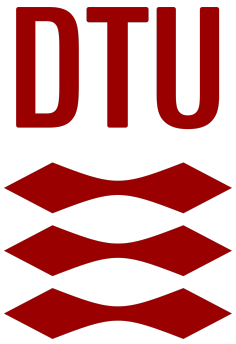
# Spaceship Project Report

Overengineering or good software patterns?




DTU Space
Technical University of Denmark
30010 Programming Project
22/01/2022

Group 8

Polly Clémentine Nielsen Boutet-Livoff - 214424

# 1 Abstract

In this work we document the development and technical details of designing and implementing a space shooter game. The game is developed for the STM32F302R8T6 Nucleo board in the C language. We utilize modern game programming patterns and a layered abstraction model in order to maintain a good project structure, and develop reusable API/HAL modules. We compromise between the limited processing power and bandwidth of a microcontroller and our creative ambitions to develop a complete project within the allotted time.

# 2 Resume

I dette værk dokumenterer vi udviklingen af et skyde spil og alle de tekniske detaljer som følger med det. Spillet er udviklet for STM32F302R8T6 Nucleo boardet i programmeringsproget C. Vi benytter os af moderne spilprogrammings design og API/HAL abstraktioner for at danne en overskuelige software struktur. Vores moduler er udviklet til at være afkoblet fra hindanden sådan at de er transportabel og kan benyttes i andre projekter. Vi finder en balance mellem det begrænset beregninskraft af mikrocontrolleren og vores kreative ambitioner for at udvikle et færdig spil indenfor den tid vi har.

# Contents

# 3 Introduction

I was tasked with creating a space shooter game on a STM32F302R8T6 Nucleo microcontroller. There are many problems to solve when creating a game, mostly related to what amounts to a good game. What would be engaging for a player? How advanced can we make the graphics? Should the game achieve realism or entertainment? But the limitations of the game running on a microcontroller with no OS introduces new and interesting problems. We don't have a game engine or even a graphics API. We have to strike a balance between developing tools to help us make the game (graphics API, parts of an engine) and making the game itself. I attempt to solve by using a few strong but simple game programming patterns while having a mocked up game from the start. Every API I implement is with a specific idea in mind, such that I don't waste time creating unneeded functionality. All the work in this report and the software was done by me (Polly), so I am omitting to elaborate on that in each section.

# 4 Specification requirements

I want my game to be built around pixel graphics. The terminal interface provides us with 16 colors and we can approximate 1:1 pixels. Within these visual limitation I created a mockup seen in figure 1, showing various game elements. The game should mainly consist of fighting enemy ships while dodging enemy fire, asteroids and black holes. All the game elements would react to the gravitation field. The player would have 3 weapons; rockets, a laser and bombs. Health, extra lives and ammunition would be collected from destroyed asteroids. Shield power ups would make you temporarily invulnerable, and audio through the buzzer. The LCD display would show your lives and points while the terminal view would show health and ammunition. Almost all of this was eventually scraped as requirements for an individual are less substantial than for groups.

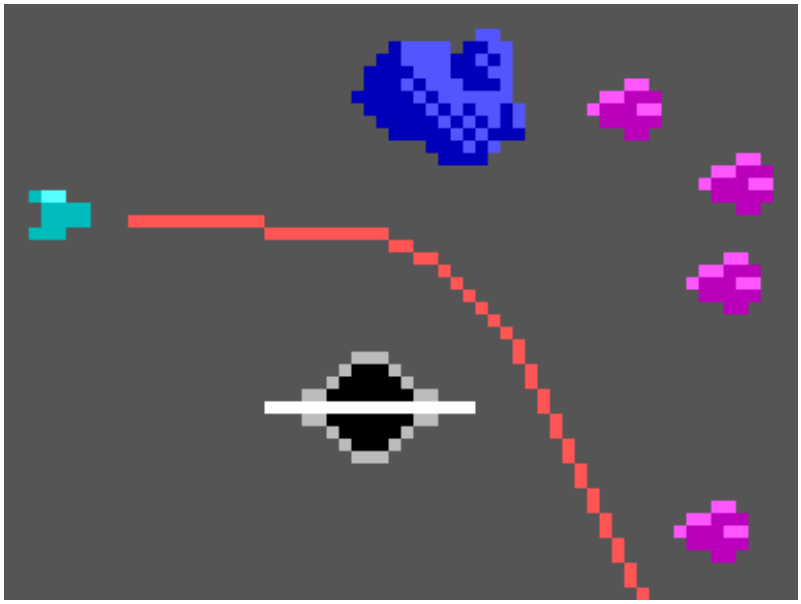


Figure 1: A mockup of the game showing the player, a black hole, an asteroid, a laser weapon bending under gravity, and 4 alien ships.

# 5 Project Structure

The project structure is divided into 3 layers, application (AP), application programming interface (API) and hardware abstraction layer (HAL). These offer different levels of abstraction and reusability. The block diagram shown in figure 2 shows a slightly simplified structure. Almost all of the application layer systems utilize the graphics and fixed point arithmetic modules but showing that would only clutter the diagram. The flowchart in figure 3 shows the main application including the game loop. This section documents functionality and design ideas. For exact types, function parameters, etc. please refer to the attached code listings.

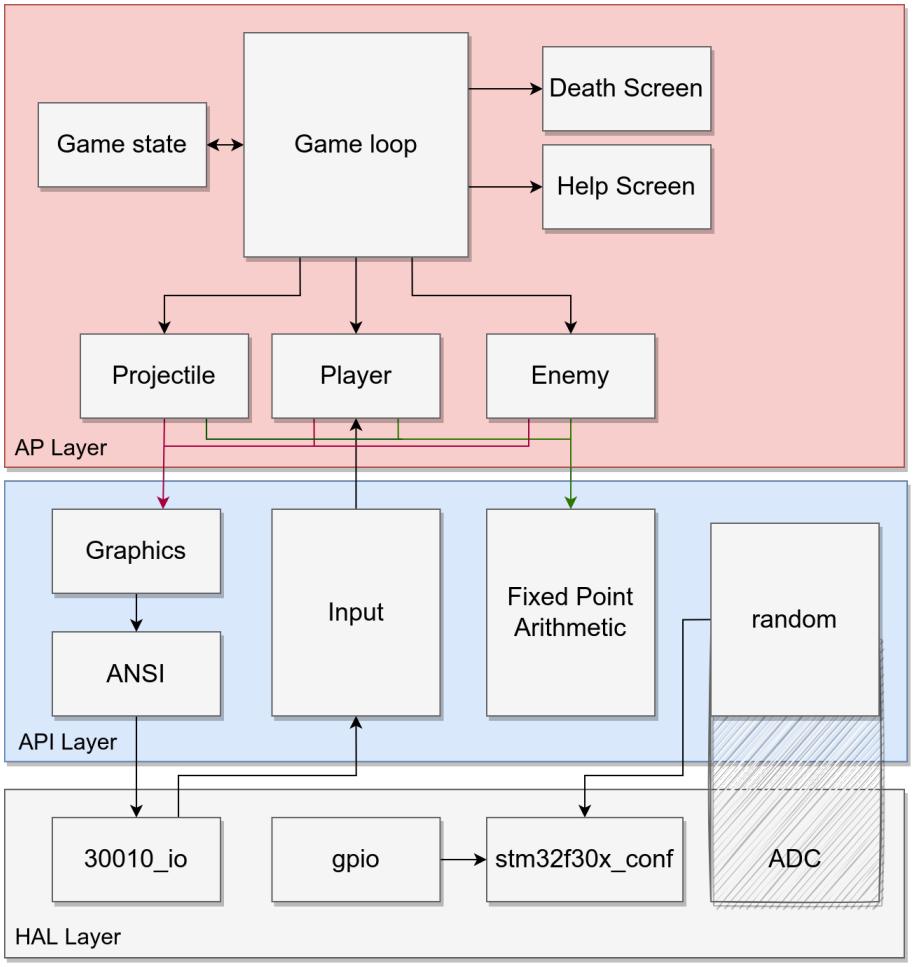



Figure 2: A block diagram highlighting the different layers of the software.

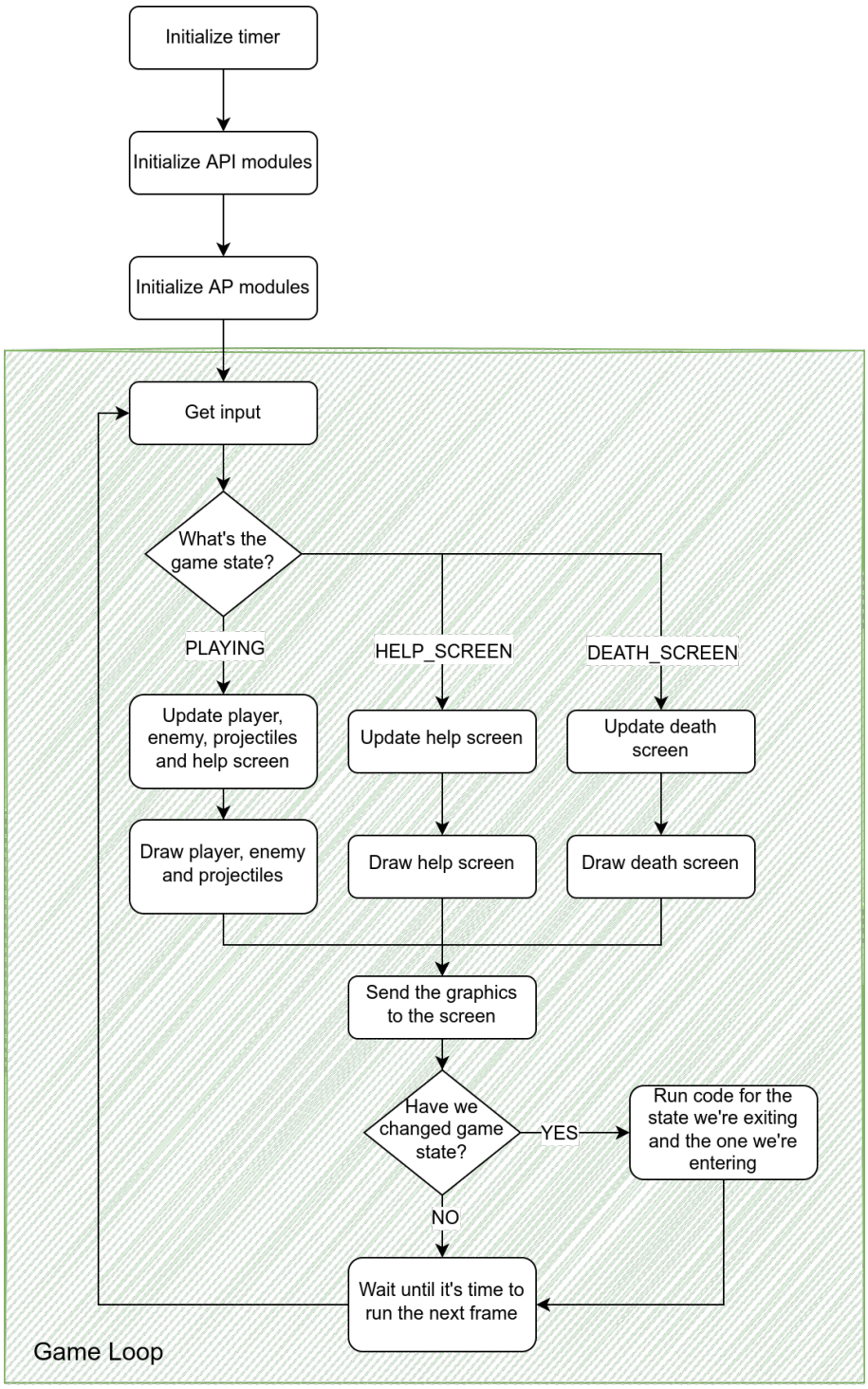



Figure 3: A flowchart of the main module.

## 5.1 Hardware Abstraction Layer

I should've added a timer module and an ADC module, but I chose not to for time. Additionally I had some strange issues with the timer's global variable `milliseconds` when trying to create a C header file for it. An easy future improvement to the project would be to create these two HAL modules. I have listed modules who exist only in the HAL domain, but some modules do interact directly with the hardware (namely main.c and random.c) and are not listed here.

### 5.1.1 GPIO

This module consists of `gpio.c` and `gpio.h`. It contains 2 functions for initializing a pin as in input pin or output pin (`set_input()`, `set_output()`) and 2 functions for reading a pin's value or writing to a pin's value (`read_input()`, `write_output()`). These 4 functions together form an easy way to interact with the microcontroller's individual pins. On top of this abstraction I have `init_joystick()`, which initialize the pins required for our joystick, and `read_joystick()`, which reads the value of each pin and constructs a 5 bit value describe the state of the joystick.

| Bit | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|
| Represents | Middle | Right | Left | Down | Up |

Table 1: Table describing the return value of read_joystick()

### 5.1.2 30010_io

This module adds functions to interact with UART and the LCD. I won't document the most of the functions as this module is provided by the course. However I have modified it to include a new function `uart_put_string(char* s)` which takes a string pointer terminated by a null byte and sends it over UART (excluding the null byte). I added this function as part of experimental optimizations where I avoided using printf, but the differences turned out to be negligible.

## 5.2 Application Programming Interface

### 5.2.1 ANSI

This module implements a few of the ANSI escape codes that exist for terminals and terminal emulators[3]. `clrscr()` clears the screen and returns the cursor to the top left corner. `home_cursor()` just returns the corner to the top left corner. `gotoxy(x, y)` moves the cursor to a coordinate on the screen. The top-left corner is $(1, 1)$ the x-axis grows to the right and the y-axis grows downwards. `hide_cursor()` makes the cursor invisible, which is useful if you don't want a visible cursor flying all over the screen when writing and overwriting text very fast. `set_color(color)` and `set_colors(fg, bg)` set the color the terminal writes text with. The parameters are one of the 16 ANSI color values and either foreground of background: 30-37, 40-47, 90-97 and 100-107. More detailed descriptions of the colors available can be found in the cited works[3].

### 5.2.2 Graphics

The graphics module does a lot of heavy lifting for the game. It's purpose is to translate between a 2d array of pixels into a string sent over UART consisting of ANSI escape codes that create an equivalent image in your terminal. The graphics module implements a 80 pixel wide, 64 pixel

tall screen and a refresh rate of 30 Hz, but this could easily by modified if desired. In order to achieve this we have a few functions, the first one being `graphics_init()` which sets up the UART connection, clears the screen, hides the cursor and returns a struct containing all the necessary data for the graphics system to function. `graphics_clear` clears the buffer. This is called before the beginning of every frame so that objects don't leave trails. `graphics_show` converts the pixel buffer into ANSI color codes and the byte value '\xDC'[1] which is the lower part of a half block. Since characters are roughly 2:1 (depending on the users' font and other settings) a block half that size is roughly 1:1. Additionally we can control the color of the bottom "pixel" by selecting the foreground color, and the top "pixel" by selecting the background color. Using this method it is possible to draw using roughly 1:1 pixels *and* double our resolution. `graphics_draw_sprite` draws a sprite at a specific position on screen. A sprite is a custom type containing a width, a height and an array of color values.

Sending bytes over UART has shown to be by far the most costly and slow part of the game, so I put in a decent amount of effort to optimize the code. If color changes aren't necessary from one pixel to the next we don't update it. I experimented with the baud rate, but couldn't get stable behavior over 2048000. I do not fully understand the details of UART communication so it is possible that you could achieve higher baud rates. Additionally I found late in development that it could be possible to "compress" repeated characters into one ANSI escape code command instead of sending them one by one. This could prove useful, but it would only reduce the size of very simple images (long same color horizontal lines). But this would only improve our best-case scenario, as `graphics_show` is very slow when dealing with images that change color many times (Changing color takes between 5 and 9 bytes). If we imagine a worst case lattice filling the entire 64 by 80 area we can estimate that it takes roughly $((9+1)\cdot 80+2)\cdot 64 = 51328 \approx 50000$ bytes per frame. Since we have 30 frames per second this is in the order of magnitude of 1 GB/s which is too much for our micro controller. I ran into this limitation as I was trying to implement Bayer matrix dithering[1, 5], which is often very similar to a lattice. One solution to this problem would be to only update the parts of the screen that have changed, or only update the screen when it changes. However both of these require complex code, and don't necessarily improve the worst-case scenario at all! Fundamentally UART and ANSI escape codes is not a suitable video interface, it is both too slow and too space inefficient. Most of this only became clear near the end of the project, so scraping it wasn't an option. Instead I tried to keep scenes visually simple so that the protocol could keep up.

