University of Minho

Master's degree in Industrial Electronic Engineering and Computers

Embedded Systems

Aquarium Control

Authors

João Araújo PG53922

Tiago Sousa PG54261

Professor:

Adriano Tavares

January 2024

Agenda

List of Acronyms

ACK	Acknowledge
CO2	Carbon Dioxide
CPU	Central Processing Unit
DBMS	Database Management System
FIFO	First-In First-Out
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HMI	Human Machine Interface
I2C	Inter-Integrated Circuit
IPC	Inter-Process Communication
OS	Operating System
QML	Qt Modeling Language
RDBMS	Relational Database Management System
RTC	Real-Time Clock
SDA	Serial Data Line
SDLC	Software Development Life Cycle
SCL	Serial Clock Line
SSD	Solid-State Drive
SQL	Structured Query Language
TDS	Total Dissolved Solids

List of Figures

List of Tables

1 | Introduction

The aquarium environment for fish, whether saltwater or freshwater, requires systematic and strict care so the ideal survival conditions can be assured as well as the well-being of the fish. Water temperature, pH, light, and food are fundamental factors for the health and quality of life of the fish.

1.1 Context and Motivations

The specific requirements of every fish species, the number of habitants in the aquarium and the constant monitoring of the water conditions are some difficulties that most of the aquarium owners might find stressful and a deal breaker, leading to neglect on the owner's part and the death of the fish.

This demonstrates the need for an embracing solution that allows the users to efficiently control the aquarium environment with minimal effort.

1.2 Problem Statement

Nowadays, more and more people tend to spend less time at home and have more worries, which leads them to automate their home as much as possible so they can make the most of their free time.

In short, in order to promote a better environment for the fish, regulating crucial characteristics of the aquarium while taking into consideration every species living in the aquarium, the "Aquarium Control" will be an automatic aquarium control system capable of monitoring and controlling the quality of water and food according to the fish species, with a user-friendly interface.

2 | Analysis

The product to develop should be capable of reading the data provided by the user and collecting only the information related to it. The HMI (Human Machine Interface) must be simple and clear so that everyone can use it effortlessly, like a touch display. The system will use different sensors to measure the temperature, pH and quality of the water and lighting of the aquarium.

Both information acquired by the HMI and by the sensors are interpreted by a processing unit, in this case a Raspberry Pi, comparing the gathered data with reference data from a specific database. If any disruption is detected by the processing unit, it activates the corresponding actuator or sends a warning message to the HMI. An overview of the described system can be seen below (Fig. 2.1).

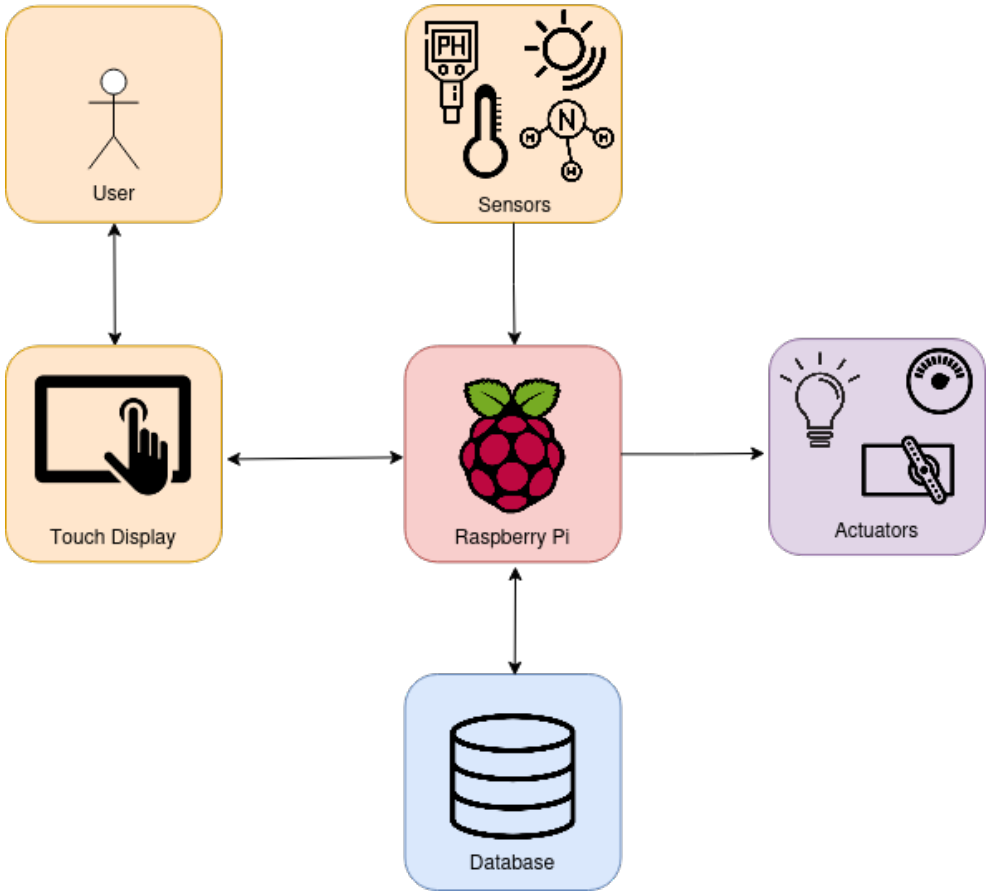

Figure 2.1: Overview System.

3 | Requeriments

There are two types of project requirements, functional and non functional.

Functional requirements specify ehat the system should do in terms of specific functionalities, describing actions and behaviors.

Non functional requirements define how the system should perform, describing user experience, reliability and system characteristics.

3.1 Functional Requirements

- Measure water quality, pH, temperature and lighting
- Control water temperature
- Control the amount of food provided
- Control aquarium lighting
- Allow the user to select the fish species and quantity in the aquarium
- Interact with a complete and always available database

3.2 Non Functional Requirements

- Efficient energy consumption
- Low budget
- Provide a user-friendly touch display
- Simple, intuitive and secure operation
- Precise measurements of all sensors
- Maintainability

4 | Constraints

There are two types of project constraints, technical and non technical.

Technical constraints are limitations related to the hardware, software or specific project requirements.

Non technical constraints are limitations related to time or deadlines, budget, human resources or regulatory specifications.

4.1 Technical Constraints

- Raspberry Pi
- C++ programming language
- Buildroot
- Linux OS
- Implement a device driver

4.2 Non Technical Constraints

- Two member team
- Limited budget
- Project deadline at the end of the semester

5 | Market

5.1 Market Research

A smart aquarium monitoring system is an automatic system that takes care of the pet fish in the aquarium. It is a tedious process and should be properly set up and maintained in a healthy manner or else the fish will be destined for an unpleasant and short period of life. Therefore, it is important to monitor aquarium conditions and improve the water quality of the mini aquarium tanks. Fish require the utmost care, they need some specific conditions like the temperature level and pH. The aquarium in which they are kept should have proper monitoring parameters to keep them in good health condition. [4]

The global reef aquarium market size to be valued at USD 11 billion by 2028 and is expected to grow at a compound annual growth rate (CAGR) of 10.7% during the forecast period. Growing millennials' interest in colorful ornamental fish for aquariums as a part of a luxury lifestyle is expected to remain a key factor for the market growth. Moreover, fish tanks occupy a lot less space, which also has been driving the demand. This trend is creating several growth opportunities for the market in the residential sector. [5]

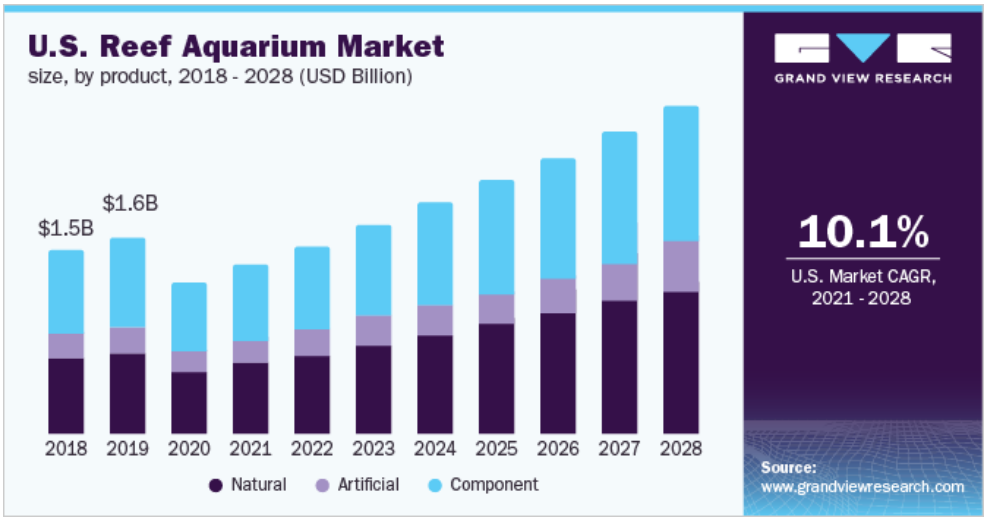

Figure 5.1: US Reef Aquarium Market.

Modern technology has made it easier than ever to cultivate and maintain a thriving aquarium at home. For instance, monitoring water chemistry in aquariums has been a constant concern for many years, but new in-tank devices can now deliver real-time measurements for everything, from pH levels to water hardness. [5]

5.2 End use insights

Household was the largest end-use segment in 2020 with a revenue share of 43%. Household use of reef aquariums is on the rise as these improve the aesthetics of a residential environment. Economic growth-led household income rise is also a key factor defining the nature of aquariums procured by the populace of a particular region. [5]

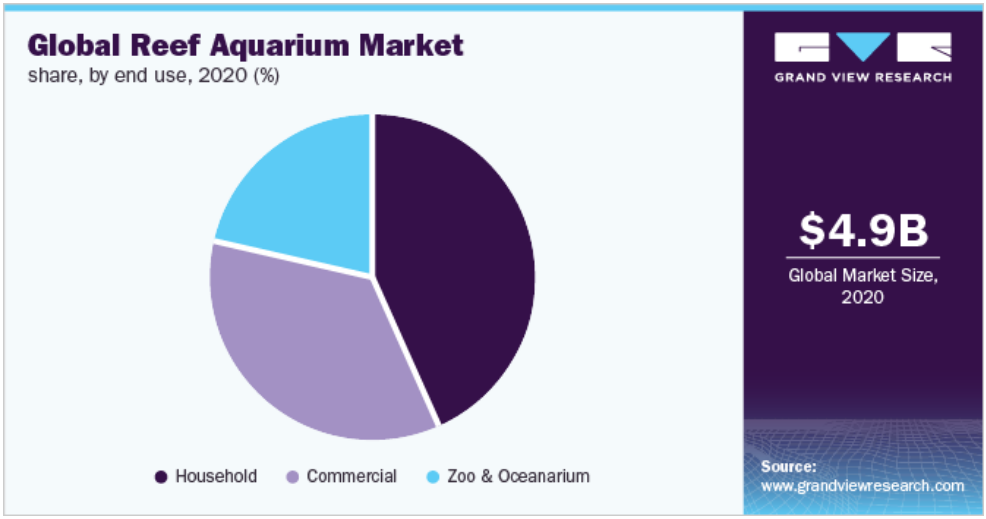

Figure 5.2: Global Reef Aquarium Market.

