MES: Plant Controller Report

Friedemann Drepper

November 19, 2022

Contents

1 Application Description

Plant Controller is a watering/irrigation system used to water plants at home. When active, it checks the soil moisture around the plant. If the moisture level is below a preset value it will water the plant until the soil is moist again. When inactive for a while, it will go to sleep. Every 10 minutes it wakes up and checks for the soil moisture, the water level of the tank and then goes back to sleep again. If the soil moisture is below the set value, it will water the plant and go into active mode. If the tank is empty, the system will notify the user by blinking an LED and displaying a warning message on the LCD screen while also preventing the system from working until the tank is filled again. The system needs the user to push the button for a few seconds to tell it, the tank has been filled again.

The user has two ways to interact with the system. First, by pressing a button the system gets active (when in sleep mode) and displays information on an LCD screen which you can cycle through by pressing the button again. A long press on the button will cause the system to go into sleep mode. Second, by interfacing to it with a UART Terminal, that is used for debugging and setting the moisture level at which the system should water the plant.

For the watering itself, the system uses a small tank and a submersible water pump. The water level in the tank is monitored by an ultrasonic distance sensor. If the tank is empty, the system will notify the user by blinking an LED and displaying a warning message on the LCD screen. The distance at which the system considers the tank to be empty is set via the UART Terminal.

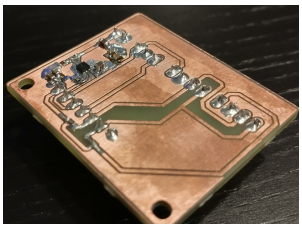
The system also makes predictions, based on previous times, about how long the remaining water in the tank will last. This information is also available on the LCD Screen by cycling through it with the button or by requesting them over the UART Terminal. It can be used to calculate how much water the plant will probably need for example if you go on vacation.

To find out what soil moisture level is correct you need to calibrate the system with experimental data of dry and moist soil and then figure out a good moisture value.

The full state machine of the system can be found here.

2 Hardware Description

The core of the system is the processor, a STM32F446RE. For this project I used the Nucleo-F446 which is a development board. Hardware wise the Nucleo includes a Button and a LED I use. It is connected to an I^2C Multiplexer (PCF8574A) used to drive a LCD Screen that only has a parallel interface. On the same I^2C Bus is the Ultrasonic Distance Sensor (HC-SR04P) which supports I^2C in newer editions though there is no official documentation on it. The I^2C Bus has two 4k7 pull up resistors. Further, the system has a capacitive soil moisture sensor and a submersible water pump. The pump requires round about 100mA at 3V. To achieve this I used a LDO IC (NCP718) which made me design and build a simple prototype board which also includes all connectors from Nucleo to Board and Board to Sensors, Pump and LCD Screen.



Finally, here you can see the whole system assembled:

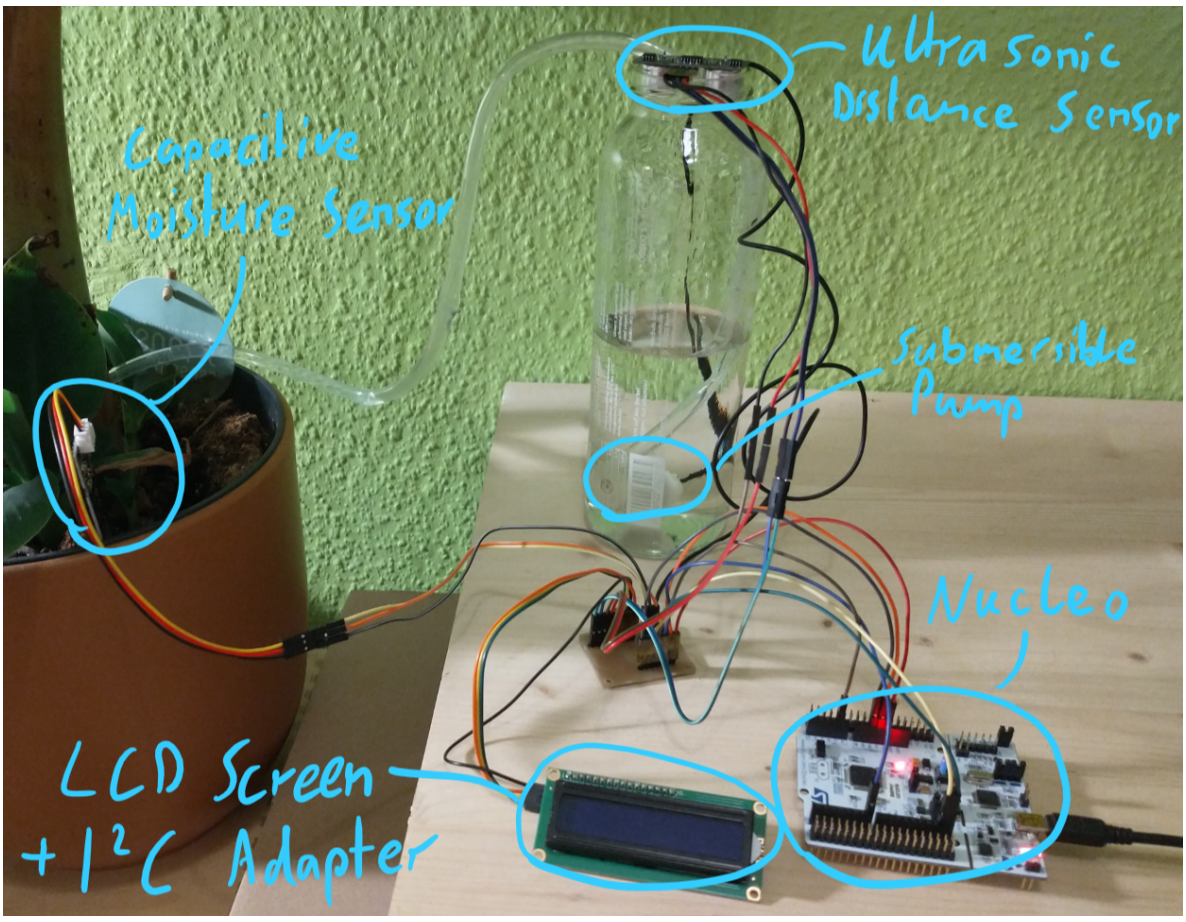


Figure 1: Plant Controller System Setup

TODO: ADD LINKS TO PARTS AND IMAGES OF ASSEMBLED HW

3 Software Description

Starting on a high level, the systems behaviour is documented in its state machine.

	States (Rows) / Events (Columns)	0 Idle Timeout [0]	1 Long Button Press [1]	2 Tank Empty [2]	3 Tank Not Empty [3]	4 Soil is dry [4]	5 Soil is Wet [5]	6 RTC wakeup [6]	7 Error [7]	Description
0	ACTIVE	SLEEP	ACTIVE	ERROR TANK EMPTY	ACTIVE	WATERING	ACTIVE	ACTIVE	SYSTEM_ERROR	System is active, responds to incoming messages, shows information on LCD and check for water and moisture level
1	SLEEP	SLEEP	ACTIVE	SLEEP	SLEEP	SLEEP	SLEEP	PERIODIC CHECK	SYSTEM_ERROR	Sleep mode, can be either woken up by the RTC to perform a periodic check of the system, by button press or by sending a message over Terminal
2	PERIODIC CHECK	SLEEP	SLEEP	ERROR TANK EMPTY	SLEEP	WATERING	SLEEP	SLEEP	SYSTEM_ERROR	From Sleep mode the System performs a periodic check on water level and soil moisture every few minutes and goes back to sleep afterwards
3	WATERING	WATERING	WATERING	ERROR TANK EMPTY	WATERING	WATERING	ACTIVE	WATERING	SYSTEM_ERROR	Watering the Plant, always transitioning to active mode when done
4	ERROR TANK EMPTY	ERROR TANK EMPTY	ACTIVE	ERROR TANK EMPTY	ERROR TANK EMPTY	ERROR TANK EMPTY	ERROR TANK EMPTY	ERROR TANK EMPTY	SYSTEM_ERROR	When the Tank is empty, the system goes into error mode, only returning back to normal when the user tells they refilled the tank.
5	SYSTEM_ERROR	SYSTEM_ERROR	SYSTEM_ERROR	SYSTEM_ERROR	SYSTEM_ERROR	SYSTEM_ERROR	SYSTEM_ERROR	SYSTEM_ERROR	SYSTEM_ERROR	Something failed, stop everything. Will only leave this mode by resetting.

Figure 2: System State Machine

It is an event-centric state machine where the events are the user interactions and the sensor readings. This means that the system always needs to keep track of the sensors and compare them to the set values for tank emptiness and soil dryness. The system's state machine also is implemented in it to keep track of what mode it is in right now.