### 5.2.3 Input

For the input API I was inspired by the Bevy[2] input API[3]. I wanted a function `is_down` that checked if a key was pressed and another function `just_pressed` that checked if a key had been pressed down this frame. This is useful as some actions should be repeated while a key is held (accelerating your ship), while others should only be performed once (we shouldn't shoot a new projectile every frame). The way I implemented this is by storing the state of all keys this frame an the previous frame. To check if a key is down, I simply check the state for the current frame. To check if it was just pressed, I check that it wasn't pressed last frame and is pressed this frame. `input_update` parses data received over UART like 'w' or "\x1b[A" (which means up arrow) and reads the joystick. It combines both of these sources into one input system so that players can

[1] For our project PuTTy was set up to use codepage 850, which maps the value 220 or 0xDC to a character that fills the bottom half of a cell.

[2] `https://bevyengine.org/`

[3] `https://docs.rs/bevy/0.6.0/bevy/input/struct.Input.html`

choose between using the keyboard or the joystick without requiring more complexity from the program layer. Lastly, `input_init` creates a new struct with the required data and initializes the joystick.

This system has some severe limitations using the keyboard input inherent to the terminal interface. If a user holds down a key, it is not repeated for many ( 500) milliseconds. This leaves the program completely unable to differentiate between a 1 ms press and a 499 ms press, which should have very different behavior! Additionally if a key is held down, for example A, and then another key is pressed even momentarily, for example B, the first key is not longer sent! The described scenario would send "A...AAAAAAAAAAB<no more keys>" even if A is held for the entire duration. This means we can never check if more than one key is pressed. These limitations make many type of games, especially action games, poor to play over a terminal interface. Luckily GPIO has no such issues, so I added support for that.

### 5.2.4 Fixed Point Arithmetic

The fixed point module defines the type `fixedpoint_t` and many functions and macros to utilize fixed point decimal numbers. This module also contains some more complex mathematical functionality. Fixed point numbers are defined to have 1 sign bit, 17 whole bits and 14 decimal bits. They are stored as a 32 bit signed integer. We can easily convert whole numbers into decimal numbers using `FP_FROM_WHOLE(n)`. To create fractional number we can divide, multiply and add numbers together. For example, to create 500/11 you can write `FP_FROM_WHOLE(500)/11`. The rest of the functions are very simple, so I will describe them in a table.

| Function | Description | Time complexity |
|---|---|---|
| `fp_mul(a, b)` | Returns the product of a and b | $O(1)$ |
| `fp_div(a, b)` | Returns a divided by b | $O(1)$ |
| `fp_sqrt(n)` | Returns the square root of n | $O(log(n))$ |
| `fp_round(n)` | Rounds the value to then nearest whole number | $O(1)$ |
| `fp_min(a, b)` | Returns the smallest of the two | $O(1)$ |
| `fp_max(a, b)` | Returns the biggest of the two | $O(1)$ |
| `fp_print(n)` | Prints the number in base 10 | - |

Table 2: Functions operating purely on fixedpoint_t.

The module also defines a 2-dimensional `vector_t` type containing a x-component and a y-component. `vector_get_length` calculates the length of a vector and `vector_set_length` scales a vector such that it has a specific length. `clamp_vector` limits a vector between two boundary vectors (although, that should probably be changed to be a `rectangle_t`), such that if it is outside the bounds it is moved to the nearest value within the bound. This type is used for positions, velocities and accelerations.

The last type it defines is `rectangle_t` which is a position, a width and a height. The rectangles can only be axis-aligned. This type is useful for collision boundaries or defining areas in the game world with specific functionality.

### 5.2.5 Pseudo Random Number Generation

I used a 64 bit version of xor-shift[2] as it is fast, uses little memory (64 bits) and is easy to implement. `random_init()` initializes the PRNGm but in order for the PRNG to function properly,

it requires a random seed. These can be hard to generate as computers are deterministic. Whoever I noticed that the least significant bits of the digized analog reading of the potentiometers were seemingly random (as long as the meters aren't at their maximum or minimum value). As I'm not using the randomness for any cryptographic or security applications I deemed this as a good enough source of randomness. To initialize the PRNG we read the least significant bit of the digitzed signal 64 times and concatenate it into the seed. This process is probably slow (I haven't measured) and requires at least one ADC, so we only do it once and then use xor-shift for later random values. The function `random_u64_up_to` takes an upper limit and returns an unsigned value below that bound. It is slightly biased towards low values, but the difference is very small and not significant for gameplay randomness. The function `random_i32_between` is similar but takes 2 bounds and the values can be negative.

## 5.3 Application

Most of the modules in the application layer follow the same pattern. They have an initialization function ending in `_init()` that initializes hardware required (if any) and creates the initial state of the system. This state is stored in a struct that ends with `_state_t`[4]. Each frame we first call a function ending in `_update()` to advance the simulation one frame and then `_draw()` to display the state of the system. Not all modules have all of these functions and most have other functionality also. The update pattern is taken from Game Programming Patterns[4] and lets the system work mostly independent of each other; We avoid "spaghetti code". Two outliers are `main.c` and `game_state.h` but they have good reasons. I will call these collections of data and code a system. They are similar to classes as they try to encapsulate complexity and only expose the necessary interface for other parts of the code to interact with it.

### 5.3.1 Game state

This module is just a header file containing an enum with 3 variants: PLAYING, DEATH_SCREEN and HELP_SCREEN. These 3 state help us decide what systems our game runs or doesn't run. Additionally it helps us transition from when state to another; when going from the PLAYING state to the DEATH_SCREEN state we need to initialize new systems[1].

### 5.3.2 main.c

The main module ties the program together. It runs all the initialization, keeps track of some essential state and contains the game loop[4]. The game loop in its simplest terms reads input, updates the world (according to that input), renders the world and then waits until the next frame has to be created. Additionally the game loop decides which system to run based on the current game state, and the systems themselves can modify the game state. We want this module to remain pretty simple as it is only here to tie together the different modules we've created. The module also contains some debugging functionality which can be enabled at compile-time, and the timer functionality. Ideally these would put into their own modules, although currently they are not.

[4]Here state means information that is remembered, not state as in state machine. This is confusing as we later use state as in state machine.

### 5.3.3 Player

The player consists of a position, velocity, sprite, health and whether they're alive. These are initialized to defaults when `player_init` is called. `player_update` does a few thing: The position is changed with regard to the velocity, and the velocity is changed with regard to the acceleration, which in turn is defined by the input. Health is decreased by the projectile system and we are marked as dead if it drops to 0. If the player is dead we change the game state to DEATH_SCREEN. When the player presses the space key or presses the joystick down we launch a projectile using the `projectiles_add` function. The player has friction to make controlling the ship easier. `player_draw` draws the player sprite at the current position.

### 5.3.4 Enemy

The enemy system is very similar to the player system, except there can be multiple enemies and instead of user input it has a simple AI. The `enemy_update` function contains almost all of the functionality. We randomly spawn new enemies if there are less than 5. For each enemy we update their position according to their velocity and their velocity according to their acceleration. Their acceleration is a combination of friction and the controls the AI chooses. The AI is a state machine[4] with 4 states: Approaching, shooting, idle and fleeing. We can see a state diagram for it in figure 4. In the approaching state, the AI moves towards the player, and away from the player in the fleeing state. In the shooting state the AI fire projectiles towards the player. The idle state does nothing other than wait. The state changes with regard to the previous state after a random time range, or if the enemy takes too much damage.

| Function | Description |
|---|---|
| `enemy_init` | Creates an empty enemy system |
| `enemy_update` | Updates enemies (see text for details). |
| `enemy_draw` | Draws each enemy at their position |
| `enemy_add` | Adds an enemy to the system |
| `enemy_remove` | Remove an enemy from the system |
| `enemy_handle_damage` | Updates health and AI |

Table 3: Functions in the projectile module.

### 5.3.5 Projectiles

The projectile system has an array of every projectile and each projectile contains a position and a velocity. We check whether any projectile has collided with an enemy ship or the player ship, and decrement their health if that's the case. The projectiles also have a color which is used to avoid friendly fire and self damage. To draw the projectiles we simply color the pixel the projectile is at.

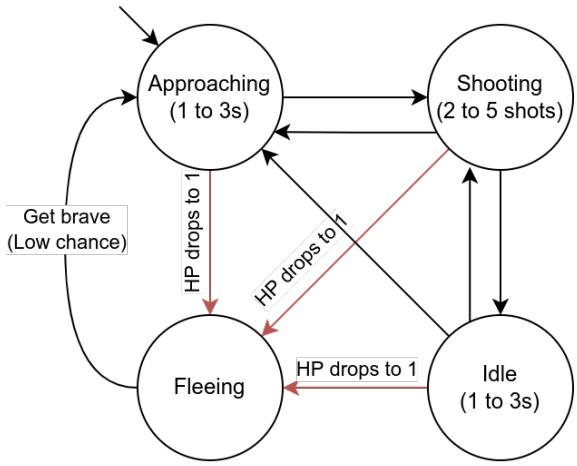



Figure 4: State diagram for the finite state machine AI.

| Function | Description |
|---|---|
| `projectiles_init` | Creates an empty projectile system |
| `projectiles_update` | Moves all projectiles and checks for collisions |
| `projectiles_draw` | Draws each projectile |
| `projectiles_add` | Adds a projectile to the system |
| `projectiles_remove` | Remove a projectile from the projectile system |

Table 4: Functions in the projectile module.

### 5.3.6 Death Screen

The death screen, or game over screen is shown when the player dies. This screen shows a neon-sign that spells ”GAME OVER” slowly blinking on, and flickering sporadically. When the game state is changed to DEATH_SCREEN we initialize the system, which randomizes when the letters turn on and when they will blink. Each letter consists of a sprite, an integer representing how long until the letter turns on, another representing how long until it blinks and a last one representing the amount of blinks. After a letter is finished blinking it will choose a new amount of time until it blinks.

| Function | Description |
|---|---|
| `death_screen_enter` | Creates a randomized death screen system |
| `death_screen_update` | Decrements all the timers and randomizes them if needed |
| `death_screen_draw` | Draws each letter, unless it is off/blinking |
| `letter_is_on` | Checks if a letter is on |

Table 5: Functions in the death screen module.

### 5.3.7 Help Screen

The final game state HELP_SCREEN simply turns of the normal graphics engine. Instead `help_screen_draw()` writes text describing how to play the game in the middle of the screen. `help_screen_update` simply checks if you press H to change you between PLAYING and HELP_SCREEN. Changing back into PLAYING re-enables the normal graphics engine and allows the game to continue as normal. Changing the game state like this also pauses the game as we don't call any relevant `_update` functions.

# 6 Project verification

I did not use any automatic software testing methods such as unit or integration tests as I believe for a project of this size, the time required to learn and/or develop the software would be far greater than the utility it would offer. While developing this project I mostly relied on testing by playing. I had expectations to what the software should do and tested them by intentionally playing in such a way to go though the different code paths that I've written. If there were discrepancies in what the software did I would use the provided debugger to step through the program and notice when it behaved differently than my mental model. If the provided debugger was too slow or didn't provide the insight I needed, (for example, bugs that occur over many frames) I would develop debug tools. One early tool I developed displayed my how many milliseconds of computation time the software had after finishing a frame. If didn't have this my program might start to dip below 30 frames per second without me noticing, this would in turn change the speed of the game in a uncontrolled way and introduce lag. I was able to increase the UART baud rate before this became an issue thanks to this debug tool. The tools can be enabled or disabled using the `DEBUG_GAME_INFO` macro. They all display information about the internals of the game. These methods can be error prone (since they rely heavily on the programmers' capabilities), but they were sufficient for a project of this size.

# 7 Project plan

I was originally very ambitious as I had planned to do the work of a group alone. After talking to my professor I decreased my ambitions by a lot, since the expectations were also lower for an individual.

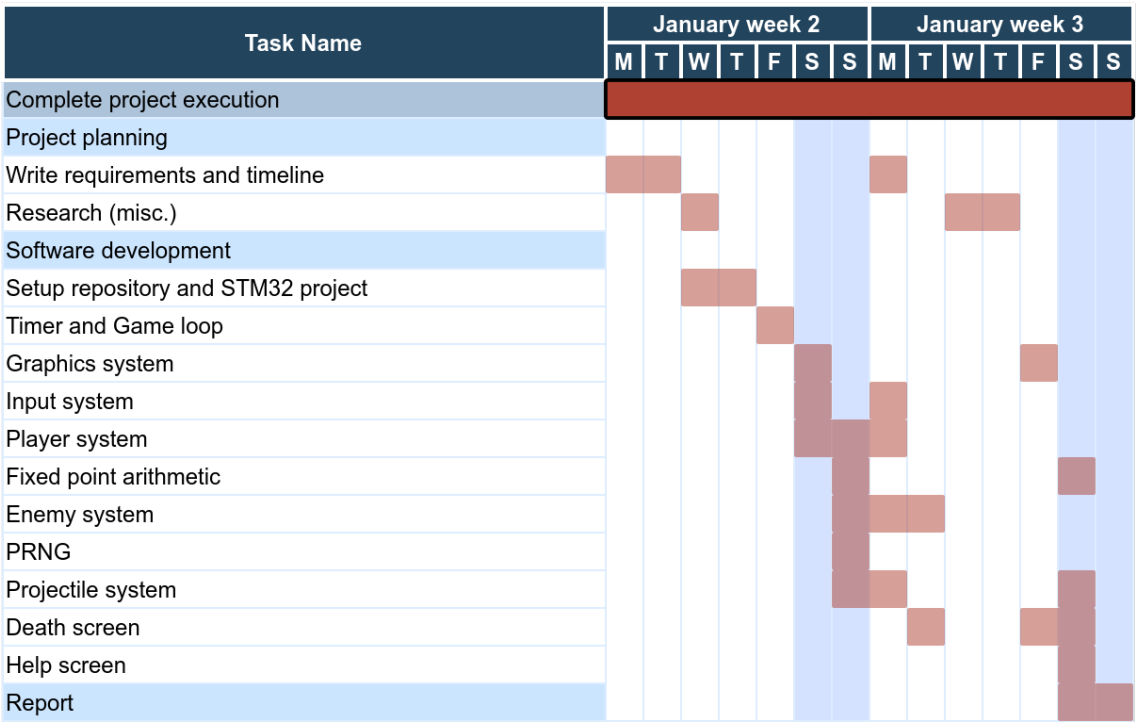



Figure 5: Gantt chart of the project.

# 8 User manual

The game visuals are shown over UART on PuTTy. To correctly display the game set the baud rate to 2048000, set the remote character set to ”CP850”. Make sure there are at least 80 collumns and 32 rows visible in your terminal. After connecting reset the microcontroller. Press H to toggle the help menu, W, A, S and D to move the ship and press the space bar to shoot a projectile. Alternatively you can use joystick to move the ship and press down on it to shoot a projectile. The objective is to survive for as long as possible. Good luck!

# 9 Conclusion

We developed a simple but bug free game with the fundamental requirements fulfilled. The program is structured well and could be easily extended to include additional features, such as the ones that were scraped for time. I think I avoided overengineering the game engine part of the program by sticking to simple game programming patterns and only implementing the API features I required. Lack of prior knowledge with the C language and compiler left me wasting a lot of time on compiler quirks. Later projects would benefit from my now improved knowledge of the language and tooling. Additionally, I spent a lot of time on research into ANSI escape codes and their limitations, only to find that there were no significantly better solutions to my problems.

# 10 Bibliography

## References


[1] Bruce Bayer. "An Optimum Method For Two-level Rendition Of Continuous-tone Pictures". In: *IEEE International Conference on Communications* 1 (1973), pp. 11–15.

[2] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6. DOI: 10.18637/jss.v008.i14.

[3] Edward Moy. *XTerm Control Sequences*. 1994. URL: https://invisible-island.net/xterm/ctlseqs/ctlseqs.html (visited on 01/22/2022).

[4] Robert Nystrom. *Game Programming Patterns*. Genever | Benning, 2014. ISBN: 9780990582915.

[5] Sean Riley. *Ordered Dithering*. 2019. URL: https://www.youtube.com/watch?v=IviNO7iICTM (visited on 01/22/2022).