Zoo and oceanarium is anticipated to be the fastest-growing end-user segment at a CAGR of 11.5% from 2021 to 2028. Aquariums used in zoos and oceanariums consist of saltwater aquariums and reef tanks. These aquariums are used for displaying marine animals and plants, specifically oceanic or pelagic fish and mammals. Moreover, public interest in visiting oceanariums and aquatic zoos as a leisure activity has propelled the development of major oceanariums across the world, thereby supporting market growth. [5]

5.3 Similar Products

Even though the home aquarium market is growing, there are not many solutions on the market, much less economical and versatile solutions for everyone's needs. Below are three examples of water quality monitoring systems (Figure 5.3)(Figure 5.4)(Figure 5.5), varying in price and target audience .

Figure 5.3: Yewhick WIFI Aquarium Monitor.

- Sensors: pH, salinity and temperature
- Remote Monitoring
- Alarm Function
- Can be connected to a CO2 pump
- Two Drive Ports
- Price: 95.99 € [3]

Figure 5.4: Seneye HOME V6.

Figure 5.5: CoralVue Hydros Control 4.

- Easy to install
- Small size and simple
- Sensors: temperature, ammonia, pH and total light
- Seneye Application
- Wired or wireless network
- Requires Windows XP or above
- Price: 144,99 €
  [2]

- Cloud based app
- Sensors: temperature, pH, water level, TDS and more hydros sensors
- Controls: control pumps, timers, level sensors and more hydros sensors
- Two Drive Ports
- Four Sense Ports
- Two Probe Ports
- 0-10V I/O Control
- Two Command Bus
- Price: 449.99 €
  [1]

5.4 Target Market

The target market for this aquarium monitoring and control system is primarily directed towards households rather than businesses. The main objective is to develop an affordable and easily deployable system that covers a wide range of essential functionalities required for an aquarium.

By focusing on the residential market, this system aims to provide aquarium enthusiasts with the tools they need to effortlessly monitor and manage their aquatic ecosystems, ensuring the well-being of their aquatic pets. It offers an accessible and cost-effective solution for individuals looking to enhance their aquarium experience.

6 | System Architecture

6.1 Hardware Architecture

The hardware layer can be divided into four separate groups: GUI, Sensors, Actuators and Local System (Figure 6.1).

The Raspberry Pi 4 is the only component of the Local System and it is the "brain" of the project, responsible for the system processing. The GUI is a touchscreen display that will be in charge of the interaction with the user. The Sensors' role is to measure a variety of physical variables and transmit the data acquired to a digital format so that it can be processed. The actuators are responsible of operating the system in physical terms in order to correct an irregularity in the physical system.

Furthermore, this system needs a power source since it isn't attached to an auxiliar system and an RTC module to save the temporal values even if the system turns off.

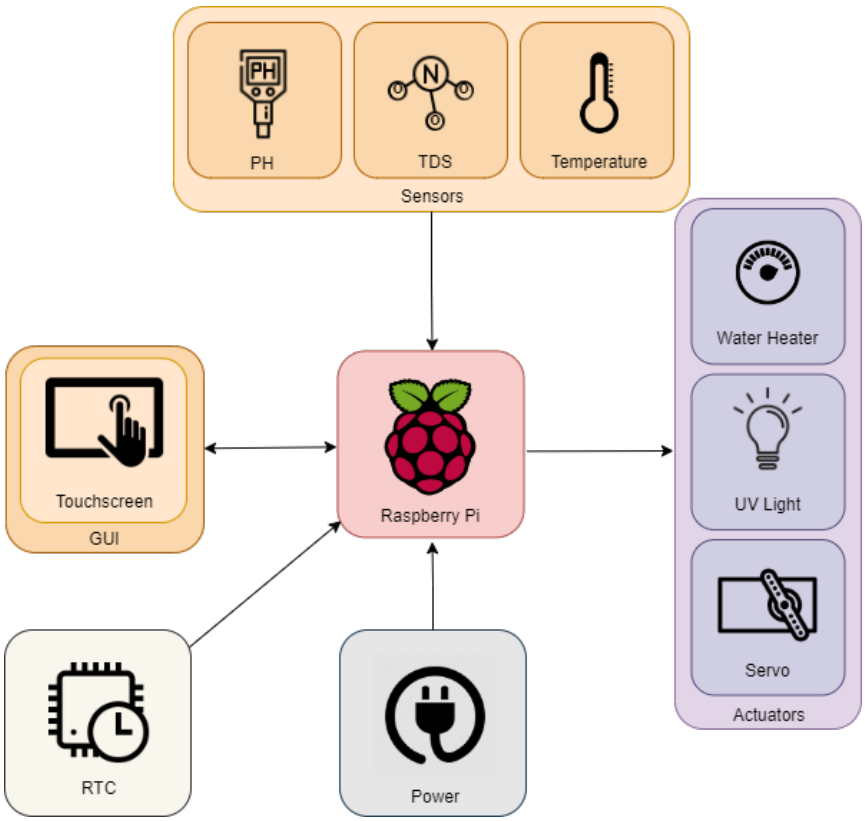

Figure 6.1: Hardware Architecture.

6.2 Software Architecture

In the software layer, the system can be divided into three distinct groups: Operating System, Middleware and User Application (Figure 6.2).

The Operating System, in this case Embedded Linux, is where all the drivers will be integrated. The middleware is represented by tools that facilitate the communication between the application and the operating system. The application is responsible for presenting an interface to the user and allowing him to communicate with the system itself.

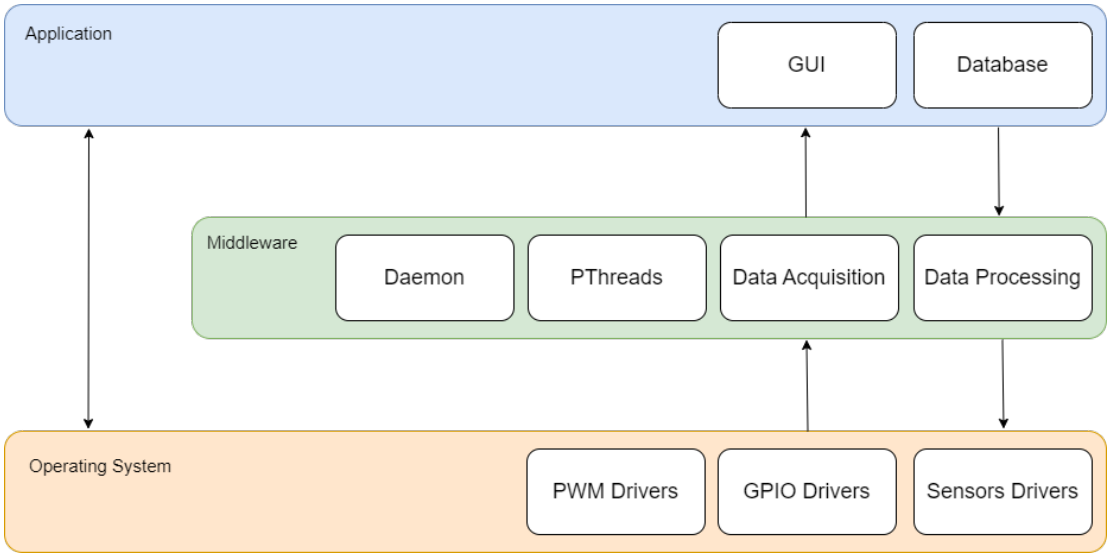

Figure 6.2: Software Architecture.

7 | System Analysis

In order to better comprehend the system functioning, there is a need to break down the major systems into smaller pieces that are easier to understand, study and develop. Designing charts and diagrams are very helpful to understand the system functionalities and goals.

7.1 State Chart Diagram

The state chart diagram makes it easier to understand project states and specify transitions and conditions for them. As can be seen below (Figure 7.1), the system starts with the configuration and then waits in an idle state for user input where it processes the desired action or waits for a defined time to give food or read sensor values.

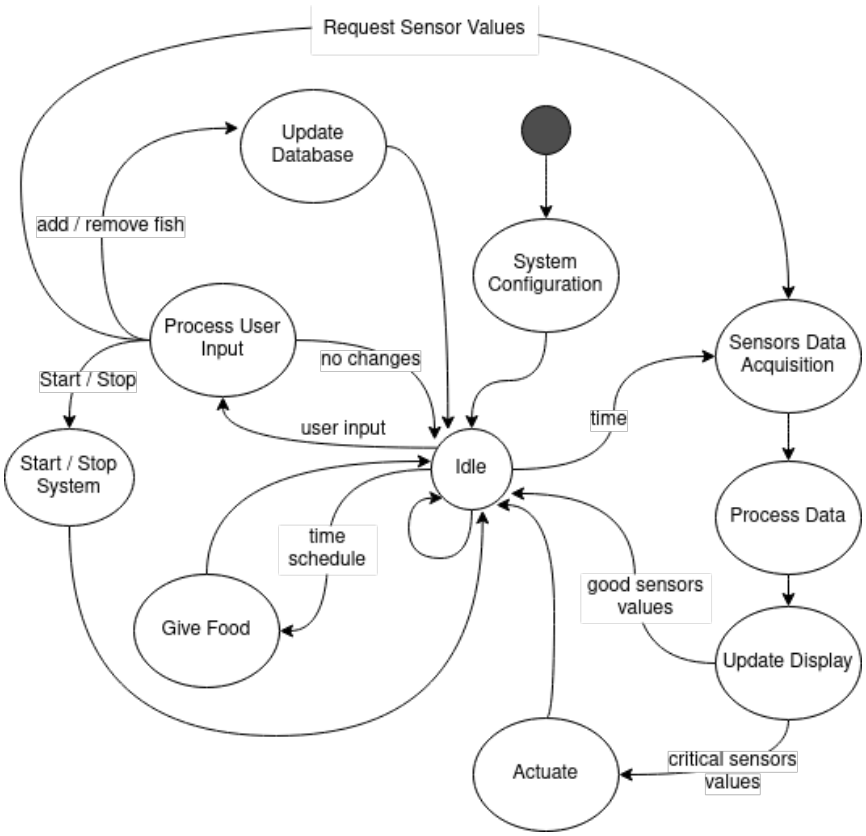

Figure 7.1: State Chart Diagram.

