

; bits 7-4: start value for envelope,
; bit 3: envelope decrease/increase,
; bits 2-0: number of envelope sweeps
SOUND_CH2_ENVELOPE EQU $FF17

; bits 7-0: lower 8 bits of the sound frequency
SOUND_CH2_LOWFREQ EQU $FF18

; bit 7: restart channel,
; bit 6: use length,
; bits 5-3: unused,
; bits 2-0: highest 3 bits of frequency
SOUND_CH2_HIGHFREQ EQU $FF19

; Channel 3 (custom wave)
; bit 7: on/off, bits 6-0: unused
SOUND_CH3_START EQU $FF1A

; bits 7-0: length of sound
SOUND_CH3_LENGTH EQU $FF1B

; bits 6-5: audio volume
; %00 is mute, %01 is loudest,
; %10 is pretty quiet and %11 is very quiet
SOUND_CH3_VOLUME EQU $FF1C

; bits 7-0: lower 8 bits of the sound frequency
SOUND_CH3_LOWFREQ EQU $FF1D

; bit 7: restart channel,

```

```

; bit 6: use length,
; bits 5-3: unused,
; bits 2-0: highest 3 bits of frequency
SOUND_CH3_HIGHFREQ EQU $FF1E

; Channel 4 (noise)
SOUND_CH4_START EQU $FF1F ; Not used

; bits 5-0: length of sound
SOUND_CH4_LENGTH EQU $FF20

; bits 7-4: start value for envelope,
; bit 3: envelope decrease/increase,
; bits 2-0: number of envelope sweeps
SOUND_CH4_ENVELOPE EQU $FF21

; bits 7-4: polynomial counter,
; bit 3: number of steps (15 or 7),
; bits 2-0: ratio of frequency division
; (%000 highest frequency, %111 lowest)
SOUND_CH4_POLY EQU $FF22