# A Program code

## A.1 gpio.h

```
#ifndef _GPIO_H_
#define _GPIO_H_

#include "stm32f30x_conf.h"

// Turns the GPIO pin into an input
void set_input(GPIO_TypeDef *gpio, char pin);

char read_input(GPIO_TypeDef *gpio, char pin);

// Turns the GPIO pin into an output
void set_output(GPIO_TypeDef *gpio, char pin);

void write_output(GPIO_TypeDef *gpio, char pin, char value);

// Order is Middle, Right, Left, Down, Up
// 0bMRLDU
char read_joystick();

void init_joystick();

#endif /* _GPIO_H_ */
```

## A.2 gpio.c

```
#include "stm32f30x_conf.h"

// Turns the GPIO pin into an input
void set_input(GPIO_TypeDef *gpio, char pin) {
    gpio->MODER &= ~(0x00000003 << (pin * 2)); // Clear mode register
    gpio->MODER |= (0x00000000 << (pin * 2)); // Set mode register (0x00 - Input, 0x01 - Output,
    ↪  0x02 - Alternate Function, 0x03 - Analog in/out)
    gpio->PUPDR &= ~(0x00000003 << (pin * 2)); // Clear push/pull register
    gpio->PUPDR |= (0x00000002 << (pin * 2)); // Set push/pull register (0x00 - No pull, 0x01 -
    ↪  Pull-up, 0x02 - Pull-down)
}

char read_input(GPIO_TypeDef *gpio, char pin) {
    return (gpio->IDR >> pin) & 1;
}

// Turns the GPIO pin into an output
void set_output(GPIO_TypeDef *gpio, char pin) {
    gpio->OSPEEDR &= ~(0x00000003 << (pin * 2)); // Clear speed register
    gpio->OSPEEDR |= (0x00000002 << (pin * 2)); // set speed register (0x01 - 10MHz, 0x02 - 2
    ↪  MHz, 0x03 - 50 MHz)
    gpio->OTYPER &= ~(0x0001 << (pin * 1)); // Clear output type register
    gpio->OTYPER |= (0x0000 << (pin * 1)); // Set output type register (0x00 - Push pull, 0x01 -
    ↪  Open drain)
    gpio->MODER &= ~(0x00000003 << (pin * 2)); // Clear mode register
    gpio->MODER |= (0x00000001 << (pin * 2)); // Set mode register (0x00 - Input, 0x01 - Output,
    ↪  0x02 - Alternate Function, 0x03 - Analog in/out)
}

void write_output(GPIO_TypeDef *gpio, char pin, char value) {
    if (value) {
```

```
        gpio->ODR |= (1 << pin);
    } else {
        gpio->ODR &= ~(1 << pin);
    }
}

// Order is Center, Right, Left, Down, Up
// ObRLDU

// UP LEFT DOWN RIGHT
char read_joystick() {
    return 0
    | (read_input(GPIOB, 5) << 4)
    | (read_input(GPIOA, 4) << 3)
    | (read_input(GPIOC, 1) << 2)
    | (read_input(GPIOB, 0) << 1)
    | (read_input(GPIOC, 0) << 0);
}

void init_joystick() {
    RCC->AHBENR |= RCC_AHBPeriph_GPIOA; // Enable clock for GPIO Port A
    RCC->AHBENR |= RCC_AHBPeriph_GPIOB; // Enable clock for GPIO Port B
    RCC->AHBENR |= RCC_AHBPeriph_GPIOC; // Enable clock for GPIO Port C

    // Joystick
    // UP     => A2 => PA_4
    // DOWN   => A3 => PB_0
    // LEFT   => A4 => PC_1
    // RIGHT  => A5 => PC_0
    // CENTER => D5 => PB_5
    set_input(GPIOA, 4);
    set_input(GPIOB, 0);
    set_input(GPIOC, 1);
    set_input(GPIOC, 0);
    set_input(GPIOB, 5);
}
```

### A.3 30010_io.h

```
#ifndef _30010_IO_H_
#define _30010_IO_H_

/* Includes ------------------------------------------------------------------*/
#include "stm32f30x_conf.h"
#include <stdio.h>

/* Exported types ------------------------------------------------------------*/
/* Exported constants --------------------------------------------------------*/
/* Exported macro ------------------------------------------------------------*/
#define UART_BUFFER_LENGTH 256

/* Exported functions ------------------------------------------------------- */
/*****************************/
/*** USB Serial Functions ***/
/*****************************/
void uart_init(uint32_t baud);
void uart_put_char(uint8_t c);
void uart_put_string(char* s);
uint8_t uart_get_char();
```

```
uint8_t uart_get_count();
void uart_clear();

/*****************************/
/*** LCD Control Functions ***/
/*****************************/
void lcd_init();
void lcd_transmit_byte(uint8_t data);
void lcd_push_buffer(uint8_t* buffer);
void lcd_reset();

#endif /* _30010_IO_H_ */
```

### A.4 30010_io.c

```
#include "30010_io.h"

/*****************************/
/*** USB Serial Functions ***/
/*****************************/
volatile uint8_t UART_BUFFER[UART_BUFFER_LENGTH] = {0};
volatile uint8_t UART_END_IDX = 0;
volatile uint8_t UART_START_IDX = 0;
volatile uint8_t UART_COUNT = 0;

uint8_t uart_get_char(){
    uint8_t val = 0;
    if (UART_COUNT > 0) {
        val = UART_BUFFER[UART_START_IDX++];
        UART_COUNT--;
    }
    return val;
}

void uart_put_char(uint8_t c) {
    USART_SendData(USART2, (uint8_t)c);
    while(USART_GetFlagStatus(USART2, USART_FLAG_TXE)  == RESET){}
}

void uart_put_string(char* s) {
  while (*s != 0) {
    uart_put_char(*s);
    s++;
  }
}

int _write_r(struct _reent *r, int file, char *ptr, int len) {
    int n;

    for (n = 0; n < len; n++) {
        if (ptr[n] == '\n') {
            uart_put_char('\r');
        }
        uart_put_char(ptr[n] & (uint16_t)0x01FF);
    }

    return len;
}
```

```
void USART2_IRQHandler(void)
{
    if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
    {
        UART_BUFFER[UART_END_IDX++] = (uint8_t)(USART2->RDR & 0xFF);
        if (UART_COUNT == UART_BUFFER_LENGTH-1){
            UART_START_IDX++;
        } else {
            UART_COUNT++;
        }
    }
}

void uart_clear(){
    UART_START_IDX = 0;
    UART_END_IDX = 0;
    UART_COUNT = 0;
}

uint8_t uart_get_count(){
    return UART_COUNT;
}

void uart_init(uint32_t baud) {
    setbuf(stdout, NULL); // Set stdout to disable line buffering
    setbuf(stdin,  NULL); // Set stdin  to disable line buffering

    // Enable Clocks
    RCC->AHBENR  |= RCC_AHBPeriph_GPIOA;    // Enable Clock for GPIO Bank A
    RCC->APB1ENR |= RCC_APB1Periph_USART2;  // Enable Clock for USART2

    // Connect pins to USART2
    GPIOA->AFR[2 >> 0x03] &= ~(0x0000000F << ((2 & 0x00000007) * 4)); // Clear alternate function
    ↪  for PA2
    GPIOA->AFR[2 >> 0x03] |=  (0x00000007 << ((2 & 0x00000007) * 4)); // Set alternate 7 function
    ↪  for PA2
    GPIOA->AFR[3 >> 0x03] &= ~(0x0000000F << ((3 & 0x00000007) * 4)); // Clear alternate function
    ↪  for PA3
    GPIOA->AFR[3 >> 0x03] |=  (0x00000007 << ((3 & 0x00000007) * 4)); // Set alternate 7 function
    ↪  for PA3

    // Configure pins PA2 and PA3 for 10 MHz alternate function
    GPIOA->OSPEEDR &= ~(0x00000003 << (2 * 2) | 0x00000003 << (3 * 2));    // Clear speed
    ↪  register
    GPIOA->OSPEEDR |=  (0x00000001 << (2 * 2) | 0x00000001 << (3 * 2));    // set speed register
    ↪  (0x01 - 10 MHz, 0x02 - 2 MHz, 0x03 - 50 MHz)
    GPIOA->OTYPER  &= ~(0x0001     << (2)     | 0x0001     << (3));        // Clear output type
    ↪  register
    GPIOA->OTYPER  |=  (0x0000     << (2)     | 0x0000     << (3));        // Set output type
    ↪  register (0x00 - Push pull, 0x01 - Open drain)
    GPIOA->MODER   &= ~(0x00000003 << (2 * 2) | 0x00000003 << (3 * 2));    // Clear mode register
    GPIOA->MODER   |=  (0x00000002 << (2 * 2) | 0x00000002 << (3 * 2));    // Set mode register
    ↪  (0x00 - Input, 0x01 - Output, 0x02 - Alternate Function, 0x03 - Analog in/out)
    GPIOA->PUPDR   &= ~(0x00000003 << (2 * 2) | 0x00000003 << (3 * 2));    // Clear push/pull
    ↪  register
    GPIOA->PUPDR   |=  (0x00000001 << (2 * 2) | 0x00000001 << (3 * 2));    // Set push/pull
    ↪  register (0x00 - No pull, 0x01 - Pull-up, 0x02 - Pull-down)

    //Configure USART2
```

```
    USART2->CR1 &= ~0x00000001; // Disable USART2
    USART2->CR2 &= ~0x00003000; // Clear CR2 Configuration
    USART2->CR2 |=  0x00000000; // Set 1 stop bits
    USART2->CR1 &= ~(0x00001000 | 0x00000400 | 0x00000200 | 0x00000008 | 0x00000004); // Clear
    ↪  CR1 Configuration
    USART2->CR1 |=  0x00000000; // Set word length to 8 bits
    USART2->CR1 |=  0x00000000; // Set parity bits to none
    USART2->CR1 |=  0x00000004 | 0x00000008; // Set mode to RX and TX
    USART2->CR3 &= ~(0x00000100 | 0x00000200); // Clear CR3 Configuration
    USART2->CR3 |=  0x00000000; // Set hardware flow control to none

    uint32_t divider = 0, apbclock = 0, tmpreg = 0;
    RCC_ClocksTypeDef RCC_ClocksStatus;
    RCC_GetClocksFreq(&RCC_ClocksStatus); // Get USART2 Clock frequency
    apbclock = RCC_ClocksStatus.USART2CLK_Frequency;

    if ((USART2->CR1 & 0x00008000) != 0) {
      // (divider * 10) computing in case Oversampling mode is 8 Samples
      divider = (2 * apbclock) / baud;
      tmpreg  = (2 * apbclock) % baud;
    } else {
      // (divider * 10) computing in case Oversampling mode is 16 Samples
      divider = apbclock / baud;
      tmpreg  = apbclock % baud;
    }

    if (tmpreg >=  baud / 2) {
        divider++;
    }

    if ((USART2->CR1 & 0x00008000) != 0) {
        // get the LSB of divider and shift it to the right by 1 bit
        tmpreg = (divider & (uint16_t)0x000F) >> 1;
        // update the divider value
        divider = (divider & (uint16_t)0xFFF0) | tmpreg;
    }

    USART2->BRR = (uint16_t)divider; // Configure baud rate
    USART2->CR1 |= 0x00000001; // Enable USART2

    USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);
    NVIC_EnableIRQ(USART2_IRQn);
}

/*****************************/
/*** LCD Control Functions ***/
/*****************************/
void lcd_transmit_byte(uint8_t data) {
    GPIOB->ODR &= ~(0x0001 << 6); // CS = 0 - Start Transmission
    while(SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_TXE) != SET) { }
    SPI_SendData8(SPI2, data);
    while(SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_TXE) != SET) { }
    GPIOB->ODR |=  (0x0001 << 6); // CS = 1 - End Transmission
}

void lcd_push_buffer(uint8_t* buffer)
{
    int i = 0;
```

```
    //page 0
    GPIOA->ODR &= ~(0x0001 << 8); // A0 = 0 - Set Command
    lcd_transmit_byte(0x00);      // set column low nibble 0
    lcd_transmit_byte(0x10);      // set column hi  nibble 0
    lcd_transmit_byte(0xB0);      // set page address  0

    GPIOA->ODR |=  (0x0001 << 8); // A0 = 1 - Set Data
    for(i=0; i<128; i++) {
       lcd_transmit_byte(buffer[i]);
    }

    // page 1
    GPIOA->ODR &= ~(0x0001 << 8); // A0 = 0 - Set Command
    lcd_transmit_byte(0x00);      // set column low nibble 0
    lcd_transmit_byte(0x10);      // set column hi  nibble 0
    lcd_transmit_byte(0xB1);      // set page address  1

    GPIOA->ODR |=  (0x0001 << 8); // A0 = 1 - Set Data
    for( i = 128 ; i < 256 ; i++ ) {
       lcd_transmit_byte(buffer[i]);
    }

    //page 2
    GPIOA->ODR &= ~(0x0001 << 8); // A0 = 0 - Set Command
    lcd_transmit_byte(0x00);      // set column low nibble 0
    lcd_transmit_byte(0x10);      // set column hi  nibble 0
    lcd_transmit_byte(0xB2);      // set page address  2

    GPIOA->ODR |=  (0x0001 << 8); // A0 = 1 - Set Data
    for(i=256; i<384; i++) {
       lcd_transmit_byte(buffer[i]);
    }

    //page 3
    GPIOA->ODR &= ~(0x0001 << 8); // A0 = 0 - Set Command
    lcd_transmit_byte(0x00);      // set column low nibble 0
    lcd_transmit_byte(0x10);      // set column hi  nibble 0
    lcd_transmit_byte(0xB3);      // set page address  3

    GPIOA->ODR |=  (0x0001 << 8); // A0 = 1 - Set Data
    for(i=384; i<512; i++) {
       lcd_transmit_byte(buffer[i]);
    }
}

void lcd_reset()
{
    GPIOA->ODR &= ~(0x0001 << 8); // A0 = 0 - Reset Command/Data
    GPIOB->ODR |=  (0x0001 << 6); // CS = 1 - Reset C/S

    GPIOB->ODR &= ~(0x0001 << 14); // RESET = 0 - Reset Display
    for (uint32_t i = 0 ; i < 4680 ; i++) { asm("nop"); }; // Wait
    GPIOB->ODR |=  (0x0001 << 14); // RESET = 1 - Stop Reset
    for (uint32_t i = 0 ; i < 390000 ; i++) { asm("nop"); }; // Wait

    // Configure Display
    GPIOA->ODR &= ~(0x0001 << 8); // A0 = 0 - Set Command

    lcd_transmit_byte(0xAE);  // Turn off display
```

```
    lcd_transmit_byte(0xA2);  // Set bias voltage to 1/9

    lcd_transmit_byte(0xA0);  // Set display RAM address normal
    lcd_transmit_byte(0xC8);  // Set update direction

    lcd_transmit_byte(0x22);  // Set internal resistor ratio
    lcd_transmit_byte(0x2F);  // Set operating mode
    lcd_transmit_byte(0x40);  // Set start line address

    lcd_transmit_byte(0xAF);  // Turn on display

    lcd_transmit_byte(0x81);  // Set output voltage
    lcd_transmit_byte(0x17);  // Set contrast

    lcd_transmit_byte(0xA6);  // Set normal mode
}

void lcd_init() {
    // Enable Clocks
    RCC->AHBENR  |= 0x00020000 | 0x00040000;    // Enable Clock for GPIO Banks A and B
    RCC->APB1ENR |= 0x00004000;                 // Enable Clock for SPI2

    // Connect pins to SPI2
    GPIOB->AFR[13 >> 0x03] &= ~(0x0000000F << ((13 & 0x00000007) * 4)); // Clear alternate
    ↪ function for PB13
    GPIOB->AFR[13 >> 0x03] |=  (0x00000005 << ((13 & 0x00000007) * 4)); // Set alternate 5
    ↪ function for PB13 - SCLK
    GPIOB->AFR[15 >> 0x03] &= ~(0x0000000F << ((15 & 0x00000007) * 4)); // Clear alternate
    ↪ function for PB15
    GPIOB->AFR[15 >> 0x03] |=  (0x00000005 << ((15 & 0x00000007) * 4)); // Set alternate 5
    ↪ function for PB15 - MOSI

    // Configure pins PB13 and PB15 for 10 MHz alternate function
    GPIOB->OSPEEDR &= ~(0x00000003 << (13 * 2) | 0x00000003 << (15 * 2));    // Clear speed
    ↪ register
    GPIOB->OSPEEDR |=  (0x00000001 << (13 * 2) | 0x00000001 << (15 * 2));    // set speed
    ↪ register (0x01 - 10 MHz, 0x02 - 2 MHz, 0x03 - 50 MHz)
    GPIOB->OTYPER  &= ~(0x0001     << (13)     | 0x0001     << (15));        // Clear output type
    ↪ register
    GPIOB->OTYPER  |=  (0x0000     << (13)     | 0x0000     << (15));        // Set output type
    ↪ register (0x00 - Push pull, 0x01 - Open drain)
    GPIOB->MODER   &= ~(0x00000003 << (13 * 2) | 0x00000003 << (15 * 2));    // Clear mode
    ↪ register
    GPIOB->MODER   |=  (0x00000002 << (13 * 2) | 0x00000002 << (15 * 2));    // Set mode register
    ↪ (0x00 - Input, 0x01 - Output, 0x02 - Alternate Function, 0x03 - Analog in/out)
    GPIOB->PUPDR   &= ~(0x00000003 << (13 * 2) | 0x00000003 << (15 * 2));    // Clear push/pull
    ↪ register
    GPIOB->PUPDR   |=  (0x00000000 << (13 * 2) | 0x00000000 << (15 * 2));    // Set push/pull
    ↪ register (0x00 - No pull, 0x01 - Pull-up, 0x02 - Pull-down)

    // Initialize REEST, nCS, and A0
    // Configure pins PB6 and PB14 for 10 MHz output
    GPIOB->OSPEEDR &= ~(0x00000003 << (6 * 2) | 0x00000003 << (14 * 2));    // Clear speed
    ↪ register
    GPIOB->OSPEEDR |=  (0x00000001 << (6 * 2) | 0x00000001 << (14 * 2));    // set speed register
    ↪ (0x01 - 10 MHz, 0x02 - 2 MHz, 0x03 - 50 MHz)
    GPIOB->OTYPER  &= ~(0x0001     << (6)      | 0x0001     << (14));        // Clear output type
    ↪ register
```

```
    GPIOB->OTYPER  |=  (0x0000     << (6)     | 0x0000     << (14));        // Set output type
    ↪  register (0x00 - Push pull, 0x01 - Open drain)
    GPIOB->MODER   &= ~(0x00000003 << (6 * 2) | 0x00000003 << (14 * 2));    // Clear mode
    ↪  register
    GPIOB->MODER   |=  (0x00000001 << (6 * 2) | 0x00000001 << (14 * 2));    // Set mode register
    ↪  (0x00 - Input, 0x01 - Output, 0x02 - Alternate Function, 0x03 - Analog in/out)
    GPIOB->PUPDR   &= ~(0x00000003 << (6 * 2) | 0x00000003 << (14 * 2));    // Clear push/pull
    ↪  register
    GPIOB->PUPDR   |=  (0x00000000 << (6 * 2) | 0x00000000 << (14 * 2));    // Set push/pull
    ↪  register (0x00 - No pull, 0x01 - Pull-up, 0x02 - Pull-down)
    // Configure pin PA8 for 10 MHz output
    GPIOA->OSPEEDR &= ~0x00000003 << (8 * 2);    // Clear speed register
    GPIOA->OSPEEDR |=  0x00000001 << (8 * 2);    // set speed register (0x01 - 10 MHz, 0x02 - 2
    ↪  MHz, 0x03 - 50 MHz)
    GPIOA->OTYPER  &= ~0x0001     << (8);        // Clear output type register
    GPIOA->OTYPER  |=  0x0000     << (8);        // Set output type register (0x00 - Push pull,
    ↪  0x01 - Open drain)


    GPIOA->MODER   &= ~0x00000003 << (8 * 2);    // Clear mode register
    GPIOA->MODER   |=  0x00000001 << (8 * 2);    // Set mode register (0x00 - Input, 0x01 -
    ↪  Output, 0x02 - Alternate Function, 0x03 - Analog in/out)

    GPIOA->MODER   &= ~(0x00000003 << (2 * 2) | 0x00000003 << (3 * 2));    // This is needed for
    ↪  UART to work. It makes no sense.
    GPIOA->MODER   |=  (0x00000002 << (2 * 2) | 0x00000002 << (3 * 2));

    GPIOA->PUPDR   &= ~0x00000003 << (8 * 2);    // Clear push/pull register
    GPIOA->PUPDR   |=  0x00000000 << (8 * 2);    // Set push/pull register (0x00 - No pull, 0x01
    ↪  - Pull-up, 0x02 - Pull-down)

    GPIOB->ODR |=  (0x0001 << 6); // CS = 1

    // Configure SPI2
    SPI2->CR1 &= 0x3040; // Clear CR1 Register
    SPI2->CR1 |= 0x0000; // Configure direction (0x0000 - 2 Lines Full Duplex, 0x0400 - 2 Lines
    ↪  RX Only, 0x8000 - 1 Line RX, 0xC000 - 1 Line TX)
    SPI2->CR1 |= 0x0104; // Configure mode (0x0000 - Slave, 0x0104 - Master)
    SPI2->CR1 |= 0x0002; // Configure clock polarity (0x0000 - Low, 0x0002 - High)
    SPI2->CR1 |= 0x0001; // Configure clock phase (0x0000 - 1 Edge, 0x0001 - 2 Edge)
    SPI2->CR1 |= 0x0200; // Configure chip select (0x0000 - Hardware based, 0x0200 - Software
    ↪  based)
    SPI2->CR1 |= 0x0008; // Set Baud Rate Prescaler (0x0000 - 2, 0x0008 - 4, 0x0018 - 8, 0x0020 -
    ↪  16, 0x0028 - 32, 0x0028 - 64, 0x0030 - 128, 0x0038 - 128)
    SPI2->CR1 |= 0x0000; // Set Bit Order (0x0000 - MSB First, 0x0080 - LSB First)
    SPI2->CR2 &= ~0x0F00; // Clear CR2 Register
    SPI2->CR2 |= 0x0700; // Set Number of Bits (0x0300 - 4, 0x0400 - 5, 0x0500 - 6, ...);
    SPI2->I2SCFGR &= ~0x0800; // Disable I2S
    SPI2->CRCPR = 7; // Set CRC polynomial order
    SPI2->CR2 &= ~0x1000;
    SPI2->CR2 |= 0x1000; // Configure RXFIFO return at (0x0000 - Half-full (16 bits), 0x1000 -
    ↪  Quarter-full (8 bits))
    SPI2->CR1 |= 0x0040; // Enable SPI2

    lcd_reset();
}
```