7.2 Table of Events

Analyzing the system's events is crucial to understand the flow of the system. It's also important to identify the source or trigger that originated the event, the system response and the type of event, which can be synchronous if its occurrence is periodical or asynchronous if it's irregular. The system events can be consulted below (Figure 7.2).

Event	System Response	Source/Trigger	Type
Touchscreen Input	Interprets Commands	User	Asynchronous
Touchscreen Refresh	Update the display	System	A/Synchronous
PH Data Acquisition	Acquire PH data from sensor	System	A/Synchronous
NO3 Data Acquisition	Acquire NO3 data from sensor	System	A/Synchronous
Temperature Data Acquisition	Acquire Temperature data from sensor	System	A/Synchronous
Turn on the UV Light	Activate the Light	System	Synchronous
Supplies food	Provides fish with food	System	Synchronous
Regulate Temperature	Heat the water	System	Asynchronous

Figure 7.2: Table of Events.

The synchronous events are simple since all of them happen periodically to maintain the ideal conditions of the aquarium, acquiring sensor data with a specific time interval between each acquisition.

The asynchronous events are mainly conditional events. The touchscreen refresh can be both synchronous or asynchronous because the screen is periodically updated with the latest data acquired from the sensors, but the user may want to acquire immediate sensor measures, updating the screen right away. For the same reason, all the sensor acquisition events can also be synchronous or asynchronous.

7.3 Use Case Diagram

In a use case diagram (Figure 7.3), it is possible to analyze the main system response when interacting with the external elements like the User and the Timer.

The user interacts with the system by adding or removing a fish, requesting an immediate sensors' acquisition and by starting or stopping the whole system. Adding or removing a fish implies some system processing and then the system is able to update the database correctly.

The timer periodically triggers a signal that requests the reading of the sensors and, according to the system time, it can also perform action on the actuators. The data acquired is then processed and compared by the system, which can lead to the activation of the actuators if something in the aquarium is out of order.

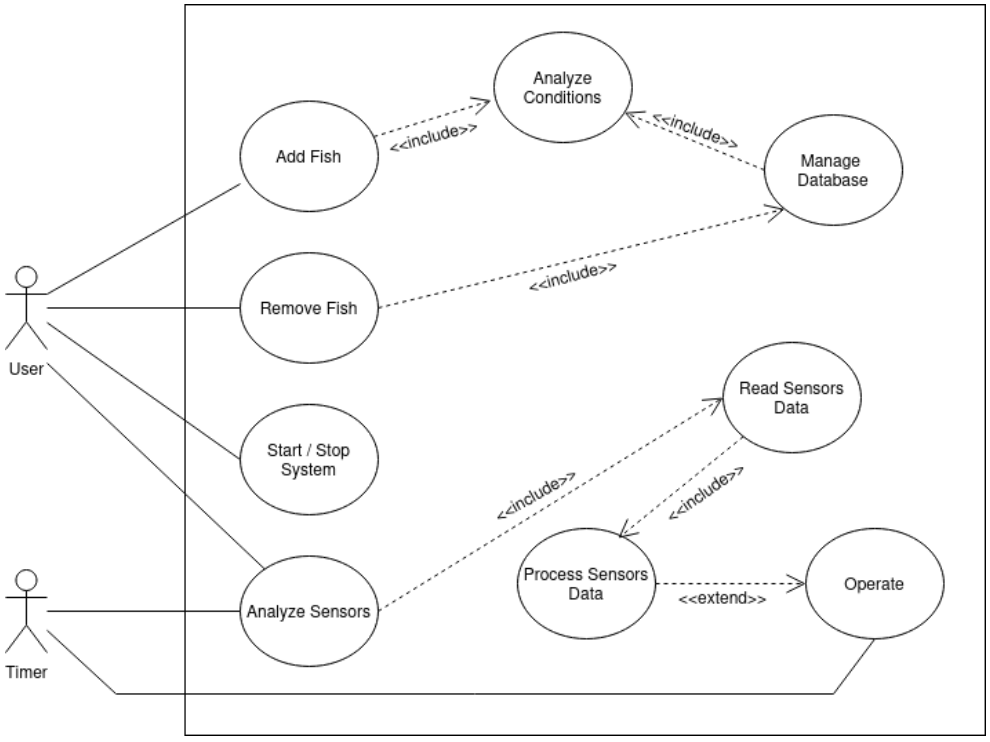

Figure 7.3: Use Cases Diagram.

7.4 Sequential Diagram

Having understood the major functionalities of the system and its possible events, the sequential diagram can be very helpful in organizing a specific scenario sequentially for a better comprehension.

The first scenario represents the two main interactions between the user and the system: adding and removing a fish from the aquarium (Figure 7.4). After pressing the "ADD" button, the system shows the fish species available in the database so the user can select the desired fish species and quantity and the system updates the database with the new fish in the aquarium. In the end, the system calculates the ideal conditions so the aquarium remains a good habitat for all the fish. After pressing the "REMOVE" button, the system shows the current fish in the aquarium so the user can select one to remove; the database is updated with less one fish in the aquarium and the system calculates the ideal conditions.

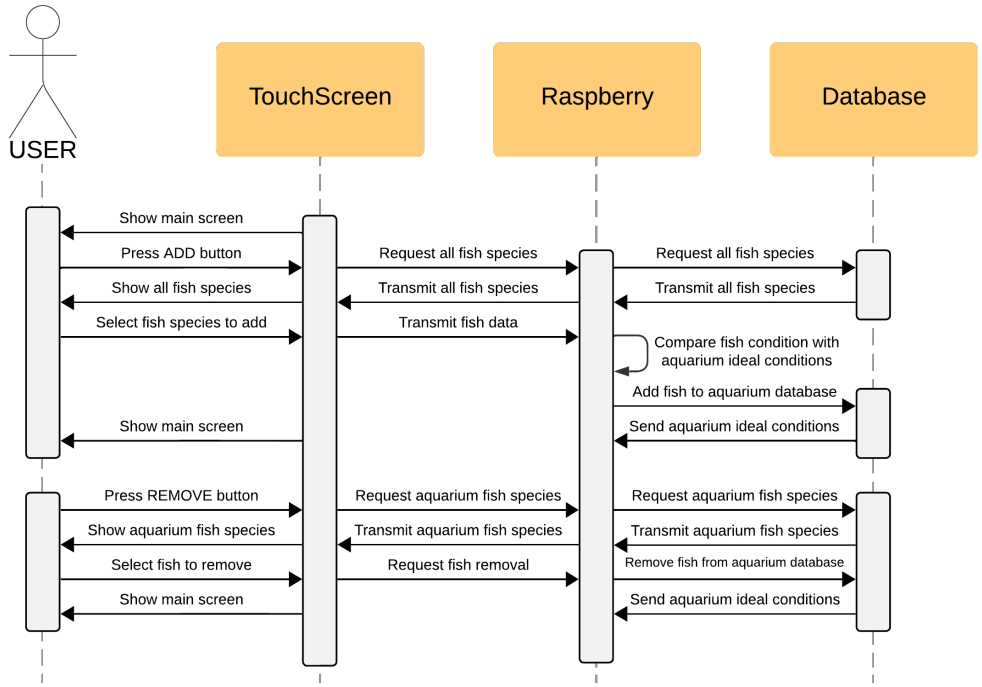

Figure 7.4: Adding and Removing a Fish Sequential Diagram.

The second scenario is a periodical data acquisition from the sensors (Figure 7.5). Once the Raspberry request a measuring, which happens within a specific interval of time, the sensors acquire the data, transmit it to the system so it can be compared with the reference values and saved in the database for further verification.

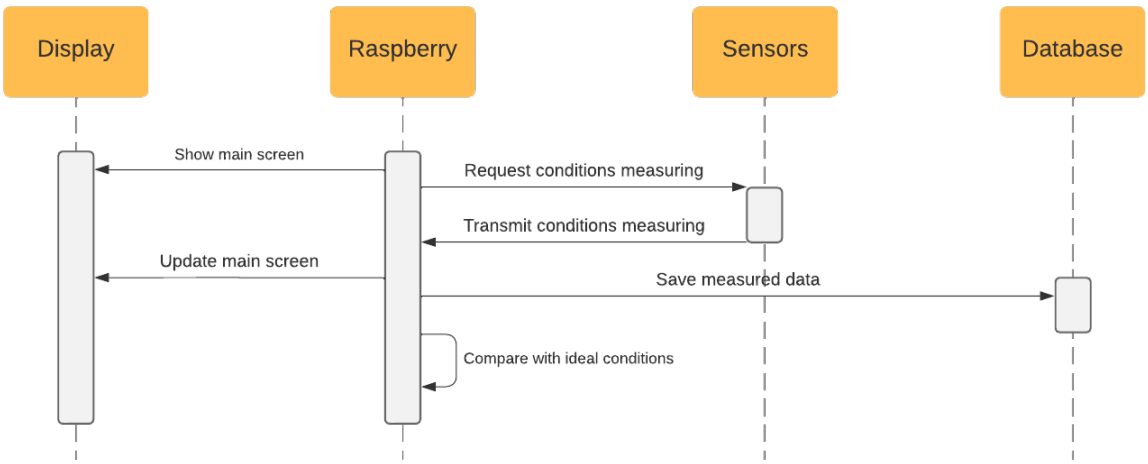

Figure 7.5: Periodical Data Acquisition Sequential Diagram.

The third scenario is a periodical data acquisition from the sensors, where the temperature is not within the tolerated limits calculated by the system taking into account all the fish living in the aquarium (Figure 7.6). Following the previous process, after comparing the acquired data with the reference, the actuators are activated and continuous data acquisition starts until the ideal conditions are met and the actuators might be deactivated.