; bit 7: restart channel,
; bit 6: use length
SOUND_CH4_OPTIONS EQU $FF23

```

When referring to bit numbers, bit 7 is the most significant bit (the furthest to the left, when written as for example %10101010), while bit 0 is the least significant bit (furthest to the right).

Channel 3 is especially interesting, as it allows custom wave forms to be played. Some games use this to play some fairly detailed sounds, like voice samples. Be aware that this requires you to constantly write new wave form data while the sound is playing, which uses up most of the Game Boy's CPU (which is why gameplay is typically inactive while such sounds are playing). It is however possible to define your own wave form once (or rarely) and use it for background music or sound effects, to give your game some more unique audio. The wave form should be written to addresses \$FF30-\$FF3F (defined as SOUND_WAVE_TABLE_START and SOUND_WAVE_TABLE_END in GingerBread).

Chapter 4

More involved topics

4.1 Memory map

As mentioned, the Game Boy has a single 16-bit address space, where each possible address between `$0000` and `$FFFF` pointing to something in the hardware. The space is divided as follows:

`$0000-$3FFF` contains the game code for bank 0 which is located in the ROM on the cartridge. It is read-only. Some parts of this range have specific purposes, like `$0040` where code execution jumps during vblank interrupts (see Chapter 3.7), and `$0104` which is where the ROM header starts (see Chapter 3.6).

`$4000-$7FFF` contains the game code for the switchable bank, which at the start of the game will be bank 1 (see Chapter 4.2). When the chosen bank changes, the same addresses here will point to a different part of the game's ROM code. It is read-only.

`$8000-$9FFF` contains VRAM, or graphics memory. In Game Boy Color mode, this section is switchable with two different banks, otherwise it will only have a single bank. It is where the data for what the tiles should look like (from `$8000-$97FF`) and where the tiles should be placed in the background (`$9800-$9BFF`) and window (`$9C00-$9FFF`). See chapter 3.9. It can both be read and written, but only at certain times (see Chapter 3.9)

`$A000-$BFFF` contains cartridge RAM, in case the cartridge has such RAM in it. This RAM is sometimes called SRAM and is usually used for save data.

It can be both read and written, but needs to be activated before use (and should be deactivated after use, see Chapter 4.5).

`$C000-$CFFF` contains general purpose RAM which is physically located inside the Game Boy itself. It is non-switchable, and often called Bank 0 of WRAM. It can both be read and written. Note that GingerBread reserves `$C000-$C1FF`, so usable RAM for your game starts at `$C200` (as defined as `USER_RAM_START` in GingerBread).

`$D000-$DFFF` contain bank 1 of WRAM, unless the game runs in GBC mode in which case it is switchable with banks 1-7. Like `$C000-$CFFF`, it is general purpose, and can both be read and written.

`$E000-$EFFF` is a copy of `$C000-$DFFF` and is usually not used. Not all emulators emulate this behaviour.

`$FE00-$FE9F` contains data for sprites, like which tiles they should look like and where they should be placed, see Chapter 3.10. It can both be read and written, but only at certain times.

`$FEA0-$FEFF` does not contain anything meaningful.

`$FF00-$FF7F` contains many different things related to different hardware, like graphics, audio, the link cable, buttons and so on. It contains both read-only parts, write-only parts and parts that can both be read and written.

`$FF80-$FFFE` contains the so-called High RAM, or HRAM, which is special because it can be used with DMA transfers (unlike WRAM) which is why it is often used for keeping sprite data. Therefore it should probably not be used for general purpose data, unless you know what you are doing. It can both be read and written.

Finally the address `$FFFF` is used for specifying which interrupts the game should use (see Chapter 3.7)

4.2 ROM banks, bank switching

In Section X, note how only `$0150-$7FFF` are used for the game's ROM data. This is only around 32 kB of data. Many games will require more space than that (remember that not only game code, but also visual and audio assets all need to be stored on the game's ROM). To solve this, the game's ROM is divided into several banks, and each bank is 16 kB in size, meaning that two banks are accessible in the memory map at any time. Bank number zero is special, in that it is always accessible at `$0150-$3FFF` and can never be switched out, while `$4000-$7FFF` is by default bank number one, but this can

be changed at runtime. By writing a byte containing a number larger than or equal to one to the bank switching address `$2000` (called `ROM_BANK_SWITCH` in `GingerBread`), the non-zero bank is changed. Beware however, that if code execution is currently on a non-zero bank when this happens, the game will most likely crash, as the CPU keeps track of the next line to execute by an address, and if the bank suddenly changes, that address will probably no longer point at what the developer had in mind (for example, if the CPU tries to execute visual assets as code, the game typically crashes, and in an emulator like `BGB`, you typically get errors like "invalid opcode").

4.3 Graphics: text and numbers

In principle, rendering text and numbers is no different from rendering any other background tiles (or sprites, if the text should move across the screen, however this chapter will only cover statically placed text, drawn to the background or window layers, as that's more commonly used). There are some tricks that make the process of rendering text and numbers easier that are worth knowing about.

By far the most common way to render text and numbers on the Game Boy is to let each character be a single tile (8×8 pixels), and the rest of this chapter will assume this. This is easy to implement and gives a text that's reasonably sized for most purposes. One can draw such a font as an image, with the letters, numbers and characters one wants and then load it into VRAM. Make sure that the numbers are ordered as `0123456789` (and not for example `1234567890`). Make sure to take note of the order of all your characters and which tile number the '0' character appears at, as this is important for the special case of writing numbers (it might be easier to simply load the font into VRAM and check in an emulator like `BGB` to easily see which tile numbers the characters end up at, see Chapter 7.1).

When writing numbers, it's important to understand the difference between decimal and hexadecimal numbers. If you store some numerical value in your game that should be presented in text, like for example the player's score, it might be tempting to store it like you would any other value, as a hexadecimal number (for example, the number 13 would be stored as `$0D`, or `%00001101`) but in order to present this on screen, in the decimal format that humans expect, the number would need to be converted, which is non-trivial on the Game Boy. Therefore, the common solution is to simply store

the number is decimal form to begin with. That means that, for example, the number 13 should be stored as \$13 (which really means 19, but for the purpose of displaying it on screen, that doesn't matter). That way, each 8-bit number contains two decimal numbers, and to display them, we can divide the 8-bit number into two 4-bit numbers and for each add them to the tile number of the 0 character, to find which tile number we should draw. The GingerBread function `RenderTwoDecimalNumbers` does this. The GingerBread function `RenderFourDecimalNumbers` works similarly but for 16-bit numbers, which give decimal numbers with four digits.

One thing we need to be aware of when working with decimal numbers like this is that normal numerical operations no longer work as expected. If you store the decimal number 9 as \$09, and then increase it with an add command, like add 2, then it will become \$0B which is not a valid decimal number. The CPU command `daa` fixes this, turning \$0B into \$11 in this case (working only on the A register). The `daa` command only works as expected if used immediately after the commands `add`, `adc`, `sub` or `sbc`.

Some more examples:

```
ld a, $39
add $12 ; A now has $4B, but we want $51
daa ; A now has $51

sub $2 ; A now has $4F, but we want $49
daa ; A now has $49
```

To display the number 49 on position $x = 12$, $y = 3$ on the background, one could now do the following with GingerBread, assuming the tile number of the '0' character is \$5A:

```
ld b, $5A ; Tile number of '0'
ld c, 0 ; Use background
ld d, 12 ; X position
ld e, 3 ; Y position
call RenderTwoDecimalNumbers
```

To write the number "1234" at $x = 1$, $y = 1$, one could do the following:

```
ld hl, $1234
ld b, $5A ; Tile number of '0'
ld c, 0 ; Use background
ld d, 1 ; X position
ld e, 1 ; Y position
```



```
call RenderFourDecimalNumbers
```

For drawing text, one needs to be aware of how RGBDS handles strings. If you define a string like

```
db "This_is_an_ASCII_string"
```

then by default, RGBDS will store the ASCII values of those characters as numbers. This might not be what you want, since most likely your character tiles will not be placed exactly according to the ASCII table (for example, your character 'A' might not be at tile number 65) making it cumbersome to figure out what tiles to draw, in order to draw the string. To fix this, RGBDS contains a command called CHARMAP which lets you map characters to numbers. For example

```
CHARMAP "A", 1
CHARMAP "B", 2
CHARMAP "C", 3
CHARMAP "<heart>", 4
CHARMAP "_", 0
```

```
AbbaCabText:
DB "ABBA_CAB<heart>"
```

will write 1,2,2,1,0,3,1,2,4 to the ROM. Note that the CHARMAP commands need to appear before the text definitions in the code. Also note that it is not possible to use a single CHARMAP command to specify the entire alphabet; one CHARMAP command is needed for each character.

If one defines one character to mark the end of a piece of text, and use the tile number of some non-text tile, like the following:

```
CHARMAP "<end>", 150
```

then one can use the GingerBread function RenderTextToEnd, which will automatically stop writing when this character appears. Otherwise, RenderTextToLength can be used, with a pre-set number of characters. For example, the following code renders the text defined above:

```
ld b, 10 ; Number of characters
ld c, 0 ; Draw to background
ld d, 5 ; X position
ld e, 6 ; Y position
ld hl, AbbaCarText ; Address of text start
call RenderTextToLength
```

And example with text with the explicit end character:

```
TextWithEndCharacter:
BD "ABA_CACABA<end>"

ld b, 150 ; End character
ld c, 0 ; Draw to background
ld d, 5 ; X position
ld e, 6 ; Y position
ld hl, TextWithEndCharacter ; Address to text start
call RenderTextToEnd
```

The functions `RenderTwoDecimalNumbers` and `RenderFourDecimalNumbers` take the tile position to draw to as X and Y coordinates. This is supposed to make them easier to understand, and might be especially useful in situations where the numbers or text need to move around the screen. For most situations however, numbers or text have static positions, and it might be beneficial to precompute position numbers to make the game execute faster, especially if there are many calls to these functions every frame. Because the drawing space is 256×256 pixels, which is 32×32 tiles, and the tile numbers are simply numbered row-first starting with 0 at the top left corner, the tile position number is computed by the formula $pos = x + 32 \times y$. Because the position numbers are 16-bit, we cannot simply implement the multiplication by 32 as a bit shift, as those are not implemented for 16-bit numbers on the CPU. Instead we need to perform 16-bit add operations 32 times. If one wants to avoid this for performance reasons, it is possible to use the functions `RenderTwoDecimalNumbersByPosition`, `RenderFourDecimalNumbersByPosition`, `RenderTextToLengthByPosition` and `RenderTextToEndByPosition` that take a 16-bit position number instead of the X and Y coordinates as input. The following example is identical to the one above, except it runs faster:

```
ld b, 150 ; End character
ld c, 0 ; Draw to background
ld de, 5 + 32*6 ; Position number
ld hl, TextWithEndCharacter ; Address to text start
call RenderTextToEndByPosition
```

Note that the multiplication is performed at compile time (the Game Boy CPU does not support multiplication).

4.4 Audio: music

GingerBread is designed so that it can be used together with GBT-Player, a music playing library for Game Boy development available here: <https://github.com/AntonioND/gbt-player>.

GingerBread includes GBT-Player as a submodule. That means that by executing