## A.5 ansi.h

```
#ifndef _ANSI_H_
#define _ANSI_H_

void clrscr();

void clreol();

void gotoxy(int x, int y);

void underline(int enable);

void blink(int enable);

void inverse(int enable);

void hide_cursor();
void home_cursor();
int set_colors(char fg, char gb);
int set_color(char color);

#endif /* _ANSI_H_ */
```

## A.6 ansi.c

```
#include <stdio.h>

void clrscr() {
    printf("\x1B[2J\x1b[H");
}

void gotoxy(int x, int y) {
    printf("\x1B[%d;%df", y, x);
}

void home_cursor() {
  printf("\x1B[H");
}

void hide_cursor() {
  printf("\x1B[?25l");
}

int set_colors(char fg, char bg) {
  return printf("\x1B[%d;%dm", fg, bg);
}

int set_color(char color) {
  return printf("\x1B[%dm", color);
}
```

## A.7 graphics.h

```
#ifndef _GRAPHICS_H_
#define _GRAPHICS_H_

#define GRAPHICS_WIDTH 80
// Height must be an even number
#define GRAPHICS_HEIGHT 64
```

```c
typedef struct {
  char buffer[GRAPHICS_WIDTH][GRAPHICS_HEIGHT];
  // We use this color to clear the screen
  char background_color;
  int enabled;
} graphics_state_t;

typedef struct {
  char width;
  char height;
  char* data;
} sprite_t ;

graphics_state_t graphics_init();
void graphics_clear(graphics_state_t* ctx);
int graphics_show(graphics_state_t* ctx);

void graphics_draw_sprite(graphics_state_t* ctx, sprite_t sprite, int x, int y);

#endif // _GRAPHICS_H_
```

### A.8 graphics.c

```c
#include "30010_io.h"
#include "graphics.h"
#include "ansi.h"

#define GRAPHICS_TRANSPARENT 0

graphics_state_t graphics_init() {
  // Setup uart connections
  uart_init(2048000);
  // Set the background color to black so that we clear with black.
  set_colors(37, 40);
  clrscr();
  hide_cursor();

  graphics_state_t ctx = {
    .background_color = 90, // Grey
    .enabled = 1
  };
  graphics_clear(&ctx);
  return ctx;
}

// Sets every pixel to be the background color
void graphics_clear(graphics_state_t* graphics_state) {
  for (int x = 0; x < GRAPHICS_WIDTH; x++) {
    for (int y = 0; y < GRAPHICS_HEIGHT; y++) {
      graphics_state->buffer[x][y] = graphics_state->background_color;
    }
  }
}

// Turns the buffer into commands and sends it over USART
int graphics_show(graphics_state_t* ctx) {
  // Store how many bytes we send for debug reasons.
  int total_bytes = 3; // home_cursor is 3 bytes.
```

```
  // Set to some value that represents no color
  char previous_bg_color = 255;
  char previous_fg_color = 255;

  home_cursor();
  for (int y = 0; y < GRAPHICS_HEIGHT; y += 2) {
    for (int x = 0; x < GRAPHICS_WIDTH; x++) {
      char bg = ctx->buffer[x][y];
        char fg = ctx->buffer[x][y+1];
      if (bg == previous_bg_color && fg == previous_fg_color) {
        // Do nothing :)
      } else if (bg == previous_bg_color) {
        total_bytes += set_color(fg);
      } else if (fg == previous_fg_color) {
        total_bytes += set_color(bg + 10);
      } else {
        total_bytes += set_colors(fg, bg + 10);
      }
      uart_put_char('\xDC');
      total_bytes++;

      previous_bg_color = bg;
      previous_fg_color = fg;
    }
    uart_put_char('\r');
    uart_put_char('\n');
    total_bytes += 2;
  }

  return total_bytes;
}

int min(int a, int b) {
    return a<b ? a : b;
}

int max(int a, int b) {
    return a>b ? a : b;
}

// Draws the sprite within the area, but only the visible part of the sprite.
// If a sprite is on the boundary as such:
//
//    +---------------+
//    |               |
//  OO|XXXX           |
//   O|XXXXXX         |
//   O|XXXXXX         |
//  OO|XXXX           |
//    |               |
//    +---------------+
//
// We still draw the X part of the sprite.
void graphics_draw_sprite(graphics_state_t* ctx, sprite_t sprite, int x, int y) {
  int cutoff_dx = max(-x, 0);
  int cutoff_dy = max(-y, 0);

  int visible_width = min(sprite.width, GRAPHICS_WIDTH-x);
```

```
  int visible_height = min(sprite.height, GRAPHICS_HEIGHT-y);

  for (int dx = cutoff_dx; dx < visible_width; dx++) {
    for (int dy = cutoff_dy; dy < visible_height; dy++) {
      char byte = *(sprite.data + sprite.width * dy + dx);
      if (byte != GRAPHICS_TRANSPARENT) {
        ctx->buffer[x+dx][y+dy] = byte;
      }
    }
  }
}
```

### A.9 input.h

```
#ifndef _INPUT_H_
#define _INPUT_H_
#include <stdint.h>

#define KEY_H 9
#define KEY_SPACE 8
#define KEY_W 7
#define KEY_A 6
#define KEY_S 5
#define KEY_D 4
#define KEY_UP 3
#define KEY_LEFT 2
#define KEY_DOWN 1
#define KEY_RIGHT 0

 // This value contains whether each button we care about is pressed.
 // Each bit represents whether the key Shift, W, A, S, D, Up, Left, Down or Right is pressed.
typedef struct {
  uint16_t last_frame;
  uint16_t current_frame;
} input_state_t;

char is_down(input_state_t* state, char key);

char just_pressed(input_state_t* state, char key);

input_state_t input_init();
void input_update(input_state_t* state);

#endif
```

### A.10 input.c

```
#include "30010_io.h"
#include "input.h"
#include "gpio.h"

 // Checks if the specified key is down in the specified frame.
char _is_down(uint16_t pressed, char key) {
  return (pressed >> key) & 1;
}

char is_down(input_state_t* state, char key) {
  return _is_down(state->current_frame, key);
}
```

```
char just_pressed(input_state_t* state, char key) {
  return !_is_down(state->last_frame, key) && _is_down(state->current_frame, key);
}

input_state_t input_init() {
  init_joystick();

  input_state_t state = {0, 0};
  return state;
}

void input_update(input_state_t* state) {
  state->last_frame = state->current_frame;
  uint16_t current = 0;

  // Joystick
  current |= read_joystick() << 4;

  // Keyboard
  while (1) {
    uint8_t c = uart_get_char();
    switch (c) {
    case 'W':
    case 'w':
      current |= 1 << KEY_W;
      break;
    case 'A':
    case 'a':
      current |= 1 << KEY_A;
      break;
    case 'S':
    case 's':
      current |= 1 << KEY_S;
      break;
    case 'D':
    case 'd':
      current |= 1 << KEY_D;
      break;
    case ' ':
      current |= 1 << KEY_SPACE;
      break;
    case 'H':
    case 'h':
      current |= 1 << KEY_H;
      break;
    // Up =    "\x1b[A"
    // Down =  "\x1b[B"
    // Right = "\x1b[C"
    // Left =  "\x1b[D"
    case '\x1b':
      // The arrow keys are more complicated, so we have to parse more than
      // one byte. We assume the second byte is '['.
      uart_get_char();
      switch (uart_get_char()) {
      case 'A':
        current |= 1 << KEY_UP;
        break;
      case 'B':
```

```
        current |= 1 << KEY_DOWN;
        break;
      case 'C':
        current |= 1 << KEY_RIGHT;
        break;
      case 'D':
        current |= 1 << KEY_LEFT;
        break;
      default:
        //Ignore.
        break;
      }
      break;
    default:
      // Ignore.
      break;
    case 0:
      // No more characters
      state->current_frame = current;
      return;
    }
  }
}
```

## A.11 fixedpoint.h

```
#ifndef _FIXEDPOINT_H_
#define _FIXEDPOINT_H_
#include <stdint.h>

// 14 decimal bits, 17 whole part bits, and 1 sign bit.
#define FP_DECIMAL_BITS 14
#define FP_WHOLE_BITS 17
#define FP_TOTAL_BITS 32

// Get the whole or the fracitonal part
#define FP_WHOLE(n) ((int32_t) ((n) >> FP_DECIMAL_BITS))
#define FP_FRACTIONAL(n) ((n) & ((1 << FP_DECIMAL_BITS)-1))

// Create a new fixed point value from a whole number
#define FP_FROM_WHOLE(n) ((n) << FP_DECIMAL_BITS)

typedef int32_t fixedpoint_t;

typedef struct {
  fixedpoint_t x;
  fixedpoint_t y;
} vector_t;

// Axis-alligned rectabgle. Used for collision detection.
typedef struct {
  fixedpoint_t x;
  fixedpoint_t y;
  fixedpoint_t w;
  fixedpoint_t h;
} rectangle_t;

// Fixed-point
// fp stands for fixed-point
```

```c
fixedpoint_t fp_mul(fixedpoint_t a, fixedpoint_t b);
fixedpoint_t fp_div(fixedpoint_t a, fixedpoint_t b);
fixedpoint_t fp_sqrt(fixedpoint_t n);
int32_t fp_round(fixedpoint_t n);
fixedpoint_t fp_min(fixedpoint_t a, fixedpoint_t b);
fixedpoint_t fp_max(fixedpoint_t a, fixedpoint_t b);

void fp_print(fixedpoint_t n);

// Vectors
vector_t vector_from_whole(int16_t x, int16_t y);
vector_t vector_sub(vector_t* a, vector_t* b);
void clamp_vector(vector_t* vector, vector_t lower_bound, vector_t upper_bound);

fixedpoint_t vector_get_length(vector_t* vector);
void vector_set_length(vector_t* vector, fixedpoint_t length);

// Rectangles
rectangle_t rectangle_from_whole(int16_t x, int16_t y, int16_t w, int16_t h);
int rectangle_contains(rectangle_t rectangle, vector_t point);

#endif
```

### A.12 fixedpoint.c

```c
#include "fixedpoint.h"
#include <stdio.h>

// fp stands for fixed-point

// Multiplies two fixed point numbers with no loss of accuracy beyond necessary (I think)
fixedpoint_t fp_mul(fixedpoint_t a, fixedpoint_t b) {
  return (fixedpoint_t)((int64_t)a * (int64_t)b >> FP_DECIMAL_BITS);
}

fixedpoint_t fp_div(fixedpoint_t a, fixedpoint_t b) {
  return (fixedpoint_t)(((int64_t)a << FP_DECIMAL_BITS) / (int64_t)b);
}

fixedpoint_t fp_sqrt(fixedpoint_t n) {
  fixedpoint_t left = 0;
  fixedpoint_t right = n + 1;

  while (left != right - 1) {
    fixedpoint_t middle = (left + right) / 2;

    int64_t res = (int64_t)middle * (int64_t)middle >> FP_DECIMAL_BITS;
    if(res <= (int64_t)n) {
      left = middle;
    } else {
      right = middle;
    }
  }

  return left;
}

// Round to the neareast whole number, and return the value as an integer.
// Implements this algorithm: https://en.wikipedia.org/wiki/Rounding#Round_half_up
```

```
int32_t fp_round(fixedpoint_t n) {
  int32_t o = FP_WHOLE(n);
  if (FP_FRACTIONAL(n) >> (FP_DECIMAL_BITS - 1)) {
    o++;
  }

  return o;
}

fixedpoint_t fp_min(fixedpoint_t a, fixedpoint_t b) {
    return a<b ? a : b;
}

fixedpoint_t fp_max(fixedpoint_t a, fixedpoint_t b) {
    return a>b ? a : b;
}

// Debug print a fixed point value.
void fp_print(fixedpoint_t n) {
    if (n < 0) {
        putchar('-');
        // undo 2's compliment
        n = ~n + 1;
    }

    printf("%ld", FP_WHOLE(n));

    uint32_t number = FP_FRACTIONAL(n);
  if (number) {
    putchar('.');
  }

  while (number) {
    number *= 10;
    char digit = '0' + FP_WHOLE(number);
    putchar(digit);
    number = FP_FRACTIONAL(number);
  }
}

// Create a new vector from two whole numbers.
vector_t vector_from_whole(int16_t x, int16_t y) {
  vector_t vector = {
    FP_FROM_WHOLE(x),
    FP_FROM_WHOLE(y),
  };

  return vector;
}

vector_t vector_sub(vector_t* a, vector_t* b) {
  vector_t res;
  res.x = a->x - b->x;
  res.y = a->y - b->y;
  return res;
}

void clamp_vector(vector_t* vector, vector_t lower_bound, vector_t upper_bound) {
  fixedpoint_t x = vector->x;
```

```
  vector->x = fp_min(upper_bound.x, fp_max(lower_bound.x, x));

  fixedpoint_t y = vector->y;
  vector->y = fp_min(upper_bound.y, fp_max(lower_bound.y, y));
}

fixedpoint_t vector_get_length(vector_t* vector) {
  // sqrt(x**2 + y**2)
  return fp_sqrt(fp_mul(vector->x, vector->x) + fp_mul(vector->y, vector->y));
}

void vector_set_length(vector_t* vector, fixedpoint_t length) {
  fixedpoint_t current_length = vector_get_length(vector);
  vector->x = fp_mul(vector->x, fp_div(length, current_length));
  vector->y = fp_mul(vector->y, fp_div(length, current_length));
}

// Rectangles
rectangle_t rectangle_from_whole(int16_t x, int16_t y, int16_t w, int16_t h) {
  rectangle_t res = {
    .x = FP_FROM_WHOLE(x),
    .y = FP_FROM_WHOLE(y),
    .w = FP_FROM_WHOLE(w),
    .h = FP_FROM_WHOLE(h),
  };
  return res;
}

// Returns 1 if the rectangle does contain it, 0 otherwise.
int rectangle_contains(rectangle_t rectangle, vector_t point) {
  return point.x > rectangle.x
  && point.x < rectangle.x + rectangle.w
  && point.y > rectangle.y
  && point.y < rectangle.y + rectangle.h;
}
```