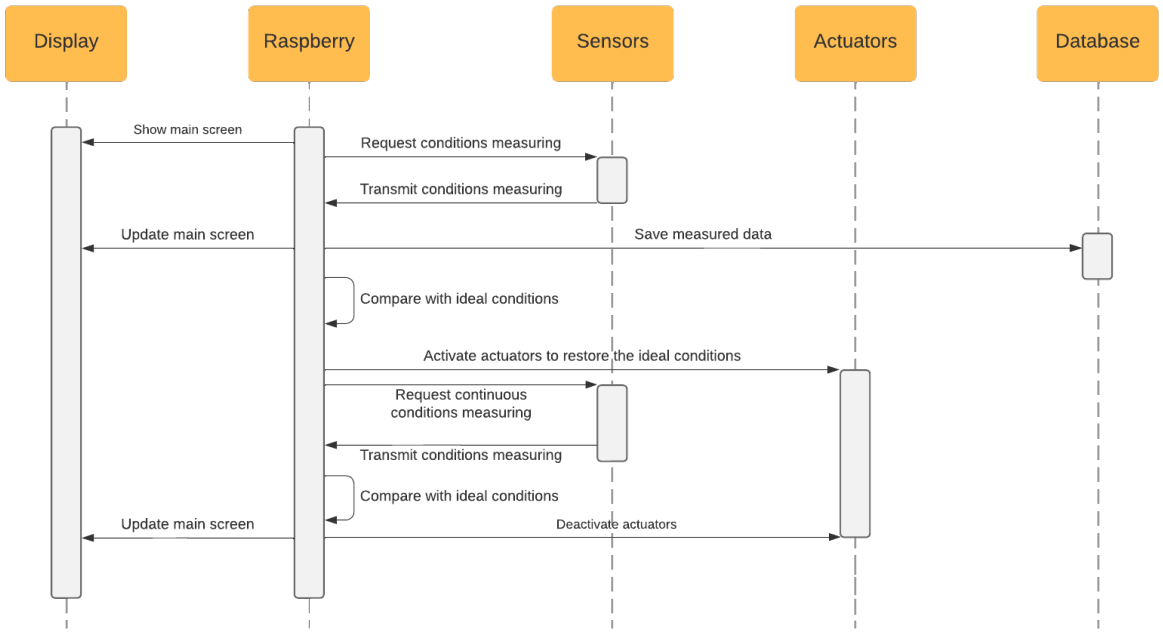

Figure 7.6: Data Acquisition with Temperature Problem Sequential Diagram.

The last scenario is very similar to the previous one, but in this scenario it's the pH or the NO3 that are not within the limits established (Figure 7.7). The process is similar, but instead of activating an actuator, a warning message is shown on the touchscreen so the user can fix it.

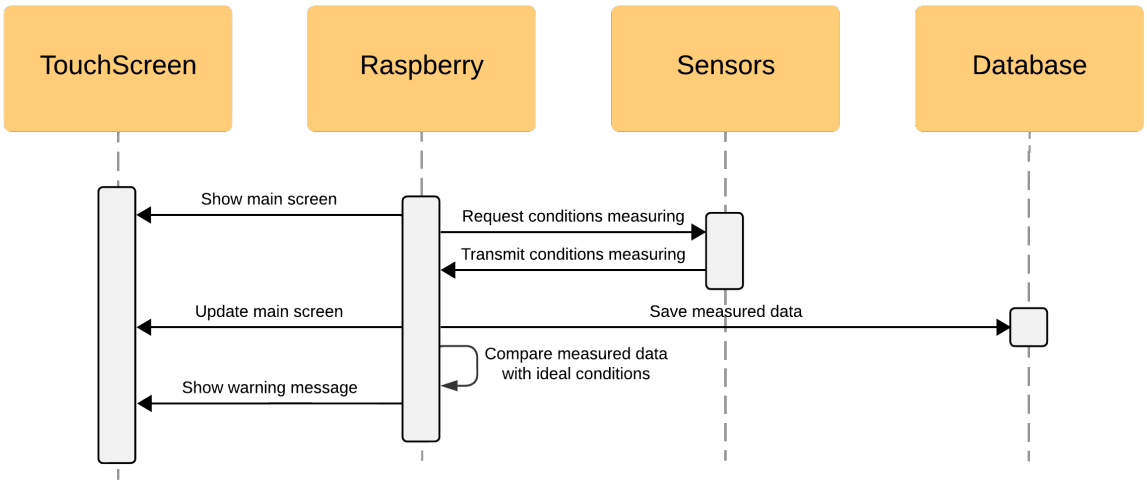

Figure 7.7: Data Acquisition with pH or NO3 Problem Sequential Diagram.

8 | Theoretical Foundations

8.1 Waterfall Model

The Waterfall Model(Fig.8.1), also referred to as the Linear Sequential Life Cycle Model, is a sequential approach to the SDLC for software development. It characterizes the software development process in a linear sequential flow, resembling the water flow over the edge of a mountain, being the "mountain", the previously defined key steps/phases of the development process. This implies that each phase begins only when the previous one is complete and no phase can be revisited after its completion.

Generally, the sequential phases in a waterfall model for the development of a software engineering product are:

- **Requirements Gathering** - Acquisition and documentation of potential requirements, deadlines and guidelines
- **Analysis** - Analysis of system specifications, technical and financial resources and definition of the overall system architecture
- **System Design** - Hardware, software, architecture and service specification
- **Implementation** - Source code development using the prior phases specifications
- **Integration and Testing** - Integration of the previously developed units into a whole system; verification of any errors and faults in the system
- **Deployment of system** - The product is deemed fully functional
- **Maintenance** - Continuous maintenance in order to improve and update the final product

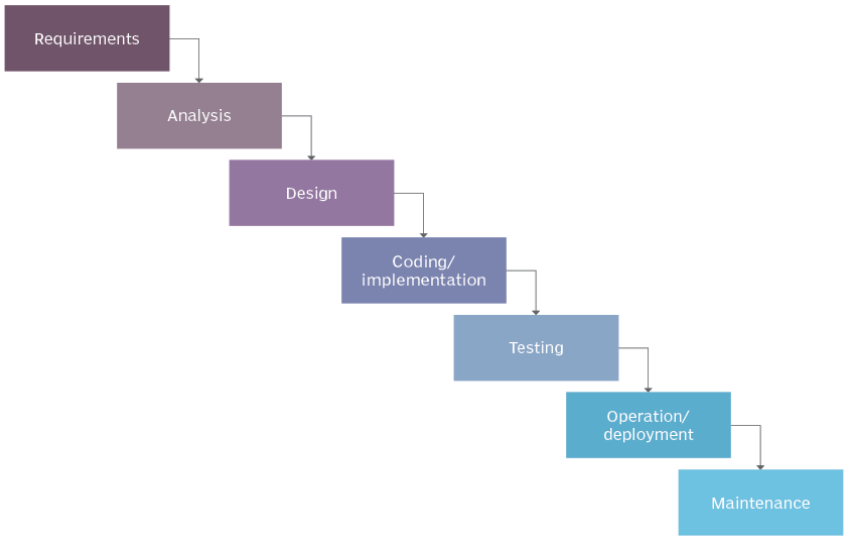

Figure 8.1: General waterfall model representation.[10]

Nowadays this methodology is often used since it allows large or changing teams to work toward the same goal without overlapping each other, simplifies the understanding of the process, clearly defines deadlines and possesses a well structured organization. Despite its advantages, the waterfall model is not perfect because it does not consider any error correction (when a flaw is detected the whole process must start over) and delays the testing and prototyping until the end of the implementation.

8.2 Process

A Process is an instance of an executing program. A single program can be employed to generate numerous processes, or conversely, multiple processes can be concurrently executed simultaneously in the same program. In other words, a process is an abstract entity established by the kernel, to which system resources are assigned for the purpose of running a program. [11]

The memory allocated to a process is divided into different sections:

- **text** section contains the machine language instructions (read-only for safety reasons)
- **initialized data** section stores the global and static variables that are clearly initialized
- **uninitialized data** stores the global and static variables that are not clearly initialized
- **heap** section is used for dynamic memory allocation
- **stack** section stores the function's local variables, function parameters, arguments and return values

A process can assume three possible states: running, ready and blocked.

8.3 Threads

Threads provide a mechanism for applications to execute multiple tasks simultaneously. Unlike processes, threads coexist within a single process and share the same global memory. This shared memory enables threads to collaborate efficiently, as they all have access to the same data.

Concurrently, the threads within a process can execute (Fig.8.2), and in a multiprocessor system, they may run in parallel. This parallel execution is especially beneficial when one thread is blocked on I/O operations, as it allows other threads to continue their tasks, ensuring better CPU utilization.

Sharing information between threads is made straightforward and quick, as they can directly access shared variables and data. Nevertheless, to prevent potential issues arising from concurrent data modification by multiple threads, synchronization techniques come into play.

Additionally, creating threads is notably faster than creating separate processes. When creating processes, it involves duplicating various attributes such as page tables and file descriptor tables, which can be a time-consuming process. In contrast, thread creation is faster because many attributes that would be duplicated when creating individual processes are shared among threads.

POSIX Threads is a library that simplify the use of threads. Using this library, threads can be created using `pthread_create()`, and each thread can terminate independently using `pthread_exit()`. It's important to note that when any thread calls exit(), all threads within the process terminate immediately. Unless explicitly marked as detached, a thread must be joined by another thread.

The ability of threads to efficiently share information and resources, coupled with their performance advantages, often positions them as the preferred choice for applications that demand concurrent execution and data sharing.

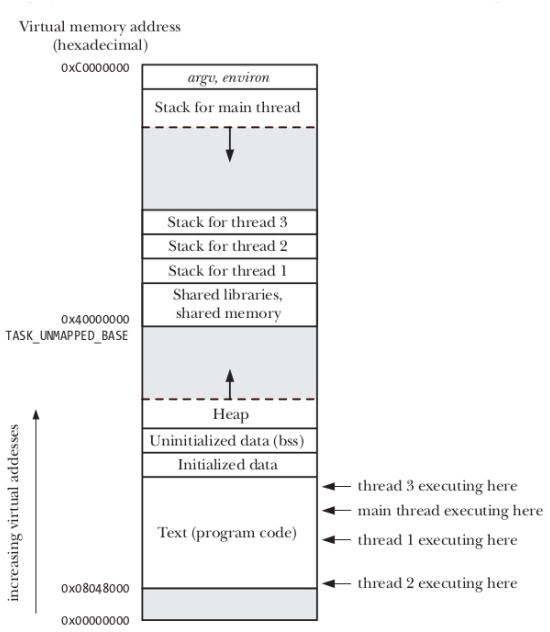

Figure 8.2: Threads executing in a process.