I generated all the code regarding system setup, interrupt handlers and peripherals with the CubeMX tool by ST. I also use the CubeHAL and LL drivers to program the hardware, though I created subsystems that work as adapters to be platform independent, allowing easier migration to another processor by ST (or even another vendor). The CMSIS drivers used by the HAL use the Apache License and the CubeHAL uses the BSD-3-Clause License Talking of vendors, inside the `vendor` folder are all third party libraries found I used. I use the console and logging library from anchor. Ceedling is the testing framework I use for Unit Tests by ThrowTheSwitch. All three of these are under the MIT License. The only two files I changed inside the `Core` folder, which is the generated code, are `main.c` which contains the main and `stm32f4xx_it.c` which contains the interrupt. All files in `inc` and `src` are my code, containing sensor drivers, subsystems, modules and controllers. The Unit Tests are found in `test`.

Now I am coming to my own software, starting with the Hierarchy of the system. At the top are the system control and the system events. The system control includes the main loop and the state machine table. The system events are used to trigger events and interface them to the system control with a simple priority system.

Below that are the controllers which are the most interesting point of the whole system, as they incorporate the actual functions used to run the system. The controllers (nearly) all have a pair of functions they have to implement: `Init` and `Enable`. `Init` is pretty self explanatory. `Enable` has a Boolean parameter used to enable or disable the controller's part of the system. The controllers are also the one to trigger events by calling the `System_Event_Trigger_Event` function with their event (exported in the system event header).

Next are the modules, which are used to abstract a piece of hardware into an object. The modules are the 'classes' and instances of them are to be found in the controller that uses the module. At the bottom are the subsystems which are adapters on top of the CubeHAL and LL drivers by ST to make my code as independent of them as possible.

Now to the parts that exist outside of the above described hierarchy. First we have the `logging` and the `shell` module. These both are the interfaces to third party libraries for a console and logging respectively. The shell module is initializing the console by supplying the write function. It is also the file where all command definitions are found. The only relevant exported functions are the Init

function and the Shell Read function. The latter needs to be called whenever there is nothing else to do. In the system it is called in every pass of the main loop (to be found in system control). For the sake of simplicity I am using the UART peripheral in blocking read mode. In the future that surely needs to be changed to be in Interrupt or DMA mode, using a circular buffer. The logging module is initializing the logging by supplying the write function and it exports the macro to log a message. The log module colors header is only exporting some ASCII codes for a nicely colored logging output. Lastly there is the version file, which is an idea I got from my workplace, though not finished. The Idea is to have stored in the firmware, what git commit and branch it is. The only thing the file is doing in my case is supplying a reset log function called on every system startup or reset.