### A.13 random.h

```
// Shamelessly stolen (and modified) from: https://en.wikipedia.org/wiki/Xorshift
// It's Creative Commons! So it's legal. But I still say it because then it's not plagiat, I
↪  hope.

#ifndef _RANDOM_H_
#define _RANDOM_H_

#include <stdint.h>

typedef struct {
  uint64_t a;
} random_state_t;

uint64_t xorshift64(random_state_t *state);

// Everthing after this is my code.
random_state_t random_init();
uint64_t random_u64_up_to(random_state_t *state, uint64_t max);
int32_t random_i32_between(random_state_t *state, int32_t min, int32_t max);

#endif
```

## A.14 random.c

```
// Shamelessly stolen (and modified) from: https://en.wikipedia.org/wiki/Xorshift
// It's Creative Commons! So it's legal. But I still say it because then it's not plagiat, I
↪  hope.
// Xorshift is a simple algorithm, and this implementation will only repeat after 2^64-1 calls.
// For our pseudo-random purposes this is fine. However, we still need a seed. If we had a
↪  constant
// as a seed we would always see the same behavior, which would be pretty boring gameplay wise.
// (and completely defeat the purpose of a PRNG). Therefore we read the lower bits of the ADC
↪  upon
// initialization to create a random seed.

#include "random.h"
#include <stdint.h>
#include <stdio.h>
#include "stm32f30x.h"
#include "gpio.h"

uint64_t xorshift64(random_state_t *state) {
  uint64_t x = state->a;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  return state->a = x;
}

// Everything after this is my own code.
random_state_t random_init() {
    RCC->AHBENR |= RCC_AHBPeriph_GPIOA; // Enable clock for GPIO Port A

    set_input(GPIOA, 0);
    set_input(GPIOA, 1);

  RCC->CFGR2 &= ~RCC_CFGR2_ADCPRE12; // Clear ADC12 prescaler bits
  RCC->CFGR2 |= RCC_CFGR2_ADCPRE12_DIV6; // Set ADC12 prescaler to 6
  RCC->AHBENR |= RCC_AHBPeriph_ADC12; // Enable clock for ADC12

  ADC1->CR = 0x00000000; // Clear CR register
  ADC1->CFGR &= 0xFDFFC007; // Clear ADC1 config register
  ADC1->SQR1 &= ~ADC_SQR1_L; // Clear regular sequence register 1

  ADC1->CR |= 0x10000000; // Enable internal ADC voltage regulator
  for (int i = 0 ; i < 1000 ; i++) {} // Wait for about 16 microseconds

  ADC1->CR |= 0x80000000; // Start ADC1 calibration
  while (!(ADC1->CR & 0x80000000)); // Wait for calibration to finish
  for (int i = 0 ; i < 100 ; i++) {} // Wait for a little while

  ADC1->CR |= 0x00000001; // Enable ADC1 (0x01 - Enable, 0x02 - Disable)
  while (!(ADC1->ISR & 0x00000001)); // Wait until ready

  ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_1Cycles5);

  // We take 1 bit per measurement and construct a random 64 bit value.
  uint64_t seed = 0;
  for (int i = 0; i < 64; i++) {
    ADC_StartConversion(ADC1);
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0); // Wait for ADC read
```

```
    uint16_t analog_value = ADC_GetConversionValue(ADC1); // Read the ADC value

    // Move everything left
    seed <<= 1;
    // Set least significant bit of seed to least significant bit of the
    // analog value
    seed |= analog_value & 1;
  }

  random_state_t state = {
    .a = seed,
  };

  return state;
}

// Generates a random number from 0 to the bound excluding the bound.
uint64_t random_u64_up_to(random_state_t *state, uint64_t bound) {
  // In order to assure randomness, we sometimes have to throw numbers out.
  // For example, if we had a PRNG give us a number between 0 and 9 (inlcuding both),
  // and we did n % 3 to limit the number to 0, 1 and 2 we'd have a higher chance of
  // geting 0 than the rest.
  //
  // (0, 3, 6, 9) => 0, (1, 4, 7) => 1, (2, 5, 8) => 2
  //
  // If we imagine the random numbers we can get as a string that the PRNG selects from
  // we simply cut off the last period of our numbers.
  // 0 1 2 0 1 2 0 1 2|0
  //                  ^ Everything after | is discarded. If our PRNG selects it, we simply
  ↪  reroll.

  // What we're doing is 2**64 - (2**64 % max) but we can't use that number because it's too big.
  // uint64_t extended_max = 0 - (UINT64_MAX % max + 1) % max);
  // Apparently underflow is undefined behavior in C, so I'll give up on this for now.

  // while (1) {
  //   uint64_t n = xorshift64(state);
  //   if (n < extended_max) {
  //     return n % max;

  //   }
  //   // Try with new number
  // }
  return xorshift64(state) % bound;
}

// min has to be smaller than max.
int32_t random_i32_between(random_state_t *state, int32_t min, int32_t max) {
  return min + (int32_t) random_u64_up_to(state, max - min);
}
```

## A.15 game_state.h

```
#ifndef _GAME_STATE_H_
#define _GAME_STATE_H_

typedef enum {
  PLAYING,
  DEATH_SCREEN,
```

```
  HELP_SCREEN
} game_state_t;

#endif
```

### A.16 main.c

```
// HAL
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"     // Input/output library for this course
#include "gpio.h"

// API
#include "ansi.h"
#include "graphics.h"
#include "input.h"
#include "fixedpoint.h"
#include "random.h"

// AP
#include "game_state.h"
#include "player.h"
#include "enemy.h"
#include "projectiles.h"
#include "help_screen.h"
#include "death_screen.h"

// We pick to run our game at 30 Hz, which means each frame is 33.33 ms
#define FRAME_DURATION 33
// Run using debug features
// #define DEBUG_GAME_INFO

volatile int32_t milliseconds = 0;

// Timer 2's interupt
void TIM2_IRQHandler() {
  milliseconds++;

  TIM2->SR &= ~0x0001;
}

void timer_init() {
  // Set up clock with 100Hz frequency
  RCC->APB1ENR |= RCC_APB1Periph_TIM2; // Enable clock line to timer 2
  // No remapping, No clock division, Not buffered, Edge-aligned, Up-counting mode,
  // One-pulse mode disabled, Any update request source, update events disabled
  TIM2->CR1 = 0x0000;
  TIM2->ARR = 64000 - 1;
  TIM2->PSC = 0;
  // Enable interrupt
  TIM2->DIER |= 0x0001;
  NVIC_SetPriority(TIM2_IRQn, 2);
  NVIC_EnableIRQ(TIM2_IRQn);

  // Enable timer
  TIM2->CR1 = 0x0001;
}

void print_binary(uint16_t value) {
```

```
  for (int i = 16; i > 0; i--) {
    if ((value >> i) & 1) {
      uart_put_char('1');
    } else {
      uart_put_char('0');
    }
  }
}

int main(void) {
  game_state_t gamestate = PLAYING;
  game_state_t previous_gamestate = PLAYING;

  timer_init();
  random_state_t random_state = random_init();
  graphics_state_t graphics_state = graphics_init();
  input_state_t input_state = input_init();

  player_state_t player_state = player_init();
  enemy_state_t enemy_state = enemy_init();
  projectiles_state_t projectiles_state = projectiles_init();

  death_screen_state_t deathscreen_state;

  // The current frame we're on
  int frame = 0;
  while(1) {
    // Get input
    input_update(&input_state);

    switch (gamestate) {
    case PLAYING:
      // Update world
      player_update(
        &player_state,
        &input_state,
        &projectiles_state,
        &gamestate
      );
      enemy_update(&enemy_state, &projectiles_state, &player_state, &random_state);
      projectiles_update(&projectiles_state, &player_state, &enemy_state);
      // So that we can enter the help screen.
      help_screen_update(&input_state, &gamestate);

      // Render world (into buffer)
      enemy_draw(&enemy_state, &graphics_state);
      player_draw(&player_state, &graphics_state);
      projectiles_draw(&projectiles_state, &graphics_state);
      break;
    case DEATH_SCREEN:
      death_screen_update(&deathscreen_state, &random_state);
      death_screen_draw(&deathscreen_state, &graphics_state);
      break;
    case HELP_SCREEN:
      help_screen_update(&input_state, &gamestate);
      help_screen_draw(&graphics_state);
      break;
    }
```

```
    int bytes_sent = 0;
    if (graphics_state.enabled) {
      // Send rendered world over USART
      bytes_sent = graphics_show(&graphics_state);
    }

    // Clean (get ready for next frame)
    frame++;
    graphics_clear(&graphics_state);

    // Check if we've switch gamestate
    if (gamestate != previous_gamestate) {

      // Call exit functions
      switch (previous_gamestate) {
      case HELP_SCREEN:
        // Re-enable the normal graphics system.
        graphics_state.enabled = 1;
        break;
      default:
        break;
      }

      // Call enter functions
      switch (gamestate) {
      case DEATH_SCREEN:
        deathscreen_state = death_screen_enter(&graphics_state, &random_state);
        break;
      case HELP_SCREEN:
        // Disable the normal graphics system.
        graphics_state.enabled = 0;
        break;
      default:
        break;
      }

      previous_gamestate = gamestate;
    }

#ifdef DEBUG_GAME_INFO
    // Debug info
    set_colors(32, 40);
    printf("   Frame: %d\n", frame-1);
    printf("   ms left: %ld \n", (frame)*FRAME_DURATION - milliseconds);
    uart_put_string("This frame: ");
    print_binary(input_state.current_frame);
    uart_put_string("\n\rLast frame: ");
    print_binary(input_state.last_frame);
    uart_put_string("\n\r");
    printf("Bytes sent: %d ", bytes_sent);
#endif

    // wait until next frame
    while (milliseconds < frame * FRAME_DURATION) {}
  }
}
```

## A.17 player.h

```
#ifndef _PLAYER_H_
#define _PLAYER_H_
#include "game_state.h"
#include "graphics.h"
#include "input.h"
#include "fixedpoint.h"

typedef struct {
  sprite_t sprite;
  vector_t position;
  vector_t velocity;
  // If it's 0, we're alive
  char dead;
  // A number representing how much health we have
  int health;
} player_state_t;


// I have to move some of the includes down here, because there's a circular
// dependency between the types for the methods. I really feel like C compilers
// shouldn't be having this problem with modern compilers, but there's probably
// some arcane reason they can't modify the header parser to lazily parse type
// names. My solution to this problem is inspired by:
// https://stackoverflow.com/questions/46150724/circular-dependency-between-c-header-files
#include "projectiles.h"

player_state_t player_init();

void player_update(
  player_state_t* player_state,
  input_state_t* input_state,
  projectiles_state_t* projectiles_state,
  game_state_t* gamestate
);

void player_draw(
  player_state_t* player_state,
  graphics_state_t* graphics_state
);

#endif
```

## A.18 player.c

```
#include "player.h"

#define PLAYER_WIDTH 5
#define PLAYER_HEIGHT 4

#define PLAYER_SPEED 40
#define PLAYER_FRICTION_INV 10

static const char data[] = {
  36, 96, 96,  0,  0,
   0, 36, 36, 36, 36,
   0, 36, 36, 36, 36,
  36, 36, 36,  0,  0
};
```

```
static const sprite_t player_sprite = {
  .width = 5,
  .height = 4,
  .data = (char*) &data,
};


player_state_t player_init() {
  player_state_t player_state = {
    .sprite = player_sprite,
    .position = vector_from_whole(5, 32),
    .velocity = vector_from_whole(0, 0),
    .dead = 0,
    .health = 5
  };
  return player_state;
}

void player_update(
  player_state_t* player_state,
  input_state_t* input_state,
  projectiles_state_t* projectiles_state,
  game_state_t* gamestate
) {
  vector_t acceleration = vector_from_whole(0, 0);

  if (is_down(input_state, KEY_W)) {
    acceleration.y -= FP_FROM_WHOLE(PLAYER_SPEED);
  }
  if (is_down(input_state, KEY_A)) {
    acceleration.x -= FP_FROM_WHOLE(PLAYER_SPEED);
  }
  if (is_down(input_state, KEY_S)) {
    acceleration.y += FP_FROM_WHOLE(PLAYER_SPEED);
  }
  if (is_down(input_state, KEY_D)) {
    acceleration.x += FP_FROM_WHOLE(PLAYER_SPEED);
  }

  // Apply friction
  fixedpoint_t length = vector_get_length(&player_state->velocity);
  acceleration.x -= fp_mul(player_state->velocity.x, length) / PLAYER_FRICTION_INV;
  acceleration.y -= fp_mul(player_state->velocity.y, length) / PLAYER_FRICTION_INV;


  // Apply acceleration
  player_state->velocity.x += acceleration.x / 30;
  player_state->velocity.y += acceleration.y / 30;

  // Update position
  player_state->position.x += player_state->velocity.x / 30;
  player_state->position.y += player_state->velocity.y / 30;

  // Limit position
  clamp_vector(
    &player_state->position,
    vector_from_whole(0, 0),
    vector_from_whole(GRAPHICS_WIDTH-PLAYER_WIDTH, GRAPHICS_HEIGHT-PLAYER_HEIGHT)
```

```
  );


  // Shooting
  if (just_pressed(input_state, KEY_SPACE)) {
    projectile_t projectile = {
      .position = player_state->position,
      .velocity = vector_from_whole(30, 0),
      .color = 36,
    };
    // Offset the projectile
    projectile.position.x += FP_FROM_WHOLE(PLAYER_WIDTH) / 2;
    projectile.position.y += FP_FROM_WHOLE(PLAYER_HEIGHT) / 2;
    projectiles_add(projectiles_state, projectile);
  }

  // Health
  if (player_state->health <= 0) {
    // We're dead
    player_state->dead = 1;
    *gamestate = DEATH_SCREEN;
  }
}

void player_draw(
  player_state_t* player_state,
  graphics_state_t* graphics_state
) {
  if (!player_state->dead) {
    int x = fp_round(player_state->position.x);
    int y = fp_round(player_state->position.y);
    graphics_draw_sprite(
      graphics_state,
      player_state->sprite,
      x,
      y
    );
  }
}
```

### A.19 enemy.h

```
#ifndef _ENEMY_H_
#define _ENEMY_H_
#include "graphics.h"
#include "input.h"
#include "fixedpoint.h"
#include "random.h"

#define MAX_ENEMIES 5

typedef enum {
  ACTION_APPROACHING,
  ACTION_FLEEING,
  ACTION_SHOOTING,
  ACTION_IDLE
} action_t ;

typedef struct {
```

```
  sprite_t sprite;
  vector_t position;
  vector_t velocity;
  int health;

  // AI State Machine
  action_t action;
  int time_until_next_action;
} enemy_t ;


typedef struct {
  int count;
  enemy_t enemies[MAX_ENEMIES];
} enemy_state_t;


// I have to move some of the includes down here, because there's a circular
// dependency between the types for the methods. I really feel like C compilers
// shouldn't be having this problem with modern compilers, but there's probably
// some arcane reason they can't modify the header parser to lazily parse type
// names. My solution to this problem is inspired by:
// https://stackoverflow.com/questions/46150724/circular-dependency-between-c-header-files
#include "player.h"

enemy_state_t enemy_init();

void enemy_add(enemy_state_t* enemy_state, enemy_t enemy);
void enemy_remove(enemy_state_t* enemy_state, int index);

void enemy_handle_damage(enemy_state_t* enemy_state, int index, int damage);

void enemy_update(
  enemy_state_t* enemy_state,
  projectiles_state_t* projectile_state,
  player_state_t* player_state,
  random_state_t* random_state
);

void enemy_draw(
  enemy_state_t* enemy_state,
  graphics_state_t* graphics_state
);

#endif
```