8.4 Daemon

A Daemon is a special program that runs continuously as a background process, separated from direct control of the user and the controlling terminal. A daemon process typically initiates during the system boot and operates until the system is powered down.

In order to create a daemon, some major steps must be executed: execute as a background task by forking and exiting; detach from the invoking session and create a new session to become its leader and detach from the controlling terminal; change the "umask" command to 0 so the operating system calls can provide their own permissions over newly created files; close the file descriptors 0, 1 and 2 (standard streams); change the working directory to the root directory (/) so the process does not keep any directory that may be on a mounted file system.

8.5 Mutex

A Mutex, derived from "MUTual EXclusion", is a program object responsible for preventing multiple threads from accessing the same shared resource, which makes it one of the most important synchronization mechanisms. A shared resource is a code element in a critical section, the part of the code that manages the code element. Critical section code segments should always be executed atomically. In a multithreaded environment, the mutex ensures that multiple concurrent threads do not try to execute a critical section at the same time; without a mutex application, race conditions might happen, leading to data incorrectly read or written or program crashing.

A mutex exhibits two potential conditions: locked or unlocked. When a thread locks a mutex, it assumes ownership of that mutex and only the owner can release it.

The usage of a mutex may lead to a deadlock, i.e. a situation where the code is blocked in a way that makes it impossible for the flow of the execution to proceed. To avoid deadlocks, it must be defined a mutex Hierarchy. Logically assigning hierarchies to each mutex makes sure that whenever threads lock multiple mutexes, they do so in the same order, that way, avoiding deadlocks.

8.6 Condition Variables

Just like the mutex, Condition Variables are also synchronization mechanisms that block a thread until another thread both executes an operation with a shared resource and notifies the blocked thread that it can continue its execution. Condition variables are generally associated with a mutex, so race conditions may be avoided.

The condition variables main operations are `wait()` and `signal()`. The `wait()` call is executed when a thread wishes to stay in a suspended state until a specific condition is met; the `signal()` call is executed when a specific condition is met and the thread wishes to wake a suspended thread that is waiting for that condition. The mutex is locked at the moment `wait()` is called, but the `wait()` call itself will release the mutex during the suspended state and lock it again once the thread awakens.

8.7 Message Queues

In order for processes to be able to communicate with each other, an IPC method is required, in this case, the Message Queue.

A message queue is a form of asynchronous communication that allows transmission of arbitrary data structures between tasks. Normally, it is similar to a FIFO buffer, where data is written to the end and removed from the front of the queue, although it is possible to do it the other way around or overwrite existent data.

The main functions in the POSIX message queue API are the following:

- The **`mq_open()`** function creates a new message queue or opens an existing queue, returning a message queue descriptor for use in later calls
- The **`mq_send()`** function writes a message to a queue
- The **`mq_receive()`** function reads a message from a queue
- The **`mq_close()`** function closes a message queue that the process previously opened
- The **`mq_unlink()`** function removes a message queue name and marks the queue for deletion when all processes have closed it.[11]

8.8 Signals

Signals in the context of the Linux operating system are asynchronous notifications that enable process communication and management, providing a way to control the behavior of processes in response to specific events.

Signals serve a variety of purposes, including notifying a process about events such as the completion of a child process or the pressing of interrupt keys (example: CTRL + C triggers the SIGINT signal). Each signal is associated with a unique number and can result in different actions when received by a process or a thread. Processes and threads can be designed to respond in a specific manner to signals, performing appropriate actions based on the received signal.

Signals play a critical role as a means of communication between processes or within a single process and are an integral component of the operating system.

8.9 Device Drivers

Device drivers are a fundamental component of computer systems, bridging the gap between hardware devices and the operating system. These software modules play a fundamental role in enabling efficient communication and control between software applications and hardware peripherals (Fig.8.3).

Device drivers can be categorized further into character device drivers and block device drivers. Character devices communicate directly with the user space program, so no buffer is required, often used for devices like keyboards.

Block devices, on the other hand, are accessed by the user space program by a system buffer that works like a data cache. There's no need for management or allocation routines as the system transfers the data from/to the device and are commonly used for devices such as hard drives and SSDs.

Device drivers are often identified by major and minor numbers. The major number specifies the driver or device type, while the minor number differentiates between individual devices of the same type, allowing the operating system to route requests to the appropriate driver.

In Linux, device drivers are often implemented as kernel modules. Kernel modules are loadable pieces of code that can be dynamically added to or removed from the running kernel. The insmod command is used to load a module, and rmmod is used to remove it. This dynamic loading and unloading of modules enable flexibility and efficient resource utilization in the system

Device drivers provide an interface between software applications and hardware devices. They make it possible for users to interact with and utilize a wide range of hardware peripherals while maintaining system stability and security.

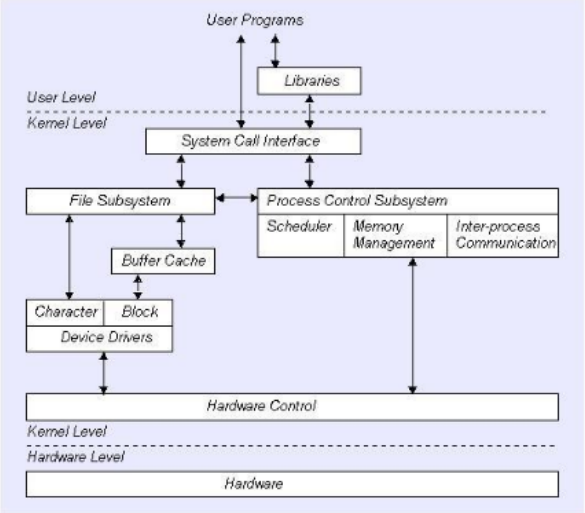

Figure 8.3: Communication between hardware, Kernel and user level.

8.10 I2C

I2C is a protocol that serves as a communication standard for connecting and facilitating data exchange between various electronic devices within a circuit or across interconnected boards (Fig.8.4).

I2C enables the connection and communication of electronic devices, including sensors, peripherals, and microcontrollers, each of which is assigned a unique address, making it straightforward to identify and select specific devices on the bus.

Each bus consists of two signals: SDA and SCL. SDA (Serial Data) is the data signal and SCL is the clock signal, and by employing only two communication lines (SDA and SCL), I2C minimizes the complexity of cabling in electronic systems, enhancing their overall efficiency.

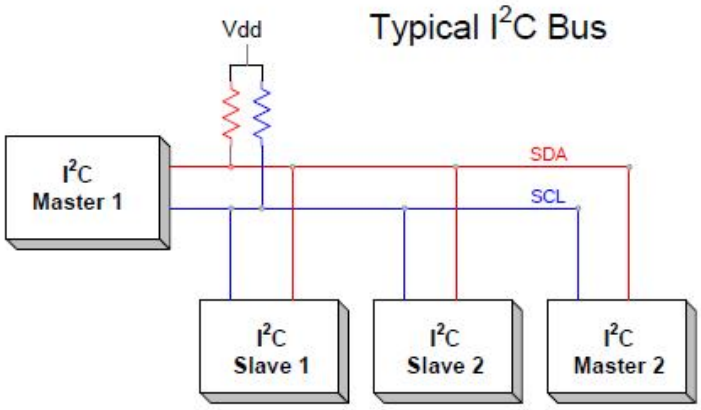

Figure 8.4: I2C Bus.

Messages in I2C are divided into two types of frames: address frames and data frames. Address frames specify the recipient peripheral, while data frames carry data messages between the controller and peripheral. Data is transferred on the SDA line after the SCL line goes low, and it's sampled after the SCL line goes high.

To initiate an address frame, the controller keeps SCL high and pulls SDA low, alerting all peripherals that a transmission is starting and in case multiple controllers attempt to gain control simultaneously, the first one to pull SDA low takes control.

The address frame always starts a new communication sequence. For a 7-bit address, the address is transmitted most significant bit first, followed by a read (1) or write (0) indication. The 9th bit is the NACK/ACK bit.

Following the address frame, data transmission begins. The controller generates clock pulses at regular intervals, and data is placed on SDA by either the controller or the peripheral, depending on the R/W bit. The number of data frames is flexible, and many peripherals auto-increment the internal register for subsequent reads or writes.

After sending all the data frames, the controller triggers a stop condition in I2C communication. Stop conditions are characterized by a transition of SDA from low to high (0->1) following a similar transition of SCL, with SCL staying high. In regular data writing operations, it's crucial to ensure that the value on SDA remains constant when SCL is high to prevent unintended false stop conditions.

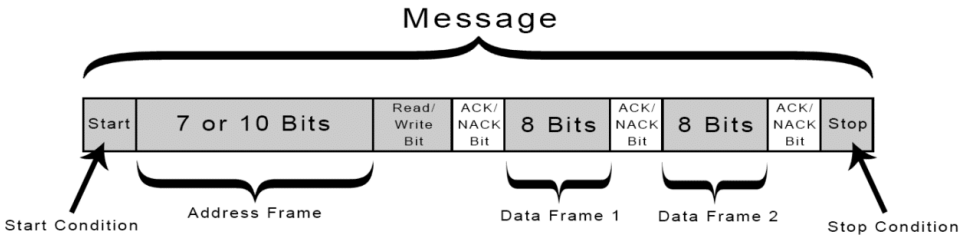

Figure 8.5: I2C Communication Protocol.

8.11 One Wire

The One-Wire protocol is a single-wire interface, half-duplex, bidirectional, low-speed and power, long-distance serial-data communication protocol (Fig.8.6).

The One-Wire protocol is a serial communication method that uses a single data line to transmit information between devices. This line is used both to transmit data and to supply power to One-Wire devices. Communication occurs through variations in line voltage, with short voltage pulses representing '1' bits and the absence of a pulse representing '0' bits. A pull-up resistor is often used to keep the line in a high state when not in use.

This protocol, unlike i2c, does not use a clock signal. Instead, the slave devices are internally clocked and synchronized with a signal from the master device. The master device is solely responsible for the read and write operations of the slave devices, so they cannot initiate a data transfer on their own.