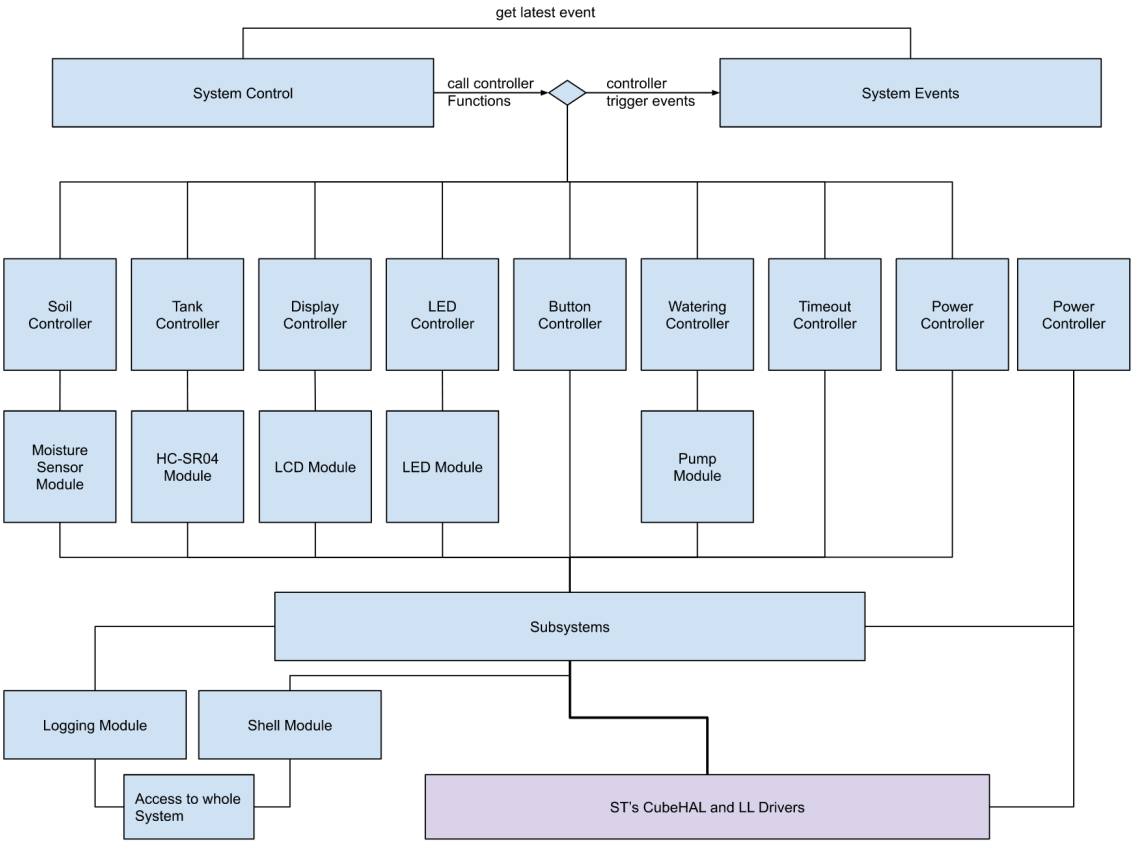


Figure 3: Software Hierarchy of Control Diagram

That being the overview, now I will go back to the top again, explaining some of the parts in more detail. I will not thoroughly go through the subsystems as most of them are very simple adapters. Only thing to mention here might be, that in their Inits are the calls to the CubeMX Init functions to be found. Another thing worth mentioning is the weird date poll inside the RTCs get time function. That is a RTC specific thing, that needs to be done to unlock the RTCs shadow registers.

Here I will also briefly explain how I use Interrupts. Inside the subsystem headers I declare weak functions (meaning they don't need to be implemented to compile) as callbacks. In the CubeMX generated interrupt file (`Core/stm32f4xx_it.c` which contains the IRQ Handler, I have added calls to these functions. The definition of the callbacks are inside the controllers that use the respective

peripheral.

The modules are more complex than the subsystems but also not too interesting or confusing. Therefore I will only explain the sr04 module (the ultrasonic distance sensor). The SR04 module exports a start measurement and a read measurement function. These have to be called one after the other with a delay in between. The first one basically tells the sensor to start a measurement and the second reads the result and saves it into the object. Though the LCD module is complex, I'll not get into it in further detail here, because that is an implementation related to the data sheet and too much to explain here. Worth mentioning is, that I have to use the LCD in 4 bit mode due to the 8bit I^2C GPIO Expander I use to drive the parallel Interface.

Now the controllers. Let's start with the Exception to the rule of not using the CubeHAL or LL drivers, the **power controller**. It's task is to put the system in Stop mode. That will shut down most of the system but preserves the RAM. When woken up the execution will continue after the Instruction that put it to sleep. First the system clock needs to be reconfigured. Then the wakeup source is determined by reading the Button Pin (which is active low). Dependent on wakeup source the corresponding event is triggered. Handling the power modes of a microcontroller can be quite delicate which is why you should use the HAL if provided (and it gives enough flexibility). Though adapting the HAL is basically senseless as every processor can be different in what modes the have and how they enter them.

Next we have the **button controller**. This one is used to differentiate between a long and a short button press and also debounce the button signal. Therefore I start a timer. When that has finished I check if the button is still pressed to filter out bounces. If so I restart the timer with a new period, leading into the second stage. The second stage checks if the button is still pressed and if not cycles the information displayed on the LCD. Otherwise the timer restarts again with a new period for the third an last stage in which a long button press gets checked and triggered as event if still pressed. Any button press also resets the idle timeout.

The **led controller** is by far the simplest one, only caring about the one LED of the system. It provides functions to turn off the led and change its blinking mode.

Not as simple are the **soil** and **tank controllers**. These are the controllers that contain the sensor reading and triggering events depending on the measurements. They also provide functions to get the current measured values and getter and setter for the limits compared to to generate the empty or wet/dry events. Updating the measured value and generating the event happens in the `Update` function both of them provide.

Together with those two, the **watering controller** is the part of the system that takes care of the main application, watering a plant. To do this, the watering controller activates the pump for a short amount of time. The system control then delays for some time to let the water get into the soil and then checks for soil moisture. This gets repeated until the soil is wet again.