### A.20 enemy.c

```
#include "enemy.h"
#include "graphics.h"
#include "input.h"
#include "fixedpoint.h"
#include "random.h"
#include "player.h"

static char data[] = {
   0,  0,  0, 95, 95,  0,
   0, 95, 95, 35, 35, 35,
  95, 35, 35, 35, 95, 95,
```

```
   0, 35, 35, 35, 35, 35,
   0,  0,  0, 35, 35,  0,
};

#define ENEMY_WIDTH 6
#define ENEMY_HEIGHT 5

#define ENEMY_SPEED FP_FROM_WHOLE(15)
#define ENEMY_FRICTION_INV 50

#define BULLET_SPEED FP_FROM_WHOLE(25)

// If enemies go outside this area they vanish
const rectangle_t ENEMY_AREA = {
  FP_FROM_WHOLE(-10),
  FP_FROM_WHOLE(-10),
  FP_FROM_WHOLE(GRAPHICS_WIDTH+10),
  FP_FROM_WHOLE(GRAPHICS_HEIGHT+10)
};

static sprite_t enemy_sprite = {
  .width = 6,
  .height = 5,
  .data = (char*) &data,
};

enemy_state_t enemy_init() {
  enemy_state_t enemy_state = {
    .count = 0,
    .enemies = {}
  };

  return enemy_state;
}

void enemy_add(
  enemy_state_t* enemy_state,
  enemy_t enemy
) {
  if (enemy_state->count >= MAX_ENEMIES) {
    return;
  }

  enemy_state->enemies[enemy_state->count] = enemy;
  enemy_state->count++;
}

void enemy_remove(enemy_state_t* enemy_state, int index) {
  // Move enemies i + 1 to enemy_state->enemy back by 1
  for (int i = index; i < enemy_state->count - 1; i++) {
    enemy_state->enemies[i] = enemy_state->enemies[i+1];
  }
  enemy_state->count--;
}

void enemy_handle_damage(enemy_state_t* enemy_state, int index, int damage) {
  enemy_t* enemy = &enemy_state->enemies[index];
  enemy->health -= damage;
  if (enemy->health <= 0) {
```

```
    // It's dead
    enemy_remove(enemy_state, index);
  } else if (enemy->health == 1) {
    // If we're on our last sliver of health, run away
    enemy->action = ACTION_FLEEING;
    enemy->time_until_next_action = 2*30; // 2 secs
  }

  // We could show the damage taken by flashing here or something.
}

void enemy_update(
  enemy_state_t* enemy_state,
  projectiles_state_t* projectile_state,
  player_state_t* player_state,
  random_state_t* random_state
) {
  if (enemy_state->count < MAX_ENEMIES && random_u64_up_to(random_state, 100) == 0) {
    enemy_t new_enemy = {
      .sprite = enemy_sprite,
      .velocity = vector_from_whole(0, 0),
      .health = 3,
      .time_until_next_action = random_i32_between(random_state, 30, 90),
      .action = ACTION_APPROACHING
    };

    new_enemy.position.x = FP_FROM_WHOLE(GRAPHICS_WIDTH);
    new_enemy.position.y = random_i32_between(random_state, 0, FP_FROM_WHOLE(GRAPHICS_HEIGHT));
    enemy_add(enemy_state, new_enemy);
  }

  // Update each enemy
  for (int i = enemy_state->count - 1; i >= 0; i--) {
    enemy_t* enemy = &enemy_state->enemies[i];

    if (enemy->time_until_next_action <= 0) {
      switch (enemy->action) {
      case ACTION_APPROACHING:
        // We always shoot after approaching
        enemy->action = ACTION_SHOOTING;
        enemy->time_until_next_action = random_i32_between(random_state, 15*2, 15*5);
        break;
      case ACTION_SHOOTING:
        // After shooting we give the player a break by either moving or drifting.
        enemy->action = random_u64_up_to(random_state, 1) ? ACTION_IDLE : ACTION_APPROACHING;
        enemy->time_until_next_action = random_i32_between(random_state, 30, 3*30);
        break;
      case ACTION_IDLE:
        // After giving the player a break we either shoot now, or shoot after approaching
        enemy->action = random_u64_up_to(random_state, 1) ? ACTION_SHOOTING : ACTION_APPROACHING;
        if (enemy->action == ACTION_SHOOTING) {
          enemy->time_until_next_action = random_i32_between(random_state, 15*2, 15*5);
        } else {
          enemy->time_until_next_action = random_i32_between(random_state, 30, 3*30);
        }
        break;
      case ACTION_FLEEING:
        // Roll to be brave!
        // One in six chance
```

```
    enemy->action = random_u64_up_to(random_state, 6) ? ACTION_FLEEING : ACTION_APPROACHING;
    enemy->time_until_next_action = 60;
    break;
  }
}

vector_t acceleration = vector_from_whole(0, 0);

vector_t direction;

switch (enemy->action) {
case ACTION_APPROACHING:
  direction = vector_sub(&player_state->position, &enemy->position);
  vector_set_length(&direction, ENEMY_SPEED);
  acceleration.x += direction.x;
  acceleration.y += direction.y;
  break;
case ACTION_SHOOTING:
  // Only shoot 2 times per second
  if (enemy->time_until_next_action % 15 == 0) {
    direction = vector_sub(&player_state->position, &enemy->position);
    vector_set_length(&direction, BULLET_SPEED);

    projectile_t projectile = {
      .position = enemy->position,
      .velocity = direction,
      .color = 95
    };
    // Offset the position
    projectile.position.x += FP_FROM_WHOLE(ENEMY_WIDTH) / 2;
    projectile.position.y += FP_FROM_WHOLE(ENEMY_HEIGHT) / 2;

    projectiles_add(projectile_state, projectile);
    acceleration.x -= enemy->velocity.x;
    acceleration.y -= enemy->velocity.y;
  }
  break;
case ACTION_IDLE:
  break;
case ACTION_FLEEING:
  direction = vector_sub(&enemy->position, &player_state->position);
  vector_set_length(&direction, ENEMY_SPEED);
  acceleration.x += direction.x;
  acceleration.y += direction.y;
  break;
}

// Friction
fixedpoint_t length = vector_get_length(&enemy->velocity);
acceleration.x -= fp_mul(enemy->velocity.x, length) / ENEMY_FRICTION_INV;
acceleration.y -= fp_mul(enemy->velocity.y, length) / ENEMY_FRICTION_INV;

// Apply acceleration
enemy->velocity.x += acceleration.x / 30;
enemy->velocity.y += acceleration.y / 30;

// Update position
enemy->position.x += enemy->velocity.x / 30;
enemy->position.y += enemy->velocity.y / 30;
```

```
    // If enemies are too far away we remove them
    if (!rectangle_contains(ENEMY_AREA, enemy->position)) {
      enemy_remove(enemy_state, i);
    }

    enemy->time_until_next_action -= 1;
  }
}

void enemy_draw(
  enemy_state_t* enemy_state,
  graphics_state_t* graphics_state
) {
  for (int i = 0; i < enemy_state->count; i++) {
    enemy_t* enemy = &enemy_state->enemies[i];

    int x = fp_round(enemy->position.x);
    int y = fp_round(enemy->position.y);
    graphics_draw_sprite(
      graphics_state,
      enemy->sprite,
      x,
      y
    );
  }
}
```

### A.21 projectiles.h

```
#ifndef _PROJECTILES_H_
#define _PROJECTILES_H_
#include "fixedpoint.h"
#include "graphics.h"

#define MAX_PROJECTILES 50

typedef struct {
  vector_t position;
  vector_t velocity;
  uint8_t color;
} projectile_t;

typedef struct {
  projectile_t projectiles[MAX_PROJECTILES];
  uint8_t count;
} projectiles_state_t;

// I have to move some of the includes down here, because there's a circular
// dependency between the types for the methods. I really feel like C compilers
// shouldn't be having this problem with modern compilers, but there's probably
// some arcane reason they can't modify the header parser to lazily parse type
// names. My solution to this problem is inspired by:
// https://stackoverflow.com/questions/46150724/circular-dependency-between-c-header-files
#include "enemy.h"
#include "player.h"

projectiles_state_t projectiles_init();
```

```
void projectiles_add(
  projectiles_state_t* projectiles_state,
  projectile_t projectile
);

void projectiles_remove(projectiles_state_t* projectiles_state, int index);

void projectiles_update(
  projectiles_state_t* projectiles_state,
  player_state_t* player_state,
  enemy_state_t* enemy_state
);

void projectiles_draw(
  projectiles_state_t* projectiles_state,
  graphics_state_t* graphics_state
);

#endif
```

## A.22 projectiles.c

```
#include "projectiles.h"

const rectangle_t SCREEN = {
  FP_FROM_WHOLE(0),
  FP_FROM_WHOLE(0),
  FP_FROM_WHOLE(GRAPHICS_WIDTH),
  FP_FROM_WHOLE(GRAPHICS_HEIGHT)
};

projectiles_state_t projectiles_init() {
  projectiles_state_t projectiles_state = {
    .count = 0,
    .projectiles = {}
  };

  return projectiles_state;
}

void projectiles_add(
  projectiles_state_t* projectiles_state,
  projectile_t projectile
) {
  if (projectiles_state->count >= MAX_PROJECTILES) {
    return;
  }

  projectiles_state->projectiles[projectiles_state->count] = projectile;
  projectiles_state->count++;
}

void projectiles_remove(projectiles_state_t* projectiles_state, int index) {
  // Move prjectile i + 1 to projectiles_state->projectile_count back
  for (int i = index; i < projectiles_state->count - 1; i++) {
    projectiles_state->projectiles[i] = projectiles_state->projectiles[i+1];
  }
  projectiles_state->count--;
}
```

```
void projectiles_update(
  projectiles_state_t* projectiles_state,
  player_state_t* player_state,
  enemy_state_t* enemy_state
) {
  // Loop backwards so we can remove bullets easily
  for (int i = projectiles_state->count - 1; i >= 0; i--) {
    projectile_t* projectile = &projectiles_state->projectiles[i];
    // Update position
    projectile->position.x += projectile->velocity.x / 30;
    projectile->position.y += projectile->velocity.y / 30;

    // Check if we're colling with any enemy
    for (int j = 0; j < enemy_state->count; j++) {
      enemy_t* enemy = &enemy_state->enemies[j];
      rectangle_t collider_box = {
        .x = enemy->position.x,
        .y = enemy->position.y,
        .w = FP_FROM_WHOLE(enemy->sprite.width),
        .h = FP_FROM_WHOLE(enemy->sprite.height)
      };

      if (projectile->color == 36 && rectangle_contains(collider_box, projectile->position)) {
        // Hit! Apply damage and delete projectile
        enemy_handle_damage(enemy_state, j, 1);

        projectiles_remove(projectiles_state, i);
        goto outer_continue;
      }
    }
    // Check if we're colliding with the player
    rectangle_t collider_box = {
      .x = player_state->position.x,
      .y = player_state->position.y,
      .w = FP_FROM_WHOLE(player_state->sprite.width),
      .h = FP_FROM_WHOLE(player_state->sprite.height)
    };

    if (projectile->color == 95 && rectangle_contains(collider_box, projectile->position)) {
      // Hit! Apply damage and delete projectile
      player_state->health--; // 1 damage.

      projectiles_remove(projectiles_state, i);
      goto outer_continue;
    }

    // Check if out of bounds
    if (!rectangle_contains(SCREEN, projectile->position)) {
      projectiles_remove(projectiles_state, i);
    }

    outer_continue:;
  }
}

void projectiles_draw(
  projectiles_state_t* projectiles_state,
  graphics_state_t* graphics_state
```

```
) {
  for (int i = 0; i < projectiles_state->count; i++) {
    projectile_t* projectile = &projectiles_state->projectiles[i];
    int32_t x = FP_WHOLE(projectile->position.x);
    int32_t y = FP_WHOLE(projectile->position.y);
    graphics_state->buffer[x][y] = projectile->color;
  }
}
```

### A.23 death_screen.h

```
#ifndef _DEATH_SCREEN_H_
#define _DEAHT_SCREEN_H_
#include "graphics.h"
#include "random.h"

typedef struct {
  int time_until_on;
  int time_until_blink;
  int blink_count;
  sprite_t sprite;
  int x;
  int y;
} letter_t ;

#define LETTER_COUNT 8

typedef struct {
  letter_t letters[LETTER_COUNT];
} death_screen_state_t ;

static const char G_DATA[] = {
   0, 97, 97, 97, 97,
  97, 97,  0,  0,  0,
  97, 97,  0, 97, 97,
  97, 97,  0,  0, 97,
   0, 97, 97, 97,  0
};

static const sprite_t G_sprite = {
  .data = (char*) &G_DATA,
  .width = 5,
  .height = 5
};

static const char A_DATA[] = {
   0, 97, 97, 97,  0,
  97, 97,  0, 97, 97,
  97, 97,  0, 97, 97,
  97, 97, 97, 97, 97,
  97, 97,  0, 97, 97
};

static const sprite_t A_sprite = {
  .data = (char*) &A_DATA,
  .width = 5,
  .height = 5
};
```

```
static const char M_DATA[] = {
  97, 97,  0,  0,  0,  0, 97, 97,
  97, 97, 97,  0,  0, 97, 97, 97,
  97, 97, 97, 97, 97, 97, 97, 97,
  97, 97,  0, 97, 97,  0, 97, 97,
  97, 97,  0, 97, 97,  0, 97, 97
};

static const sprite_t M_sprite = {
  .data = (char*) &M_DATA,
  .width = 8,
  .height = 5
};

static const char E_DATA[] = {
  97, 97, 97, 97, 97,
  97, 97,  0,  0,  0,
  97, 97, 97, 97,  0,
  97, 97,  0,  0,  0,
  97, 97, 97, 97, 97
};

static const sprite_t E_sprite = {
  .data = (char*) &E_DATA,
  .width = 5,
  .height = 5
};

static const char O_DATA[] = {
   0, 97, 97, 97,  0,
  97, 97,  0, 97, 97,
  97, 97,  0, 97, 97,
  97, 97,  0, 97, 97,
   0, 97, 97, 97,  0
};

static const sprite_t O_sprite = {
  .data = (char*) &O_DATA,
  .width = 5,
  .height = 5
};

static const char V_DATA[] = {
  97, 97,  0, 97, 97,
  97, 97,  0, 97, 97,
  97, 97,  0, 97, 97,
   0, 97, 97, 97,  0,
   0, 97, 97, 97,  0
};

static const sprite_t V_sprite = {
  .data = (char*) &V_DATA,
  .width = 5,
  .height = 5
};

static const char R_DATA[] = {
  97, 97, 97, 97,  0,
  97, 97,  0, 97, 97,
```

```
  97, 97, 97, 97, 97,
  97, 97,  0, 97,  0,
  97, 97,  0, 97, 97
};

static const sprite_t R_sprite = {
  .data = (char*) &R_DATA,
  .width = 5,
  .height = 5
};

death_screen_state_t death_screen_enter(
  graphics_state_t* graphics_state,
  random_state_t* random_state
);

void death_screen_update(
  death_screen_state_t* deathscreen_state,
  random_state_t* random_state
);

void death_screen_draw(
  death_screen_state_t* deathscreen_state,
  graphics_state_t* graphics_state
);

#endif
```