One-Wire devices follow a set of commands and a specific protocol to perform tasks such as initiating communication, transmitting data, and receiving responses. Each device has a unique 64-bit address that identifies it on the network. The protocol supports various network topologies, including daisy-chained or star-connected devices. Additionally, One-Wire devices can operate without an external power source, leveraging line power during communication.

Due to the relatively low data transfer rate of the One-wire protocol, it may not be the ideal choice for high-speed applications, however it is cost-effective as it offers a simple hardware implementation and a low-power footprint.

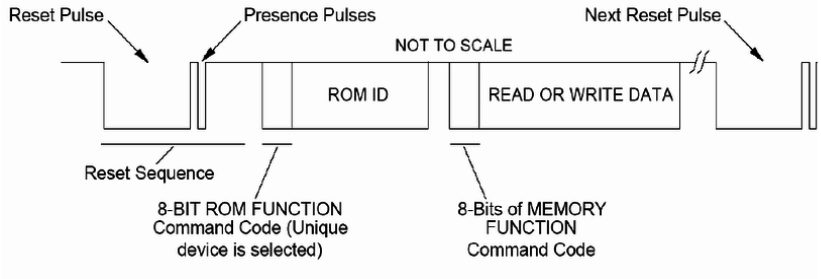

Figure 8.6: One Wire Communication Sequence.

8.12 Buildroot

Buildroot is an open-source tool for building embedded Linux systems. It automates the compilation process, enabling the creation of custom systems, including the kernel, libraries, and applications. It is widely used to generate minimal and efficient Linux system images for embedded devices. Buildroot can handle tasks such as building the cross-compilation toolchain, creating a root file system, compiling a Linux kernel image, and generating a boot loader for the target embedded system.

Buildroot is capable of generate a custom Linux image for a Raspberry Pi 4. The primary aim is to optimize efficiency while conserving resources by integrating only the essential components. This tool enables constructing an image that aligns with hardware and application requirements, ensuring a highly optimized system that utilizes minimal resources while delivering a good performance.

8.13 Database

A database is an organized collection of structured data, controlled by a DBMS. The data is typically modeled in rows and columns in a series of tables so it can be easily accessed, managed and controlled. The majority of databases use SQL for writing and querying data.

There are three main types of DBMS models: Relational (independent tables), Network (graphical representations) and Hierarchical (tree-like structure).

From DBMS can be identified the RDBMS model, which is the most used model currently, for example, MySQL or SQLite.

9 | Hardware Specification

9.1 Development Board

The Raspberry Pi is a compact computing board that can be used to perform computing tasks, control devices, collect data, and run applications in a small space, offering a cost-effective and efficient solution for incorporating computing capabilities into this project.

Figure 9.1: Raspberry PI 4B.

Specifications:

- Processor
  - Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- Memory
  - 4GB LPDDR4
- Conectivity
  - Wireless 2.4 GHz e 5.0 GHz IEEE 802.11b/g/n/ac
  - LAN, Bluetooth 5.0, BLE
  - Gigabit Ethernet
  - 2× USB 3.0 and 2x USB 2.0 ports
- GPIO
  - 40 pins
  - Output Voltage of 3.3v
  - Max Current of 16mA
- Video
  - 2× micro HDMI ports
  - 2-lane MIPI DSI display port
- Card support SD
  - Micro SD
- Power
  - 5V DC via USB-C or via GPIO header
- Working Temperature 0 a 50°C

9.2 Touchscreen Display

A touchscreen display, integrated into a Raspberry Pi, becomes a necessary tool for interaction and control between the user and the system. It allows touch, gestures and interactivity, making navigation more intuitive and providing a personalized graphical interface that is pleasant for the user.

Figure 9.2: Touchscreen Display.

Specifications:

- Size: 5 inch
- Screen: LCD
- Screen Proportion: 16:10
- Resolution: 800x480
- Touch Type: Capacitive
- Connection: DSI display port
- Touch Points: 5-Points

9.3 Sensors

9.3.1 DS18B20

The DS18B20 temperature sensor is a highly accurate and versatile digital device designed to measure temperature in varied environments. Its main feature is the ability to operate in wet and submerged conditions, making it an ideal choice for the "Aquarium Control" project.

Figure 9.3: DS18B20 Sensor.

Specifications:

- Operation voltage: 3-5.5V;
- Measuring range: -55°C to +125°C;
- Accuracy: ±0.5°C between -10°C and +85°C;
- Stainless steel tip;
- One wire interface.

9.3.2 TDS

The TDS (Total Dissolved Solids) sensor in an aquarium plays a critical role in maintaining water quality, which is vital to the health and success of fish and other aquatic life. It helps ensure that the water is kept within acceptable dissolved solids concentration limits, creating a healthy environment for the aquarium inhabitants.

Specifications:

- Input Voltage: 3.3 - 5.5V
- Output Voltage: 0 - 2.3V
- Working Current: 3 - 6mA
- TDS Measurement Range: 0 - 1000ppm

Figure 9.4: TDS Sensor.

9.3.3 PH-4502C

The PH-4502C sensor is a specific device for measuring pH in liquid solutions. It is designed to provide accurate measurements of the pH of water, helping to maintain ideal conditions for various purposes such as the health of aquatic organisms, chemical processes and water treatment . The sensor is essential, so controlling and maintaining the pH within the ideal range is crucial to prevent health problems in fish and to ensure that the aquatic ecosystem is stable and healthy.

Figure 9.5: PH-4502C Sensor.

Specifications:

- Voltage: 3.3 - 5V
- Current: 5 - 10 mA

- Range: 0 - 14 PH
- Resolution: 0.15PH
- Probe Interface: BNC
- Measure Temperature: 0 - 60
- Alkali error: 0.2PH

9.4 Actuators

9.4.1 SG90

A servo motor is a precision motion control device that maintains accurate position, speed, and acceleration through a closed-loop system, enabling real-time adjustments based on feedback.

Figure 9.6: SG90.

Specifications:

- Rotation: 0 - 180 degrees
- Stall torque: 1.8 kg/cm
- Operating speed: 0.1 s/60 degree
- Operating voltage: 5V

9.4.2 Water Heater

A water heater is a device designed to heat water in an aquarium. It is usually submerged in the aquarium water and plays a key role in controlling temperature, ensuring that the water is within the appropriate temperature range for fish and other aquatic organisms.

Figure 9.7: Water Heater.

Specifications:

- Material: glass
- Power: 50W
- Supply 220-240V

9.4.3 UV Light

A UV light is a device designed to emit ultraviolet radiation into aquarium water. It plays an essential role in maintaining water quality and controlling unwanted organisms such as algae and bacteria. Therefore, UV light is essential in maintaining a healthy environment for fish and other aquatic organisms.

Specifications:

- Material: glass
- Power: 11W
- Supply 220-240V

Figure 9.8: UV Light.

9.5 DS3231 RTC

The DS3231 RTC (Real-Time Clock) module is an electronic component that allows you to maintain precise time control in electronic systems. It is known for its high precision, having a very low margin of error, just a few seconds per year. It is also capable of keeping information such as date (day, month and year) and managing it until the year 2100. The RTC module is also low energy consumption and has an integrated battery in case of a power outage.

Figure 9.9: DS3231 RTC.

Specifications:

- Supply: 2.3V - 5.5V
- Operating temperature: -40 - +85
- Real-time clock counts seconds, minutes, hours, Day, date, month, and year
- I2C bus for address and data transfer
- CR927 battery
- Low-power consumption

9.6 ADS1115

The ADS1115 is a 16-bit analog-to-digital converter (ADC) that converts analog signals to digital signals and uses I2C communication to transmit the digitized data. In the "Aquarium Control" project, this converter will be used for analog signals from PH and TDS sensors.

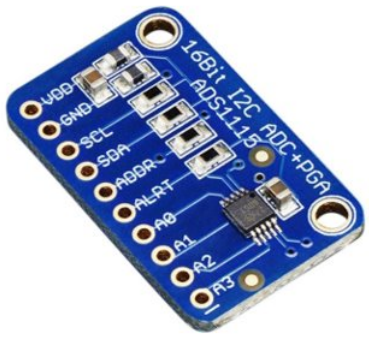

Figure 9.10: ADS1115.

Specifications:

- Supply: 2.0V to 5.5V
- Low Current Consumption
- Programmable Data Rate: 8SPS to 860SPS
- Internal Low-Drift Voltage Reference
- Internal Oscillator
- I2C Interface (Pin-Selectable Addresses)

9.7 2 Channel Relay Module

A relay is an electromechanical device that acts as a switch controlled by an electric current in a coil. Relays are often used to control a high power electrical circuit using a low power circuit. In this project, a Raspberry Pi will be used as a control circuit to operate a UV lamp and a water heater.

Specifications:

- Supply: 5V
- Trigger: high/low level trigger

Figure 9.11: 2 Channel Relay Module.

- Load Voltage: AC:250V/10A, DC:30V/10A
- Max Current: 26.8mA
- Module Life: 10 million times

9.8 MicroSD Card

The microSD card is therefore the main storage media for all essential components of the Raspberry Pi system, including the operating system, boot files and offers additional space to store your own files and user data.

Figure 9.12: MicroSD Card.

Specifications:

- Capacity: 32GB
- Performance: 100MB/s Read, UHS-I Speed Class, U1, V10
- Operating temp: -25 °C - 85 °C
- Voltage: 3.3V

9.9 Power Supply

A power supply for a Raspberry Pi is a necessary component to provide the electrical energy necessary for the sensors, actuators and for the Raspberry Pi to function correctly.

Figure 9.13: Power Supply.

Specifications:

- Input Voltage: 220-240V
- Output Voltage: 5.1V
- Currente: 3A
- Max Power: 15.3W
- Connector: USB Type-C

10 | Interface to Hardware

10.1 Hardware Connections

In order to better understand the connections of the previously specified hardware, the following figure (Fig.10.1) shows the connections of all the hardware used in the ”Aquarium Control” project.