```
git submodule init
```

in the GingerBread folder will download GBT-Player (as a subfolder called "gbt-player").

The scripts for assembling the example game in GingerBread has a switch called `include_gbt` which can be disabled to compile the game without GBT-Player. To do that, it is also necessary to comment out the line `USE_GBT_PLAYER EQU 1` in `example.asm`, so that the example game doesn't try to execute GBT-Player code that doesn't exist.

To include music in a game, it is necessary to first construct `.mod` files for the music tracks using some tracker software. Then, the application "mod2gtb" (which is included in GBT-Player) should be used to convert the songs to `.asm` files. Once the `.asm` files for the songs are included in the compilation and linking of your ROM file, you can tell GBT-Player which song it should play with commands like

```
ld de,your_song_data ; Pointer to the song data
ld bc,BANK(your_song_data) ; GBT-Player changes banks
ld a,$05 ; Set the tempo
call gbt_play ; Play song
```

Then, before every `halt` call in your code, it is necessary to call the function `gbt_update` to make sure GBT-Player keeps playing notes and advancing the music.

Note that every call to functions from GBT-Player (like `gbt_play` and `gbt_update`) changes the current ROM bank (see Chapter 4.2), so you may need to change the ROM bank back afterwards. This is especially problematic if the code that calls these functions are not placed in bank 0. The simplest solution is probably to define a function on bank 0 which calls `gbt_update`, executes `halt` and then switches to a ROM bank depending on the value in some register. The example game contains the following function for this purpose:

```

; To keep music playing at all times,
; call this before every halt command
;
; A - ROM bank to switch to after updating music
; (GBT Player switches banks when updating the music)
UpdateMusic:
IF DEF(USE_GBT_PLAYER)
    push bc
    push af
    call gbt_update

    pop af
    pop bc
    ld [ROM_BANK_SWITCH], a
ENDC
ret

```

GBT-Player allocates some RAM between `$C000` and `$C0FF`. Therefore, GingerBread doesn't store anything in this range. If a game doesn't rely on GBT-Player, this range could be used for game data as well.

4.5 Save data

If the ROM header specifies that there should exist save data (and, if the game is on a physical cart, the cart supports this) then it's possible to read and write save data to the addresses in range `$A000-$BFFF`. However, by default, the save data region is disabled, to reduce the risk of a crashed game overwriting valuable data, leading to the loss of a player's progress. The programmer must enable the save data region whenever they want to read or write from save data, and should then disable it as soon as it is no longer needed. The GingerBread functions `EnableSaveData` and `DisableSaveData` do this. Call `EnableSaveData` prior to accessing the save data, and `DisableSaveData` afterwards. Also, it is possible to have multiple banks of save data, in case it doesn't fit into the 8 kB address space, by using the `ChooseSaveDataBank` function in GingerBread, although this is only supported on cartridges with MBC5 and a sufficient amount of SRAM. Call `ChooseSaveDataBank` before `EnableSaveData` to make sure the correct SRAM bank is loaded.

Chapter 5

Super Game Boy Features

The Super Game Boy is a fascinating piece of hardware history, worthy of its own chapter in this book.

The Super Game Boy cartridge contains the hardware for an actual Game Boy inside it, except for the screen and speaker, and some electronics for connecting it to the Super Nintendo. The Super Nintendo's CPU and the Game Boy's CPU are thus running at the same time, and are mostly independent of each other. The Game Boy hardware sends images and sound to the Super Nintendo, which mostly just presents these on the TV. What makes the Super Game Boy so interesting is that a Game Boy game designed for this hardware configuration can perform specific commands to control how the Super Nintendo should display the image, create Super Nintendo sounds that play over the game's audio and specify a border image to be displayed around the game play window on the TV.

Using the Super Game Boy can get quite technical. All the details are in the CPU Manual and Pan Docs, so this chapter will focus on using the functionality included in GingerBread.

All Game Boy games that run on the original DMG model will also work on the Super Game Boy, without any additional code, but this results in a somewhat bland experience with the default borders and color palettes. This chapter will explain how to customize the experience. In order to activate Super Game Boy functionality, it is necessary to specify that in the game's header. GingerBread takes care of this, as long as `SGB_SUPPORT` is defined before GingerBread is loaded.

To send commands to the Super Game Boy, the memory address `$FF00`. This address is usually used to reading the D-pad status, but by writing to it, the Super Game Boy can read commands sent from the game. Because many games will use interrupts that react to button presses, interrupts should be disabled whenever messages are sent to the Super Game Boy.

Only the bits at positions 4 and 5 of `$FF00` are read by the Super Game Boy. These positions are often called P14 and P15. By setting both P14 and P15 to zero, the Super Game Boy starts listening to data, and then setting P14 to zero while P15 is one will send a 0, and setting P14 to one and P15 to zero will send a 1. The GingerBread function `SGBSendData` implements this, with reasonable delays to give the Super Game Boy time to read the messages. Basically, the transfer of data from the game to the Super Game Boy is a bit cumbersome and quite slow.

When sending larger amounts of data to the Super Game Boy, special commands allow the Super Game Boy to read graphics from the screen and interpret as data. This is used for sending border images, for example, which would take far too long to send over the P14/P15 channel.

Gingerbread contains functionality for the most common uses of the Super Game Boy, setting up color palettes for the game's display as well as presenting a border image around the game's display. These functions are made to try to hide the (rather complicated) details of how communication between the Game Boy and Super Nintendo works, and hopefully make these features relatively easy to implement.

5.1 Setting up SGB functionality

To add SGB functionality to Gingerbread, add the following line before including `gingerbread.asm`:

```
SGB_SUPPORT EQU 1
```

Note that setting to zero instead of one will still include the SGB support; to not include it, remove the line altogether (or comment it out).

This will make GingerBread include SGB functions. This takes up additional space on Bank 0, so if a game compiles fine without SGB support, but fails to compile with it, try moving code and assets to other banks. Also, on the topic of banks: Some SGB functionality depends on data and functions stored on Bank 1. So always change to Bank 1 before calling SGB function-

ality. If it is necessary to store data which should be sent to the SGB on some other bank, it could be copied into RAM before switching to Bank 1 (otherwise GingerBread has to be modified).

5.2 Palettes

There are no built-in functions in GingerBread specifically for the purpose of specifying the palettes to be used for the gameplay visuals on the SGB. Instead, it is necessary to write the SGB command binaries into the game, and send them to the SGB with the `SGBSendData` function.

The `SGBSendData` function assumes that the HL registers contains a memory address pointing to a table where the SGB command exists. Each SGB command is exactly 16 bytes long (some commands need to be split up into several commands, but `SGBSendData` does not automagically understand this, so it is then necessary to use it multiple times for each 16 bytes). The exact specification for all possible SGB commands (not limited to those used for defining the palettes) are available here: https://gbdev.gg8.se/wiki/articles/SGB_Function

The rest of this section will focus on some of the commands that are likely to be useful for specifying the palettes to be used during gameplay.

The palettes used to colorize the game visuals only apply to whole tiles of the screen (again, a tile means a non-overlapping 8×8 pixel square region) and this is regardless of what is displayed on those tiles. This means that the SGB, unlike the Game Boy Color, cannot give sprites different colors than backgrounds. Therefore, games on the SGB tend to not look very colorful, unless they have many static interface elements like health meters, score counters and so on. The palettes can change during the course of the game, but the process is too slow for, for example, dynamically changing the palettes wherever the main character is, to give it a different palette from the background.

There are four palettes which can be used to colorize the game window, numbered from zero to three. To define the colors in the palettes, use the `PAL01` and `PAL23` commands. `PAL01` defines palettes zero and one, while `PAL23` defines palettes two and three. An example of a `PAL01` command is as follows:

```
SGBPalettes01: ; Specifies the colors of palettes 0 and 1
DB %000000001 ; PAL01 command (%00000), length one (%001)
DB %111111111 ; Color 0 (for all palettes), %ggrrrrr
```

```

DB %01111111 ; Color 0 (for all palettes), %0bbbbbbg
DB %11100001 ; Color 1, Palette 0, %gggrrrrr
DB %01111001 ; Color 1, Palette 0, %0bbbbbbg
DB %01100001 ; Color 2, Palette 0, %gggrrrrr
DB %00110100 ; Color 2, Palette 0, %0bbbbbbg
DB %00000000 ; Color 3, Palette 0, %gggrrrrr
DB %00000000 ; Color 3, Palette 0, %0bbbbbbg
DB %11111111 ; Color 1, Palette 1, %gggrrrrr
DB %01111001 ; Color 1, Palette 1, %0bbbbbbg
DB %11100001 ; Color 2, Palette 1, %gggrrrrr
DB %00011101 ; Color 2, Palette 1, %0bbbbbbg
DB %00000000 ; Color 3, Palette 1, %gggrrrrr
DB %00000000 ; Color 3, Palette 1, %0bbbbbbg
DB 0 ; Not used

```

which can be used like so:

```

ld hl, SGBPalettes01
call SGBSendData

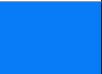
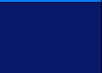
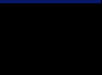


```

To help understanding the format of the colors, the following table contains the eight colors used above, in the Game Boy format as well as RGB15 color format (same as the Game Boy format, but in readable numbers) as well as roughly corresponding RGB24 (which is commonly used in computer games and the web)

Note that each byte can include multiple meanings, like the first one, of which the first five bits specify the command number (in this case 0, which is the code for PAL01) and the length of the command (one, meaning that we only send a set of 16 bytes once).

Also note that the color 0 only appears once: This is a limitation that all four palettes need to use the same color 0 (the one typically used for the white color).

One last thing to note is how the colors are stored. Each color uses two bytes, and should be read "from the bottom", so that the first bit is always a 0, followed by five bits of the blue channel (with the most significant bits coming first), followed by the two most significant bits of the green channel, and then on the next byte, the three least significant bits of the green channel, and finally the five bits of the red channel (again, the most significant bits come first). Each color channel (red, green and blue) uses five bits each, in contrast to most modern systems that typically use eight bits per color channel.

Description	Game Boy Format	RGB15	RGB24	Color
White	DB %11111111 DB %01111111	31-31-31 #7FFF	255-255-255 #FFFFFF	
Light blue	DB %11100001 DB %01111001	1-15-30 #79E1	8-123-247 #087BF7	
Dark blue	DB %01100001 DB %00110100	1-3-13 #3461	8-25-107 #08196B	
Black	DB %00000000 DB %00000000	0-0-0 #0000	0-0-0 #000000	
Pink	DB %11111111 DB %01111001	31-15-30 #79FF	255-123-247 #FF7B93	
Green	DB %11100001 DB %00011101	7-15-1 #1DE1	58-123-8 #3A7B08	

PAL23 is almost identical; the only things that needs to be changed is the first byte, which would be changed to %00001001 (command %00001, length %001).

By default, the entire gameplay screen is filled with palette 0. To use different palettes on different parts of the screen (for example, static visual elements like a health bar could have a different color scheme from the rest of the visuals), use the following commands: ATTR_BLK (which specifies a rectangular region with different palettes inside, outside and on the border of the rectangle), ATTR_LIN (which specifies the palettes on a horizontal or vertical line across the entire screen), ATTR_DIV (which splits the screen in two, either vertically or horizontally, with different palettes on the left/above the line, on the line, and to the right/below the line) or the ATTR_CHR (which sets the palette for only one tile).

An example of ATTR_BLK, which sets the entire screen to palette 2, is as follows:

```

SGBPal2Everywhere: ; Tells the SGB to use Palette 2 everywhere
; (used for title and game over)
DB %00100001 ; ATTR_BLK (%00100), length one (%001)
DB 1 ; Number of blocks we send
DB %00000100 ; Set the value "outside"
; the block (doing this with a small

```

```

; block means setting the entire screen)
DB %00101010 ; Which palettes to set
; inside (%10),
; on the border (%10)
; and outside (%10)
DB 0          ; X1 coordinate
DB 0          ; Y1 coordinate
DB 0          ; X2 coordinate
DB 0          ; Y2 coordinate
DB 0,0,0,0,0,0,0,0 ; Zero-padding

```

An example of ATTR_DIV, which sets palette 1 on the top line of the screen and palette 0 everywhere else, is as follows:

```

SGBPal01Div: ; Tells the SGB to draw the
; top horizontal line with palette 1,
; and palette 0 everywhere else
DB %00110001 ; ATTR_DIV command (%00110), length one (%001)
DB %01010100 ; Zero-padding (%0),
; horizontal split (%1),
; palette on the line (%01),
; palette above the line (%01),
; palette below the line (%00)
DB 0          ; Y-coordinate
DB 0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; Zero-padding

```

They are used as follows:

```

ld hl, SGBPal2Everywhere
call SGBSendData

```

and

```

ld hl, SGBPal01Div
call SGBSendData

```

respectively.

5.3 Border image

Gingerbread contains a Python script called `sgb_border.py` which can convert a .png image into assembler code that work with Gingerbread and RGBDS.

The script is written for Python 3 and requires Pillow, which is a commonly used image library for Python.

For it to work, the image must fulfill these rather strict requirements:

- The image must be exactly 256x256 pixels
- The image must contain a centered 160x144 black box, with true black (0,0,0 in RGB)
- Each tile (8x8 pixel block) cannot be unique; there must at most exist 256 unique tiles (including the black tiles in the middle).
- The colors are divided into four palettes of 16 colors each, and each palette includes the true black color. Each tile can only use colors from a single palette. The same color can exist in multiple palettes.

Basically, if the image does not pass through the script, use fewer unique tiles or fewer unique colors.

To use the SGB border created by the script, include it into your game with

```
INCLUDE "sgb_border.inc"
```

and then create function that looks something like:

SetupSGB:

```
SGBEarlyExit ; Without this, garbage
; would be visible on screen briefly
; when booting on a GB/GBC
```

```
call SGBFreeze ; To prevent "garbage"
; from being visible on screen
```

```
SGBBorderTransferMacro SGB_VRAM_TILEDATA1 ,
    SGB_VRAMTRANS_TILEDATA1 ,
    SGB_VRAMTRANS_GBTILEMAP
```

```
SGBBorderTransferMacro SGB_VRAM_TILEDATA2 ,
    SGB_VRAMTRANS_TILEDATA2 ,
    SGB_VRAMTRANS_GBTILEMAP
```

```
SGBBorderTransferMacro SGB_VRAM_TILEMAP ,
    SGB_VRAMTRANS_TILEMAP ,
```

SGB_VRAMTRANS_GB_TILEMAP

```
call SGBUnfreeze
ret
```

where the macros `SGBEarlyExit` and `SGBBorderTransferMacro`, and the functions `SGBFreeze` and `SGBUnfreeze` are defined by `GingerBread`, and the `SGB_*` names are defined by the code generated from the `sgb_border.py` script. `SGBEarlyExit` returns from the function if the player is not using SGB hardware. `SGBFreeze` freezes the graphics so that visual garbage, which is sent to the SGB, is not visible for the player, while `SGBUnfreeze` unfreezes the image so that the following graphical gameplay is visible. The `SGBBorderTransferMacro` takes care of presenting the border to the (frozen) screen such that the SGB can read it.

Chapter 6

Game Boy Color features

The Game Boy Color and Game Boy Advance (for the rest of this chapter, those models will be, somewhat incorrectly, collectively referred to as "Game Boy Color" or "GBC") are capable of colorizing games in a more detailed and natural manner than the Super Game Boy. Most importantly, the GBC has eight palettes for background tiles, and eight palettes for sprites. Since the palettes for tiles and sprites are separate, foreground objects can be more distinct from their backgrounds, something that is difficult to do on the SGB. Furthermore, the number of palettes allow for many more colors to be visible on screen at once.

The color palettes use the same format as they do on the SGB (see Chapter 5) but do note that the GBC mixes colors differently from the SGB so palettes copied from SGB code may need to be altered if they are to look similar.

In `GingerBread`, the `GBCApplyBackgroundPalettes` and `GBCApplySpritePalettes` functions can be used to set the palettes. They both take as input an address on HL, pointing to a table of color palettes, the starting position to write to on A and the number of bytes to write on B. Each color is two bytes (just like on the SGB) and there are four colors to each palette. So, for example, to write the first two palettes (palette 0 and 1), set A to 0 and B to 16. To write palette 5 only, set A to 40 and B to 8. Make sure HL points to a table of suitable length.

An example is

GBCBackgroundPalettes:

```
DB %11111111 ; Color 0, palette 0, %gggrrrrr
DB %01111111 ; Color 0, palette 0, %0bbbbbbg
DB %11100011 ; Color 1, palette 0, %gggrrrrr
DB %00011111 ; Color 1, palette 0, %0bbbbbbg
DB %11100001 ; Color 2, palette 0, %gggrrrrr
DB %00001001 ; Color 2, palette 0, %0bbbbbbg
DB %00000000 ; Color 3, palette 0, %gggrrrrr
DB %00000000 ; Color 3, palette 0, %0bbbbbbg
DB %11111111 ; Color 0, palette 1, %gggrrrrr
DB %01011110 ; Color 0, palette 1, %0bbbbbbg
DB %01111111 ; Color 1, palette 1, %gggrrrrr
DB %00001100 ; Color 1, palette 1, %0bbbbbbg
DB %01001111 ; Color 2, palette 1, %gggrrrrr
DB %00000100 ; Color 2, palette 1, %0bbbbbbg
DB %00000011 ; Color 3, palette 1, %gggrrrrr
DB %00000000 ; Color 3, palette 1, %0bbbbbbg
```

SetupGBC:

```
GBCEarlyExit ; No need to execute pointless code
; if we're not running on GBC

ld hl, GBCBackgroundPalettes
xor a ; Start at color 0, palette 0
ld b, 16 ; We have 16 bytes to write
call GBCApplyBackgroundPalettes
ret
```

The GBCEarlyExit macro, which does a ret if the game is not running on a GBC, is not technically necessary, as running GBC-specific commands on an older model does nothing. The colors in the above palettes are green-ish on palette 0 and red-ish on palette 1.

Sprite palettes work similarly, using GingerBread's GBCApplySpritePalettes function. The main difference is that color zero of every palette is ignored, as that color is always transparent.

Unlike the SGB, the GBC completely ignores the monochrome palette choices (see Chapter 3.3) and thus any game which is converted to work in GBC mode needs to add extra code to implement similar effects.

To choose which background tiles should use which palettes, use the memory address \$FF4F, called GBC_VRAM_BANK_SWITCH in GingerBread, to specify

SetupGBC:

```
GBCEarlyExit ; Needed to prevent garbage from
; being drawn on screen on GB

; Switch to VRAM bank 1 to write the palette map
; (which palette to use at which tile)
ld a, 1
ld [GBC_VRAM_BANK_SWITCH], a

; Copy the palette map onto VRAM
; (where the tile data usually is)
CopyRegionToVRAM 18, 20, GBCPaletteMap,
    BACKGROUND_MAPDATA_START

; Change the VRAM bank back
xor a
ld [GBC_VRAM_BANK_SWITCH], a
```

To choose which sprites should use which palettes, the last three bits of the "options" byte in the sprite table (see Chapter 3.10) are used to define which of the eight sprite palettes to use for each sprite.

Chapter 7

Practical aspects

7.1 Debugging in BGB

The emulator BGB has a competent debugger, which allows the execution of the game's code to be done in a more controlled fashion, executing code line by line, allowing each step to be controlled to find any errors, as well as setting break points, searching the code, and so on.

In order for the debugger to work optimally, a `.sym` file should exist next to the ROM file (the RGBDS compiler produces such file alongside ROMs). This file contains names of functions and routines, making the code more readable in the debugger.

The debugger contains a useful visualization tool, allowing the user to see the state of all registers and flags in each step. It also contains a green bar which measures CPU usage. This is useful not only to measure if there is additional CPU resources, but also to estimate battery usage.

There is also a VRAM viewer in BGB that visualizes graphics related memory, which is often useful. For example, it can be used to find which memory addresses tiles end up in when loaded.

7.2 Emulator versus real hardware

The BGB emulator is mostly accurate, but games should still be tested on real hardware. There have been some observed situations where the behaviour

differs.

BGB can emulate both GBC and SGB quite accurately, which is useful to test that the game runs as expected on all hardware. But it is a good idea to test games on real hardware occasionally because some differences do exist.

Chapter 8

Final notes

Hopefully, this book has given the reader a broad look at the topic of making Game Boy games in Assembler. For any topics not covered by this book, the included links should provide to good starting point for further research. Most importantly, the reader should (hopefully) be familiar enough with the technology and terminology to easily understand such materials.

I wish you the best of luck in all game development efforts.

Chapter 9

Version history

- Version 1.0: Initial release.
- Version 1.1: Fixed some typos and minor mistakes, as pointed out by Luc (u/loociano). Thanks!
- Version 1.2: Fixed errors in multiplication examples, charmaps. Removed some incorrect backslashes. Fixed Sections header, which ironically wasn't a Latex section. Thanks ISSOtm!
- Version 1.3: Fixed a minor error about xor usage. Thanks u/insta__mash! Also fixed a typo on page 17. Thanks kwdiggs!
- Version 1.4: Fixes several errors found by JL2210. Thanks!
- Version 1.5: Fixes some links, including one pointed out by JL2210. Thanks!