The **display controller** handles the LCD, supplying functions to display specific status information (like error messages or warnings). Other than that, the controller mainly cares about showing information on the display and provides a function to cycle through them (done in the button controller). It also provides a function to turn off the screen (used when the system is in Stop Mode).

The **timeout controller** cares about putting the system to sleep mode after some amount of time (5 minutes at the moment) and is realised by a timer. Whenever there is a user interaction with the system, the timeout gets reset. The controller supplies a function for that and a function to disable the idle timeout completely used to disable it when leaving the active state and re enabling it when entering it.

Lastly we have the **prediction controller**, which provides functions needed to save the time between

watering (done with a 1Hz timer that is used to count up seconds since last watering) and the tank level difference before and after a watering. It saves the last x times and tank level differences and when polled calculates how many hours the system should be able to keep running by simply averaging over those x values and using that to calculate the future consumption of water and how much time passes between watering.

To finish with the core of the whole project, system events and system control. First, **system events** is exporting an enum with all events that can be generated. Whenever an event is triggered it gets checked against the currently last generated event and if it has a higher priority the latest event gets updated. Reading the latest event also clears it which is sometimes necessary (for example when the Stop mode is used outside the sleep state).

The heart of it all, the **system control**, contains the system state machine and the main loop. The state machine implementation consists of three parts:

- The state machine table as a two dimensional array which can be indexed with a state and an event.
- A function that gets the event from system events and returns the next state when an event occured.
- An Enter, Execute and Exit function set which are self explanatory

These four functions get called inside the main loop of system control, though enter and exit only get called when there is a state transition. The other thing done in the main loop is to call the shell read function as mentioned above, which, if reading a character, resets the idle timeout.

The hierarchy introduced in the beginning is not a strict one, as controllers do interact with each other without going the way through the system state machine. Examples for that are the idle timeout getting reset in the button controller and the display controller getting values from tank and soil controllers.

For a visial overview see the software block diagram:

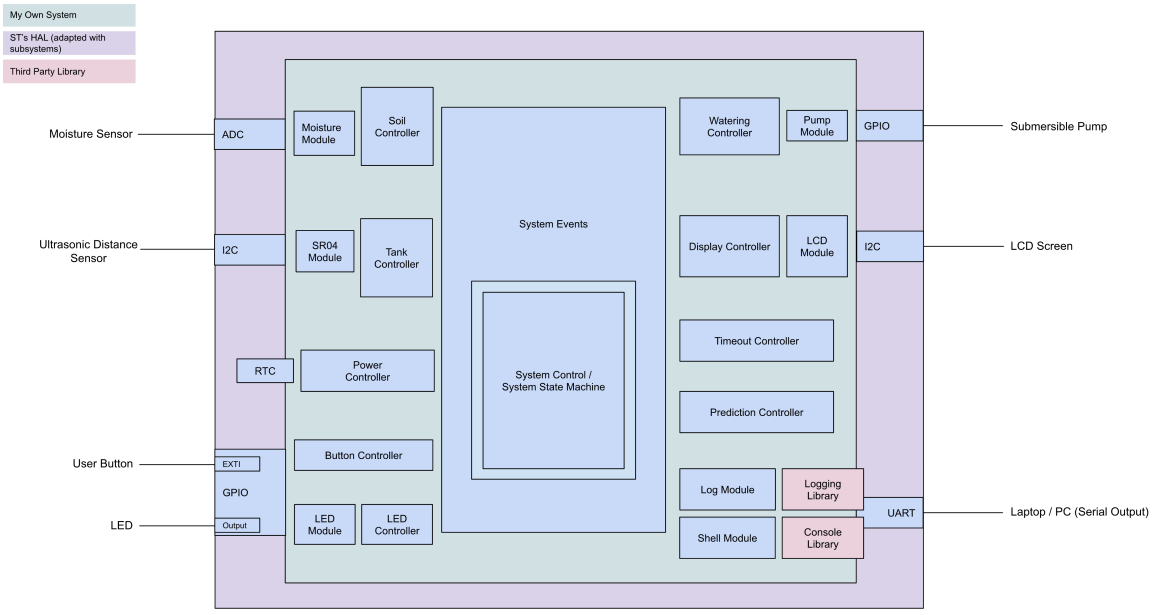


Figure 4: Software Block Diagram

4 Build Instructions

I am using the CubeMX Tool by ST to generate code setting up peripherals, startup code, linker scripts etc. For compiling and debugging I use the `arm-none-eabi` toolchain (`gcc` and `gdb`) and to flash my software to the microcontroller i am using OpenOCD. All of these are integrated into the CLion IDE that I use. All my code is written in C, so there is no need for any C++ compiler. My CMakeLists is customized to support a project file structure that separates generated code and third-party libraries from self written code. For further detail about the file structure, see the Software Description section. As operating system for my build platorom I use MacOS and as microprocessor I use the STM32F446RE mounted onto the corresponding Nucleo Board (Nucleo-F446RE).