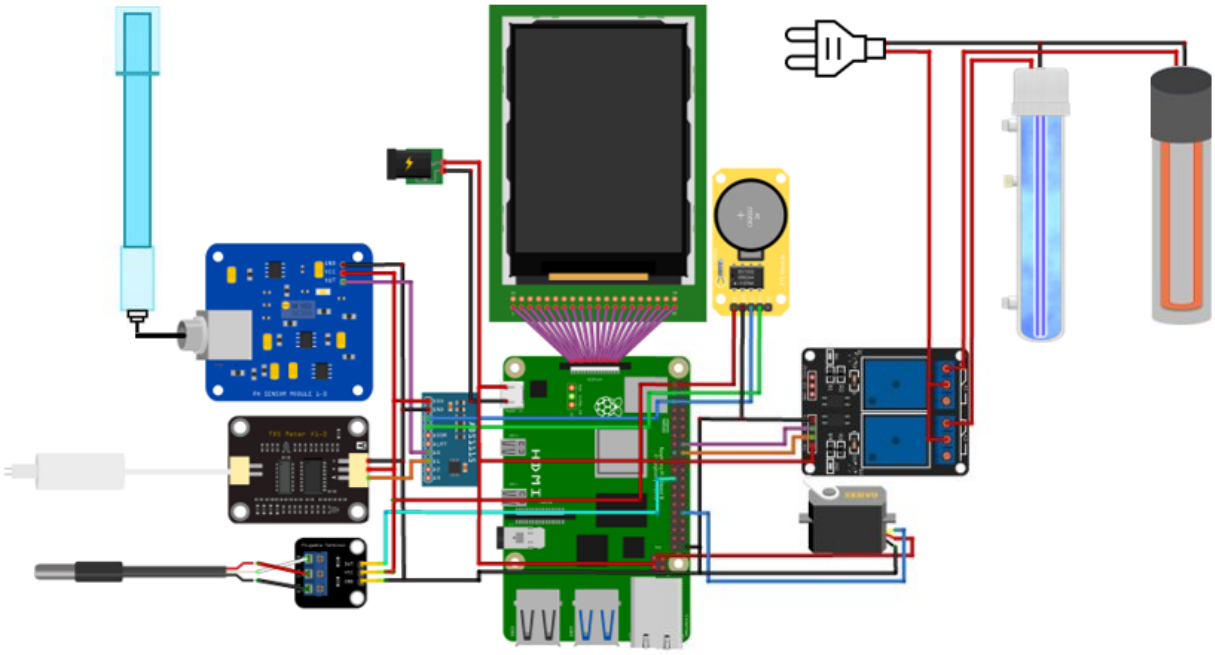

Figure 10.1: Hardware Connections.

11 | Software Specification

11.1 Buildroot

Buildroot is a fundamental tool in this project, as it allows the creation of a custom operating system image. This image is custom designed, with specific kernel configurations and software packages, meeting the unique needs of the project. Buildroot simplifies the build process, offering flexibility to customize the system, resulting in an optimized operating system that perfectly adapts to the Raspberry Pi 4 and the project context.

Figure 11.1: Buildroot Logo.

11.2 SQLite

SQLite, being a database, plays a crucial role in providing a framework for storing, organizing and retrieving data efficiently. It enables the persistence of critical information and easy access to that information, ensuring that data is consistent, reliable and available when needed.

Figure 11.2: SQLite Logo.

11.3 QT

Qt is a design tool that in this project allows the creation of an intuitive interface, adapted to the touchscreen, which allows easy and effective interaction with the Raspberry Pi, making it an ideal choice for a wide range of projects, from industrial controls to home entertainment systems.

Figure 11.3: QT Logo.

11.4 Diagrams.net

Diagrams.net is an open source diagramming tool that allows you to create high-quality flowcharts, UML, entity relationship diagrams, sequential diagrams and more in an easy and intuitive way.

Figure 11.4: Diagrams.net Logo.

11.5 Visual Studio Code

Visual Studio Code (VS Code) is a powerful and versatile source code editing platform developed by Microsoft that is available for Windows, Linux, and macOS, offering a comprehensive set of essential features for developers. Visual Studio Code is a solid choice for software development projects, providing efficiency and customization in a highly flexible code editing environment.

Figure 11.5: Visual Studio Code Logo.

11.6 PThreads

The use of Pthreads in embedded systems projects is essential for implementing multitasking and concurrency, allowing the system to perform different tasks in an efficient and optimized manner. They play a critical role in improving performance and effectively utilizing resources, making them a valuable choice for developing complex embedded systems.

Figure 11.6: PThreads.

11.7 Daemon

A daemon is a computer program designed to operate in the background, continuously and autonomously. The main function of a daemon is to perform specific tasks without the need for direct user intervention, playing a fundamental role in operating systems to allow the system to perform essential functions and services without overloading system resources.

11.8 Device Drivers

Device drivers are essential software programs that act as intermediaries between a computer's operating system and connected hardware. They play a key role in communicating and controlling devices, allowing the operating system to access and manage specific hardware in an efficient and secure way.

12 | Software Interface

12.1 Classes Diagram

To better understand how the classes should be implemented, the relationship between them must be studied and analyzed, as it is presented in the figure 12.1.

In terms of connections between classes, it should be noted that there is a MainClass made up of several classes: ServoActuator, GPIOActuator, IdealConditions, TDS, Temp and PH. The Temp, TDS and PH classes also inherit the Sensors class, which is composed of the ADS1115 class. Finally, it is also possible to observe that the Daemon uses the Database, IdealConditions and MQueueHandler classes.

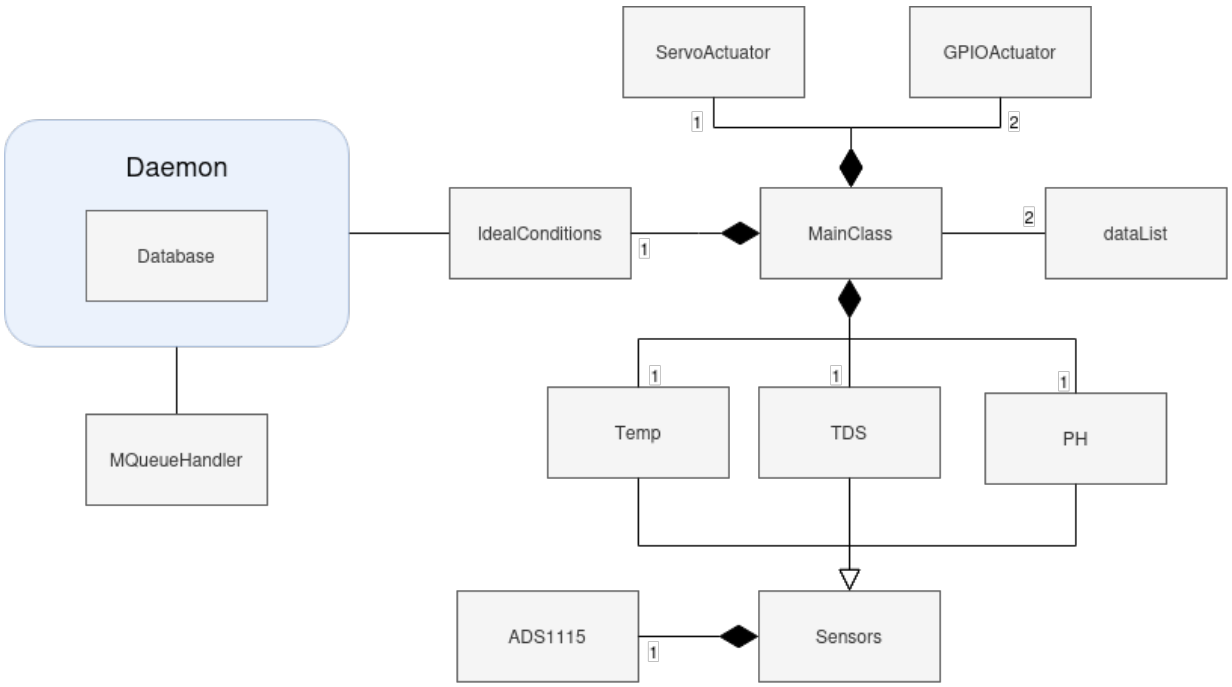

Figure 12.1: Classes Diagram.

12.1.1 MainClass class

In order to maintain obejct-oriented programming, improve the system readability and flexibility and promote the encapsulation of the primary functionality of the program, a main class called *MainClass* will be implemented where objects from other classes should be created as well as the threads functions. This class is represented in the figure 12.2.

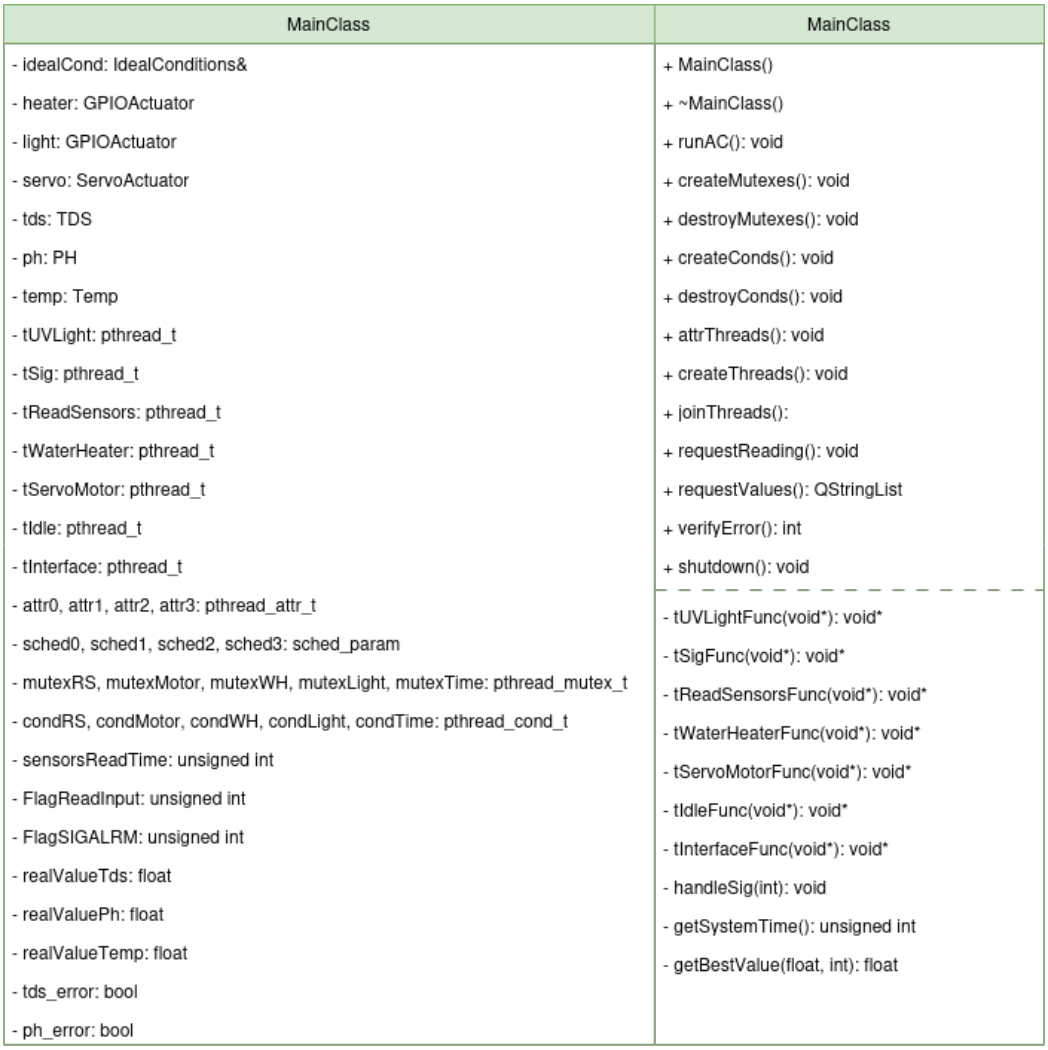

Figure 12.2: MainClass class diagram.