## A.24 death_screen.c

```
#include "death_screen.h"

death_screen_state_t death_screen_enter(
  graphics_state_t* graphics_state,
  random_state_t* random_state
) {
  graphics_state->background_color = 30; // Black
  graphics_clear(graphics_state);

  death_screen_state_t deathscreen_state = {
    .letters = {
      {
        .sprite = G_sprite,
        .x = 27,
        .y = 26
      },
      {
        .sprite = A_sprite,
        .x = 33,
        .y = 26
      },
      {
        .sprite = M_sprite,
        .x = 39,
        .y = 26
      },
      {
        .sprite = E_sprite,
        .x = 48,
```

```
        .y = 26
      },
      {
        .sprite = O_sprite,
        .x = 29,
        .y = 32
      },
      {
        .sprite = V_sprite,
        .x = 35,
        .y = 32
      },
      {
        .sprite = E_sprite,
        .x = 41,
        .y = 32
      },
      {
        .sprite = R_sprite,
        .x = 47,
        .y = 32
      }
    }
  };

  for (int i = 0; i < LETTER_COUNT; i++) {
    letter_t* letter = &deathscreen_state.letters[i];
    // Half of the letters will turn on and stay on, and the other half
    // will blink on.
    if (random_i32_between(random_state, 0, 1)) {
      letter->time_until_on = random_i32_between(random_state, 40, 80);
      letter->time_until_blink = random_i32_between(random_state, 120, 1000);
    } else {
      letter->time_until_on = random_i32_between(random_state, 40, 80);
      letter->time_until_blink = letter->time_until_on;
    }
    letter->blink_count = random_i32_between(random_state, 1, 4);
  }

  return deathscreen_state;
}

void death_screen_update(
  death_screen_state_t* deathscreen_state,
  random_state_t* random_state
) {
  for (int i = 0; i < LETTER_COUNT; i++) {
    letter_t* letter = &deathscreen_state->letters[i];

    if (letter->time_until_on > 0) {
      letter->time_until_on--;
    }
    // If we're done blinking
    if (letter->time_until_blink + letter->blink_count * 4 < 0) {
      letter->time_until_blink = random_i32_between(random_state, 300, 1000);
      letter->blink_count = random_i32_between(random_state, 1, 4);
    }

    letter->time_until_blink--;
```

```
  }
}

int letter_is_on(letter_t* letter) {
  if (letter->time_until_on > 0) {
    // Letter is off
    return 0;
  }
  if (letter->time_until_blink > 0) {
    // Letter is on, and we're not blinking
    return 1;
  }
  // Every second frame we're gonna blink
  return (letter->blink_count * 4 + letter->time_until_blink) / 2 % 2;
}

void death_screen_draw(
  death_screen_state_t* deathscreen_state,
  graphics_state_t* graphics_state
) {
  // Draw the letters
  for (int i = 0; i < LETTER_COUNT; i++) {
    letter_t* letter = &deathscreen_state->letters[i];
    if (letter_is_on(letter)) {
      graphics_draw_sprite(
        graphics_state,
        letter->sprite,
        letter->x,
        letter->y
      );
    }
  }
}
```

## A.25 help_screen.h

```
#ifndef _HELP_SCREEN_H_
#define _HELP_SCREEN_H_

#include "game_state.h"
#include "graphics.h"
#include "input.h"

void help_screen_update(input_state_t* input_state, game_state_t* gamestate);
void help_screen_draw(graphics_state_t* graphics_state);

#endif
```

## A.26 help_screen.c

```
#include "help_screen.h"
#include <stdio.h>
#include "ansi.h"

void help_screen_update(input_state_t* input_state, game_state_t* gamestate) {
  if (just_pressed(input_state, KEY_H)) {
    switch (*gamestate) {
    case PLAYING:
      *gamestate = HELP_SCREEN;
```

```
      break;
    case HELP_SCREEN:
      *gamestate = PLAYING;
      break;
    default:
      break;
    }
  }
}

void help_screen_draw(graphics_state_t* graphics_state) {
  set_colors(97, 40);
  gotoxy(23, 11);
  printf("Use the WASD keys or the joystick ");
  gotoxy(23, 12);
  printf("to move your spaceship.           ");
  gotoxy(23, 13);
  printf("Use the SPACE bar or press the    ");
  gotoxy(23, 14);
  printf("joystick to fire your weapon.     ");
  gotoxy(23, 15);
  printf("Enemy ships require multiple hits ");
  gotoxy(23, 16);
  printf("to go down, and so do you.        ");
  gotoxy(23, 17);
  printf("Try to avoid as much damage as you");
  gotoxy(23, 18);
  printf("can, as repairs are slow in space.");
  gotoxy(23, 19);
  printf("                                  ");
  gotoxy(23, 20);
  printf("                       Good luck.");
  gotoxy(0, 33);
}

// Use the WASD keys or the joystick
// to move your spaceship.
// Use the SPACE bar or press the
// joystick to fire your weapon.
// Enemy ships require multiple hits
// to go down, and so do you.
// Try to avoid as much damage as you
// can, as repairs are slow in space.
//
//                        Good luck.
```

# B Journal

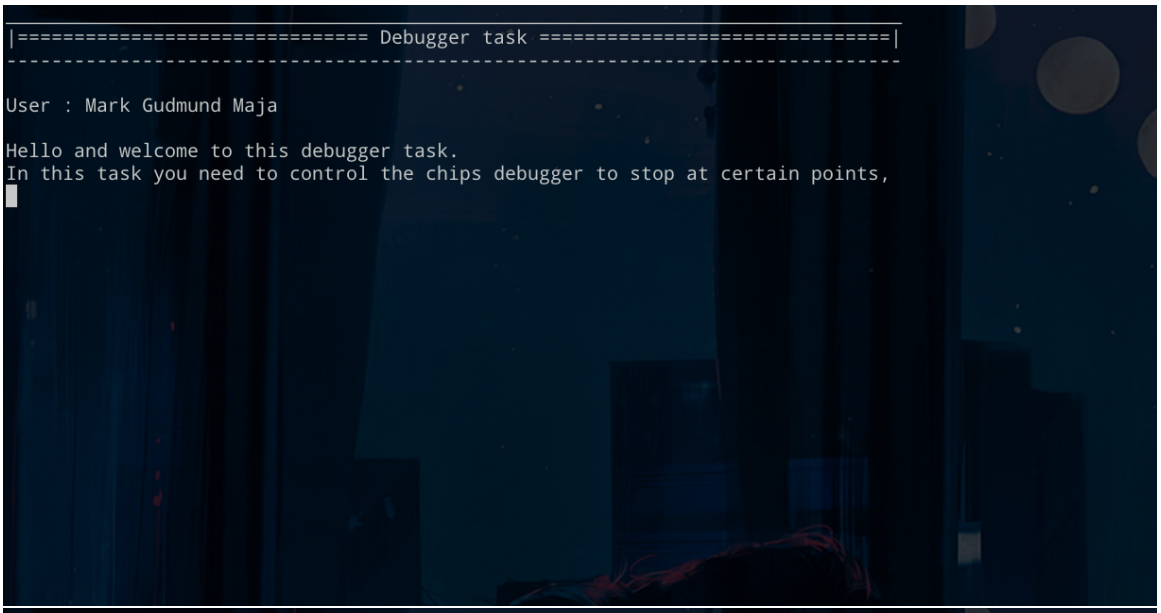
# Journal

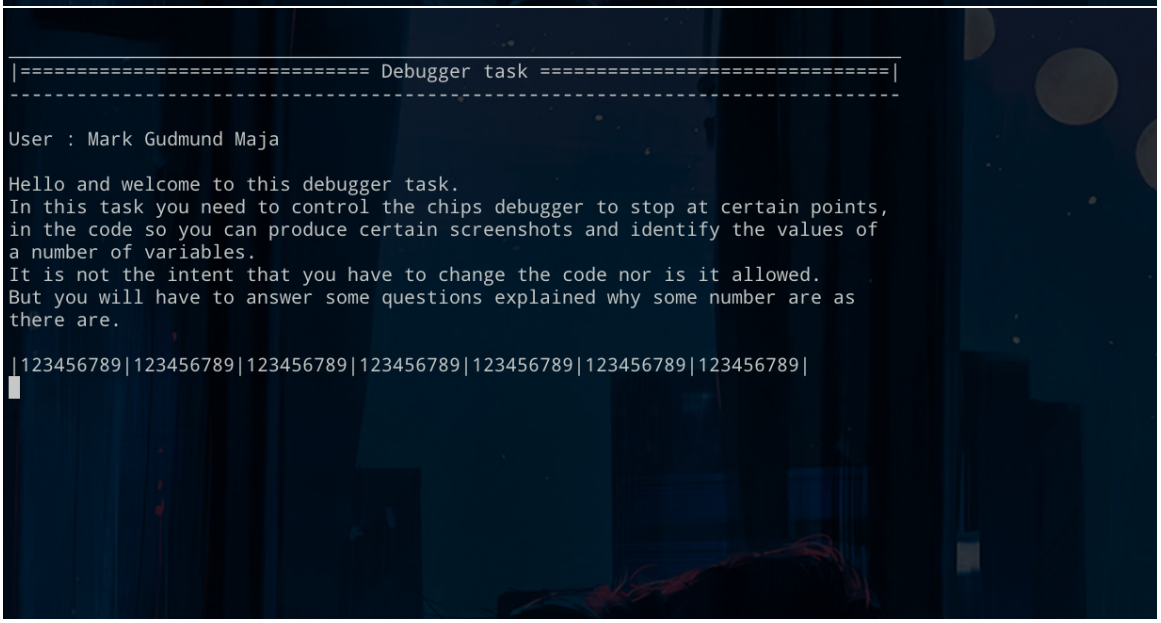
Polly

January 2022

# Dag 1

## Exercise 1.4




I stopped taking screenshots after this because my group mate had taken them already, but I finished the exercise

## Exercise 1.6

| | Unsigned | Signed-magnitude | Ones complement | Twos complement | biased |
|---|---|---|---|---|---|
| 56 | 00111000 | 00111000 | 00111000 | 00111000 | 10110111 |
| 178 | 10110010 | 00000000 10110010 | 00000000 10110010 | 00000000 10110010 | 10000000 10110001 |
| 1002 | 00000011 11101010 | 00000011 11101010 | 00000011 11101010 | 00000011 11101010 | 10000011 11101001 |
| 7586 | 00011101 10100010 | 00011101 10100010 | 00011101 10100010 | 00011101 10100010 | 10011101 10100001 |

| | Unsigned | Signed-magnitude | Ones complement | Twos complement | biased |
|---|---|---|---|---|---|
| -56 | - | 10111000 | 11000111 | 11001000 | 000001000111 |
| -178 | - | 10000000 10110010 | 11111111 01001101 | 11111111 01001110 | 01111111 01001101 |
| -1002 | - | 10000011 11101010 | 11111100 00010101 | 11111100 00010110 | 01111100 00010101 |
| -7586 | - | 10011101 10100010 | 11100010 01011101 | 11100010 01011101 | 01100010 01011101 |

## Exercise 2.1

```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"       // Input/output library for this course

void clrscr() {
        printf("\x1B[2J\x1b[H");
}

void clreol() {
        printf("\x1B[0K");
}

void gotoxy(int x, int y) {
        printf("\x1B[%d;%df", x, y);
}

void underline(int enable) {
        if (enable) {
                printf("\x1B[4m");
        } else {
                printf("\x1B[24m");
        }
}

void blink(int enable) {
        if (enable) {
                printf("\x1B[5m");
        } else {
                printf("\x1B[25m");
        }
}


void inverse(int enable) {
        if (enable) {
                printf("\x1B[7m");
        } else {
                printf("\x1B[27m");
        }
}

int main() {
        //char output[] = {'\xE2', '\x95', '\xA3'};
```

```
        char output[] = {'\xBA', '\xBA'};

        uart_init(9600);

        clrscr();
        printf("Hello! ! ! !");
        printf("%s", output);

        gotoxy(3,3);
        underline(1);
        printf("Hi!");
        underline(0);
        printf("Bye!");

        blink(1);
        printf("Blinking!!!");
        blink(0);
        printf("Not blinking...");

        inverse(1);
        printf("Inverse!!!");
        inverse(0);
        printf("Not inverse...");
}
```

## Exercise 2.2

```
int min(int a, int b) {
        return a<b ? a : b;
}

int max(int a, int b) {
        return a>b ? a : b;
}

struct WindowStyle {
        char top_left;
        char top_right;
        char bottom_left;
        char bottom_right;
        char horizontal;
        char vertical;
};

void window(int x1, int y1, int x2, int y2, char* name, struct WindowStyle* style) {
        // Both points have to be on screen
        if (x1 < 1 || y1 < 1 || x2 < 1 || y2 < 1) {
                // Oh no! Crash!
```

```
        // TODO: Check upper bound.
        printf("Points out of bounds!");
        exit(1);
}

// We turn these points into a top left corner (xl, yl) and bottom right (xh,
int xl = min(x1, x2);
int xh = max(x1, x2);
int yl = min(y1, y2);
int yh = max(y1, y2);

gotoxy(xl, yl);

// We include both points.
int width = xh - xl + 1;
// We want at least 2 places on both sides of the name.
if (strlen(name) + 4 > width) {
        // Oh no! Crash!
        printf("Not wide enough!");
        exit(1);
}

// Print 2 chars + name
putchar(style->top_left);
putchar(style->horizontal);
int printed = printf("%s", name);
// Print every char except the corner
for (int x = xl + printed + 2; x < xh; x++) {
        putchar(style->horizontal);
}
// Print corner
putchar(style->top_right);

// Sides (excluding the bottom side)
for (int y = yl + 1; y < yh; y++) {
        gotoxy(xl, y);
        putchar(style->vertical);
        gotoxy(xh, y);
        putchar(style->vertical);
}

// bottom side
gotoxy(xl, yh);
putchar(style->bottom_left);
for (int x = xl + 1; x < xh; x++) {
        putchar(style->horizontal);
}
putchar(style->bottom_right);
```

```
}

int main() {

        // printf("Hello! ");

        struct WindowStyle ascii = {
                .top_left = '+',
                .top_right = '+',
                .bottom_left = '+',
                .bottom_right = '+',
                .horizontal = '-',
                .vertical = '|',
        };

        window(10, 10, 40, 20, "Tom", &ascii);

        window(15, 14, 60, 22, "Jerry", &ascii);

}
```