For Unit Tests, I use the Ceedling framework by ThrowTheSwitch.

I mainly used Logging to debug my system because I was not able to step through my code via OpenOCD. Though i used the debugger to see where my code gets stuck.

I power the System over the USB Port of the attached ST-Link. In future that should changed to be able to get rid of the ST-Link and also make more precise power calculations. For a detailed power analysis of the system, see here.

5 Future

Some Ideas what could be done to enhance the project on a larger scope:

- Using the F-Series by ST is an overkill for such a simple system, especially regarding power consumption. Therefore using the L-Series of processors by ST which are designed for low power applications would make a lot of sense. For ease of completing the project I decided to go with a board I already had. Because of this, I am already trying to make the main system logic as independent of the underlying chip as possible.
- In future the it'd also make sense if the system would be battery powered. For plants on a balcony or in a garden, adding a solar cell to charge the battery would make sense.
- The system could send its information wireless to a central system like a raspberry pi, which makes the information online available to monitor the plants remotely. Even adding a custom watering command would be possible.
- As a hobby project the form of prototyping it with development boards and breadboards is a good and easy way, though if the system should be commercially available in the future all the circuitry should go on one PCB (having the moisture sensor attached). Also the tank, pumps and tubes should be sold with it, making it easier to use and give room for making it compact and easy to set up.
- The system could make use of more settings, plant profiles etc.

On a smaller scope, there are some issues at the moment that'd need to be fixed and some small details that would make a lot of sense to do:

- When the SR04 distance sensor is disconnected the system locks up on startup.
- The system gets locked up in watering mode due to the prediction controller time stamp function.
- The UART communication works in blocking read mode right now, reading a character on every pass of the main loop with a timeout. That is not senseful and should be replaced by an Interrupt or DMA based system using circular buffers. A good lightweight ring buffer is lwrb that I intended to use but didn't have time to.

6 Self Assessment

Criteria: Project meets minimum requirements. **My Score:** 2.5.
Comment: I think my project does exceed expectations. It uses more peripherals than needed and has a very well documented state machine. The *Algorithmic* part of my system is not too interesting as my system is more dedicated to do something but the prediction Idea is not too bad. The command line interface would need a rework.

Criteria: Completeness of deliverables. **My Score:** 3.
Comment: Video was supplied in the live demo session. The report is close to complete. I left some parts out in the software section due to amount and simplicity. Most of the software is understandable with some exceptions due to hardware specifics (e.g. the power modes of the MCU).

Criteria: Clear intentions and working code. **My Score:** 2.5.
Comment: The system performs as described in the report. There are some known bugs (and probably many more unknown to be realistic). I think the Code is pretty understandable. One part of the system does not work currently, the prediction controller locks the system in watering mode, hence not 3 points.

Criteria: Reusing code. **My Score:** 2.5.
Comment: I have used a lot of code from other while also doing a lot by myself. I used a logging and a console library, ST's HAL and LL Drivers and the CubeMX tool to generate all the setup code. All those have licenses, that are included in their directories. I have not added a license for my own code. My code is pretty easy to differentiate from others code due to file structure. The only exceptions are described in this report.

Criteria: Originality and scope of goals. **My Score:** 2.
Comment: I have definitely not made a very original project. Though i have tried to add some interesting parts to one of the example and tried to implement and design it properly.

Criteria: Bonus Power Analysis. **My Score:** 2.
Comment: I have added a full power analysis, though it is not completely accurate as it is based on some estimations and also ignores the face that I use a Nucleo instead of only the Processor.

Criteria: Bonus Version Control. **My Score:** 3.
Comment: I have used git as version control system and used the gitflow model for my project. I have some bulk commits and sometimes varied from the underlying structure of gitflow, but mostly followed it. You can see how I incrementally built my system part by part over the past weeks.