# Dag 2

## Exercise 3.1-3

```
#include <stdio.h>
#include <stdlib.h>
#include "fixed.h"

static const char digits[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
void print_decimal(i32 n) {
    if (n < 0) {
        putchar('-');
        // undo 2's compliment
        n = abs(n);
    }
    printf("%i", WHOLE_PART(n));

    u64 number = DECIMAL_PART(n);
    if (number) {
        putchar('.');
    }

    while (number) {
        number *= 10;
        char digit = digits[WHOLE_PART(number)];
        putchar(digit);
        number = DECIMAL_PART(number);
```

```
    }
}

static const u16 cos_LUT[512]={
    0x4000, 0x3FFE, 0x3FFB, 0x3FF4, 0x3FEC, 0x3FE1, 0x3FD3, 0x3FC3, 0x3FB1, 0x3F9C, 0x
    0x3EC5, 0x3E9C, 0x3E71, 0x3E44, 0x3E14, 0x3DE2, 0x3DAE, 0x3D77, 0x3D3E, 0x3D02, 0x
    0x3B20, 0x3AD2, 0x3A82, 0x3A2F, 0x39DA, 0x3983, 0x392A, 0x38CF, 0x3871, 0x3811, 0x
    0x3536, 0x34C6, 0x3453, 0x33DE, 0x3367, 0x32EE, 0x3274, 0x31F7, 0x3179, 0x30F8, 0x
    0x2D41, 0x2CB2, 0x2C21, 0x2B8E, 0x2AFA, 0x2A65, 0x29CD, 0x2934, 0x2899, 0x27FD, 0x
    0x238E, 0x22E6, 0x223D, 0x2192, 0x20E7, 0x2039, 0x1F8B, 0x1EDC, 0x1E2B, 0x1D79, 0x
    0x187D, 0x17C3, 0x1708, 0x164C, 0x158F, 0x14D1, 0x1413, 0x1354, 0x1294, 0x11D3, 0x
    0x0C7C, 0x0BB6, 0x0AF1, 0x0A2A, 0x0964, 0x089C, 0x07D5, 0x070D, 0x0645, 0x057D, 0x
    0x0000, 0xFF37, 0xFE6E, 0xFDA5, 0xFCDD, 0xFC14, 0xFB4B, 0xFA83, 0xF9BB, 0xF8F3, 0x
    0xF384, 0xF2BF, 0xF1FB, 0xF137, 0xF074, 0xEFB1, 0xEEEF, 0xEE2D, 0xED6C, 0xECAC, 0x
    0xE783, 0xE6C9, 0xE611, 0xE55A, 0xE4A3, 0xE3EE, 0xE33A, 0xE287, 0xE1D5, 0xE124, 0x
    0xDC72, 0xDBCC, 0xDB26, 0xDA83, 0xD9E1, 0xD940, 0xD8A1, 0xD803, 0xD767, 0xD6CC, 0x
    0xD2BF, 0xD232, 0xD1A6, 0xD11D, 0xD095, 0xD00F, 0xCF8A, 0xCF08, 0xCE87, 0xCE09, 0x
    0xCACA, 0xCA5B, 0xC9EE, 0xC984, 0xC91B, 0xC8B5, 0xC851, 0xC7EF, 0xC78F, 0xC731, 0x
    0xC4E0, 0xC494, 0xC44A, 0xC403, 0xC3BE, 0xC37C, 0xC33B, 0xC2FE, 0xC2C2, 0xC289, 0x
    0xC13B, 0xC115, 0xC0F2, 0xC0D1, 0xC0B2, 0xC096, 0xC07C, 0xC064, 0xC04F, 0xC03D, 0x
    0xC000, 0xC002, 0xC005, 0xC00C, 0xC014, 0xC01F, 0xC02D, 0xC03D, 0xC04F, 0xC064, 0x
    0xC13B, 0xC164, 0xC18F, 0xC1BC, 0xC1EC, 0xC21E, 0xC252, 0xC289, 0xC2C2, 0xC2FE, 0x
    0xC4E0, 0xC52E, 0xC57E, 0xC5D1, 0xC626, 0xC67D, 0xC6D6, 0xC731, 0xC78F, 0xC7EF, 0x
    0xCACA, 0xCB3A, 0xCBAD, 0xCC22, 0xCC99, 0xCD12, 0xCD8C, 0xCE09, 0xCE87, 0xCF08, 0x
    0xD2BF, 0xD34E, 0xD3DF, 0xD472, 0xD506, 0xD59B, 0xD633, 0xD6CC, 0xD767, 0xD803, 0x
    0xDC72, 0xDD1A, 0xDDC3, 0xDE6E, 0xDF19, 0xDFC7, 0xE075, 0xE124, 0xE1D5, 0xE287, 0x
    0xE783, 0xE83D, 0xE8F8, 0xE9B4, 0xEA71, 0xEB2F, 0xEBED, 0xECAC, 0xED6C, 0xEE2D, 0x
    0xF384, 0xF44A, 0xF50F, 0xF5D6, 0xF69C, 0xF764, 0xF82B, 0xF8F3, 0xF9BB, 0xFA83, 0x
    0x0000, 0x00C9, 0x0192, 0x025B, 0x0323, 0x03EC, 0x04B5, 0x057D, 0x0645, 0x070D, 0x
    0x0C7C, 0x0D41, 0x0E05, 0x0EC9, 0x0F8C, 0x104F, 0x1111, 0x11D3, 0x1294, 0x1354, 0x
    0x187D, 0x1937, 0x19EF, 0x1AA6, 0x1B5D, 0x1C12, 0x1CC6, 0x1D79, 0x1E2B, 0x1EDC, 0x
    0x238E, 0x2434, 0x24DA, 0x257D, 0x261F, 0x26C0, 0x275F, 0x27FD, 0x2899, 0x2934, 0x
    0x2D41, 0x2DCE, 0x2E5A, 0x2EE3, 0x2F6B, 0x2FF1, 0x3076, 0x30F8, 0x3179, 0x31F7, 0x
    0x3536, 0x35A5, 0x3612, 0x367C, 0x36E5, 0x374B, 0x37AF, 0x3811, 0x3871, 0x38CF, 0x
    0x3B20, 0x3B6C, 0x3BB6, 0x3BFD, 0x3C42, 0x3C84, 0x3CC5, 0x3D02, 0x3D3E, 0x3D77, 0x
    0x3EC5, 0x3EEB, 0x3F0E, 0x3F2F, 0x3F4E, 0x3F6A, 0x3F84, 0x3F9C, 0x3FB1, 0x3FC3, 0x
};

#define DEGREES(n) (n*512/360)

// 512 for this function
i16 cos_fast(int angle) {
    if (angle >= 0) {
        return cos_LUT[angle % 512];
    } else {
        return cos_LUT[-angle % 512];
    }
}
```

```
i16 sin_fast(int angle) {
    return cos_fast(angle - 128);
}

void rotate_vector(vector_t *vec, int angle) {
    i32 tmp_x = vec->x;
    i32 tmp_y = vec->y;

    vec->x = FIXED_MUL(tmp_x, cos_fast(angle)) - FIXED_MUL(tmp_y, sin_fast(angle));
    vec->y = FIXED_MUL(tmp_x, sin_fast(angle)) + FIXED_MUL(tmp_y, cos_fast(angle));
}
```

## Exercise 3.4

0x1E:
You shift it left by 8.
0x1F:
Doing so loses information, so there's many ways to do it. One would be to round down by shifting the value right 8 times and taking the bottom 8 bits.
0x20:
0x0001 = $2^{-8} = \frac{1}{256} = 0.00390625$
0x21:
You get their sum, (might overflow)
0x22:
You get their product multiplied by $2^8$, or if your calculation overflowed some pseudo-random garbage.
0x23:
You get their product, (might still overflow)
0x24:
Well doing `(char) (fixed_point >> 8)` gives you a value rounded towards 0, but that might not be what you want.
0x25:
If you want it to round to the nearest whole number, with .5 rounding up you could simply check the most significant bit of the decimal part, and plus your `(char) (f >> 8)` value by 1 if it's 1. At least for positive values.
0x26:
0x00.20
0x27:
0x00.40
0x28:
0x02.20
0x29:
0x02.40

0x2A:
0x00.08
0x2B:
It overflow

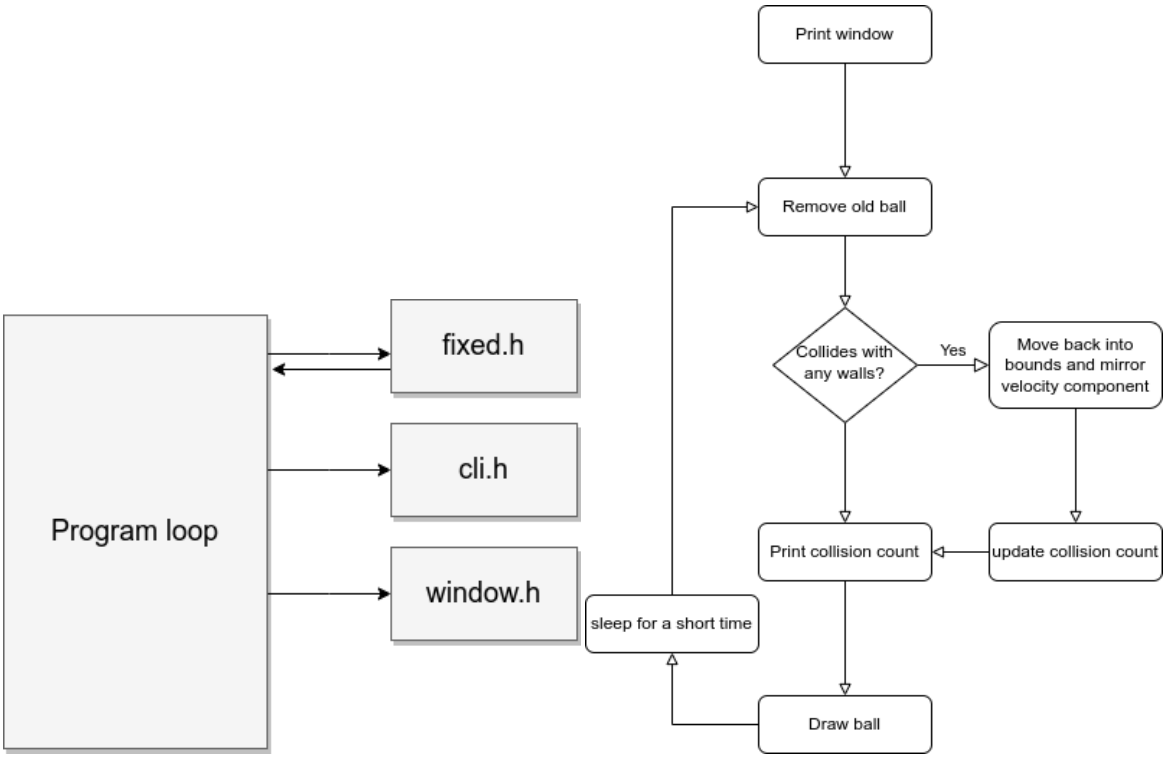
0x2D:
We can use 16-bit temporary variables to compute the multiplication and then shift it back after.

## Exercise 4

```
#include <unistd.h>
#include <stdio.h>
#include "window.h"
#include "fixed.h"
#include "cli.h"

static style_t ascii = {
    .top_left = "+",
    .top_right = "+",
    .bottom_left = "+",
    .bottom_right = "+",
    .horizontal = "-",
    .vertical = "|",
};

static style_t cp850_double = {
    .top_left = "\xC9",
    .top_right = "\xBB",
    .bottom_left = "\xC8",
    .bottom_right = "\xBC",
    .horizontal = "\xCD",
    .vertical = "\xBA",
};

static style_t cp850_single = {
    .top_left = "\xDA",
    .top_right = "\xBF",
    .bottom_left = "\xC0",
    .bottom_right = "\xD9",
    .horizontal = "\xC4",
    .vertical = "\xB3",
};

void main() {
    int xl = FIXED_NUM(10, 0);
    int yl = FIXED_NUM(10, 0);
    int xh = FIXED_NUM(40, 0);
    int yh = FIXED_NUM(20, 0);
    window(WHOLE_PART(xl), WHOLE_PART(yl), WHOLE_PART(xh), WHOLE_PART(yh), "Besties",

    vector_t pos = {FIXED_NUM(17, 0), FIXED_NUM(15, 0)};
    vector_t vel = {FIXED_ONE/10, FIXED_ONE/10};
```

```
    int collisions = 0;

    while (1) {
        // Remove old ball
        gotoxy(WHOLE_PART(pos.x), WHOLE_PART(pos.y));
        putchar('␣');

        // Update pos
        pos.x += vel.x;
        pos.y += vel.y;

        // Check collisions
        if (pos.x >= xh) {
            pos.x = xh - FIXED_ONE;
            vel.x *= -1;
            collisions++;
        }
        if (pos.x <= (xl + FIXED_ONE)) {
            pos.x = xl + FIXED_ONE;
            vel.x *= -1;
            collisions++;
        }

        if (pos.y >= yh) {
            pos.y = yh - FIXED_ONE;
            vel.y *= -1;
            collisions++;
        }
        if (pos.y <= (yl + FIXED_ONE)) {
            pos.y = yl + FIXED_ONE;
            vel.y *= -1;
            collisions++;
        }

        // Draw
        // Counter
        gotoxy(WHOLE_PART((xl+xh)/2), WHOLE_PART((yl+yh)/2));
        printf("%d", collisions);

        // Ball
        gotoxy(WHOLE_PART(pos.x), WHOLE_PART(pos.y));
        putchar('o');

        gotoxy(0,0);
        fflush(0);
        usleep(100);
    }
}
```

### Exercise 4.3




## Dag 3

### Exercise 5

```
#include "stm32f30x_conf.h"

// Turns the GPIO pin into an input
void set_input(GPIO_TypeDef *gpio, char pin) {
    gpio->MODER &= ~(0x00000003 << (pin * 2)); // Clear mode register
    gpio->MODER |= (0x00000000 << (pin * 2)); // Set mode register (0x00      Input, 0x
    gpio->PUPDR &= ~(0x00000003 << (pin * 2)); // Clear push/pull register
    gpio->PUPDR |= (0x00000002 << (pin * 2)); // Set push/pull register (0x00 - No pul
}

char read_input(GPIO_TypeDef *gpio, char pin) {
    return (gpio->IDR >> pin) & 1;
}

// Turns the GPIO pin into an output
void set_output(GPIO_TypeDef *gpio, char pin) {
    gpio->OSPEEDR &= ~(0x00000003 << (pin * 2)); // Clear speed register
```

```
    gpio->OSPEEDR |= (0x00000002 << (pin * 2)); // set speed register (0x01 - 10MHz, 0
    gpio->OTYPER &= ~(0x0001 << (pin * 1)); // Clear output type register
    gpio->OTYPER |= (0x0000 << (pin * 1)); // Set output type register (0x00 - Push pu
    gpio->MODER &= ~(0x00000003 << (pin * 2)); // Clear mode register
    gpio->MODER |= (0x00000001 << (pin * 2)); // Set mode register (0x00      Input, 0x
}

void write_output(GPIO_TypeDef *gpio, char pin, char value) {
    if (value) {
        gpio->ODR |= (1 << pin);
    } else {
        gpio->ODR &= ~(1 << pin);
    }
}

// Order is Center, Right, Left, Down, Up
// 0bRLDU
char read_joystick() {
    return read_input(GPIOA, 4)
    | (read_input(GPIOB, 0) << 1)
    | (read_input(GPIOC, 1) << 2)
    | (read_input(GPIOC, 0) << 3)
    | (read_input(GPIOB, 5) << 4);
}

void init_joystick() {
    RCC->AHBENR |= RCC_AHBPeriph_GPIOA; // Enable clock for GPIO Port A
    RCC->AHBENR |= RCC_AHBPeriph_GPIOB; // Enable clock for GPIO Port B
    RCC->AHBENR |= RCC_AHBPeriph_GPIOC; // Enable clock for GPIO Port C

    // Joystick
    // UP     => A2 => PA_4
    // DOWN   => A3 => PB_0
    // LEFT   => A4 => PC_1
    // RIGHT  => A5 => PC_0
    // CENTER => D5 => PB_5
    set_input(GPIOA, 4);
    set_input(GPIOB, 0);
    set_input(GPIOC, 1);
    set_input(GPIOC, 0);
    set_input(GPIOB, 5);
}

void init_rgb_led() {
    RCC->AHBENR |= RCC_AHBPeriph_GPIOA; // Enable clock for GPIO Port A
    RCC->AHBENR |= RCC_AHBPeriph_GPIOB; // Enable clock for GPIO Port B
    RCC->AHBENR |= RCC_AHBPeriph_GPIOC; // Enable clock for GPIO Port C
```

```
    // tricolor RGB blue
    // (R, G, B) = (PB4, PC7, PA9)
    set_output(GPIOB, 4);
    set_output(GPIOC, 7);
    set_output(GPIOA, 9);
}

// 0b111 is white
// 0bRGB
void set_led(char color) {
    write_output(GPIOB, 4, !color & 0b100);
    write_output(GPIOC, 7, !color & 0b010);
    write_output(GPIOA, 9, !color & 0b001);
}
```

## Exercise 6

```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h"           // Input/output library for this course
#include "cli.h"
#include "joystick.h"

volatile int32_t centiseconds = 0;

// Timer 2's interupt
void TIM2_IRQHandler() {
    centiseconds++;

    TIM2->SR &= ~0x0001;
}

int print_formated_centiseconds(int32_t now) {
    char centis = now % 100;
    char seconds = (now / 100) % 60;
    char minutes = (now / 6000) % 60;
    char hours = (now / 360000) % 24;
    printf("%02d:%02d:%02d.%03d\n", hours, minutes, seconds, centis);
}

int main(void) {
    lcd_init();
    uart_init(9600);
    init_joystick();

    clrscr();

    uint8_t buffer[512];
    for (int i = 0; i < 512; i++) {
```

```
    buffer[i] = (uint8_t) i;
}

lcd_push_buffer(&buffer);

// Set up clock with 100Hz frequency
RCC->APB1ENR |= RCC_APB1Periph_TIM2; // Enable clock line to timer 2
// No remapping, No clock division, Not buffered, Edge-aligned, Up-counting mode,
// One-pulse mode disabled, Any update request source, update events disabled
TIM2->CR1 = 0x0000;
TIM2->ARR = 640000 - 1;
TIM2->PSC = 0;
// Enable interrupt
TIM2->DIER |= 0x0001;
NVIC_SetPriority(TIM2_IRQn, 2);
NVIC_EnableIRQ(TIM2_IRQn);

// Enable timer
TIM2->CR1 = 0x0001;

int32_t split1_acc = 0;
int32_t split2_acc = 0;

int32_t split1_offset = 0;
int32_t split2_offset = 0;

// 0 is no one selected, 1 is split 1, 2 is split 2
char selected = 0;

while(1){
    int input = read_joystick();

    if (input == 0b10000) {
        TIM2->CR1 ^= 0x0001;
    } else if (input & 0b0100) {
        // Split time 1
        if (selected == 2) {
            split2_acc += centiseconds - split2_offset;
        }

        if (selected != 1) {
            split1_offset = centiseconds;
        }
        selected = 1;
    } else if (input & 0b1000) {
        // Split time 2
        if (selected == 1) {
            split1_acc += centiseconds - split1_offset;
```

```
            }

            if (selected != 2) {
                split2_offset = centiseconds;
            }
            selected = 2;
        } else if (input & 0b0010) {
            // Stop clock and set time to 0:00
            TIM2->CR1 = 0x0000;
            centiseconds = 0;
            split1_acc = 0;
            split2_acc = 0;
            selected = 0;
        }

        // Roughly every second we print the current time
        if (centiseconds % 10 == 0) {
            gotoxy(1,1);
            printf("Time:    ");
            print_formated_centiseconds(centiseconds);

            printf("Split 1: ");
            if (selected == 1) {
                print_formated_centiseconds(split1_acc + centiseconds - split1_offset)
            } else {
                print_formated_centiseconds(split1_acc);
            }

            printf("Split 2: ");
            if (selected == 2) {
                print_formated_centiseconds(split2_acc + centiseconds - split2_offset)
            } else {
                print_formated_centiseconds(split2_acc);
            }
        }
    }
}
```