12.1.2 Sensors classes

To monitor the quality of the aquarium water, as illustrated in the following figure (Fig.12.3), three classes were created for each sensor that are inherited from the *Sensors* class. inheritance was used in order to reuse code, thus having a sensors class with all the functions necessary for the classes that inherit it with the only exception of the *convertValue()* function which is necessary to change in each class as the values for the conversion are different and In the Temp class, the *readSensor()* function is different because it is a One Wire bus and not an I2C bus.

In the *Sensors* class, there is an I2C read function for TDS and pH measurements, and a One Wire read function for temperature measurement. It is also composed of an *ADS1115* class responsible for defining the reading channel of the *ADS1115 module* used in the conversion of ADC signals to an I2C bus.

These classes were developed to assist with the control and abstraction required for monitoring the system.

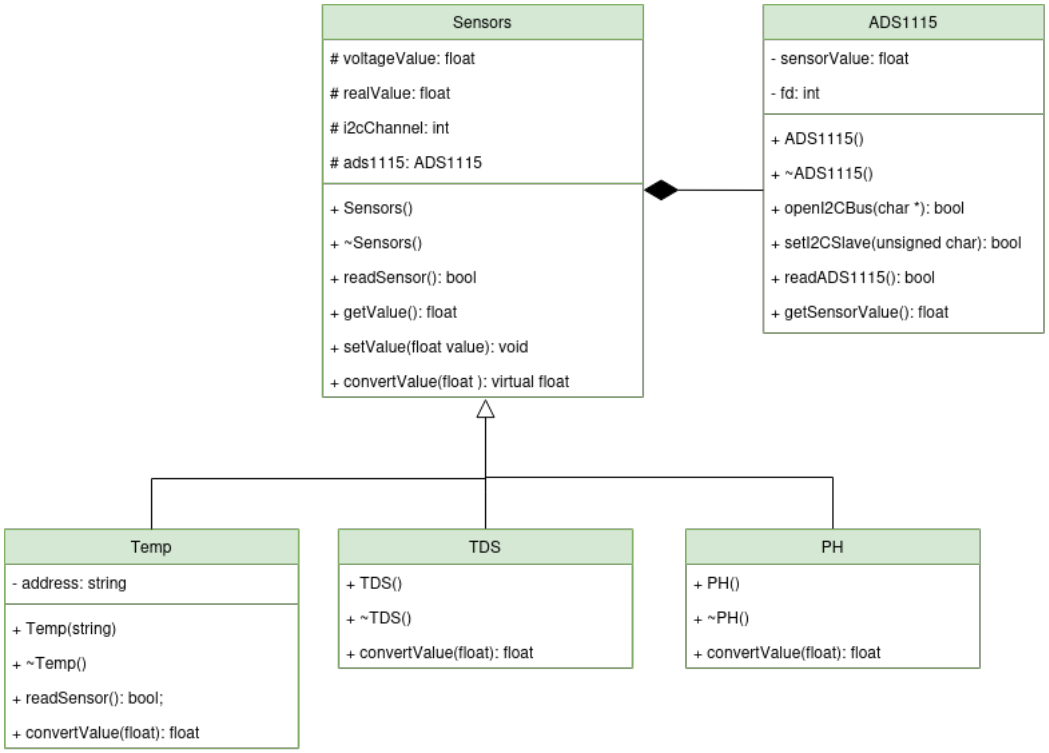

Figure 12.3: Sensors classes diagram.

12.1.3 Actuators classes

Regarding actuators, there are only two classes: *GPIOActuator* and *ServoActuator*. The *GPIOActuator* class is used to activate the UV light and the water heater and in turn provide the program with more precise control. The *ServoActuator* class is responsible for the servo motor, which in turn is used to feed the fish in the aquarium. This class has a *feed()* function used to rotate the servo motor according to the amount of food needed by the fish and a *dayTime* variable used to control its proper functioning throughout the day. These classes are represented below (Fig.12.4).

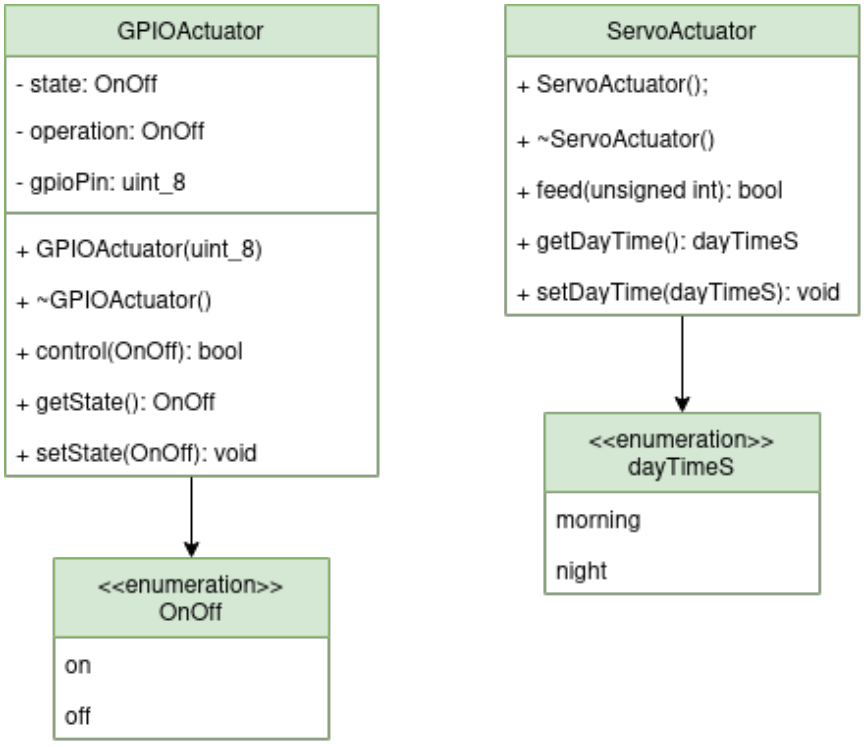

Figure 12.4: Actuators classes Diagram.

12.1.4 IdealConditions Class

In order to preserve and have quick and easy access to ideal water conditions for the fish species present in the aquarium, the *IdealConditions* class (Fig.12.5) serves the purpose by having variables to store ideal values and functions to change and obtain the ideal values at the moment.

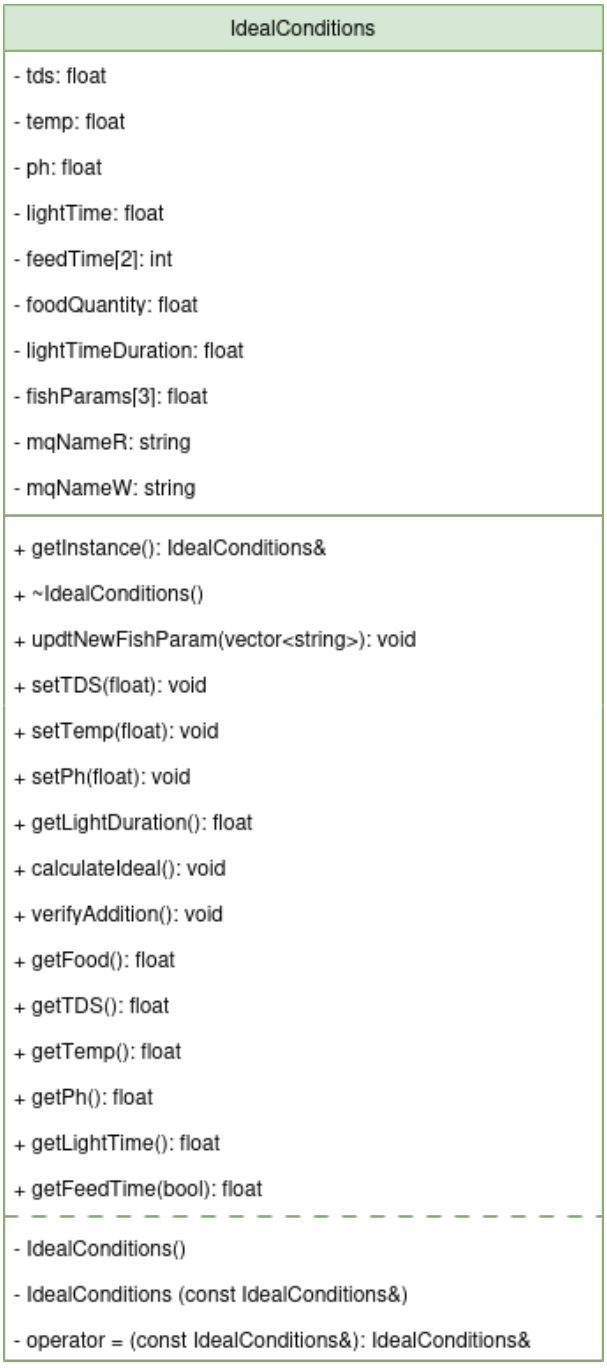

Figure 12.5: Ideal Conditions classes diagram.

12.1.5 DataList and Message Queue Handler Classes

The *dataList* and the *MQueueHandler* are two classes that will be implemented just in the interface thread in order to facilitate the data management (Fig.12.6).

To display all the species in pages and make each one easily accessable, the *dataList* class will be used, dividing all the species in chunks with a default size and providing functions to get the data of the selected one.

The *MQueueHandler* class purpose is to communicate with the daemon via message queue and convert the read values.

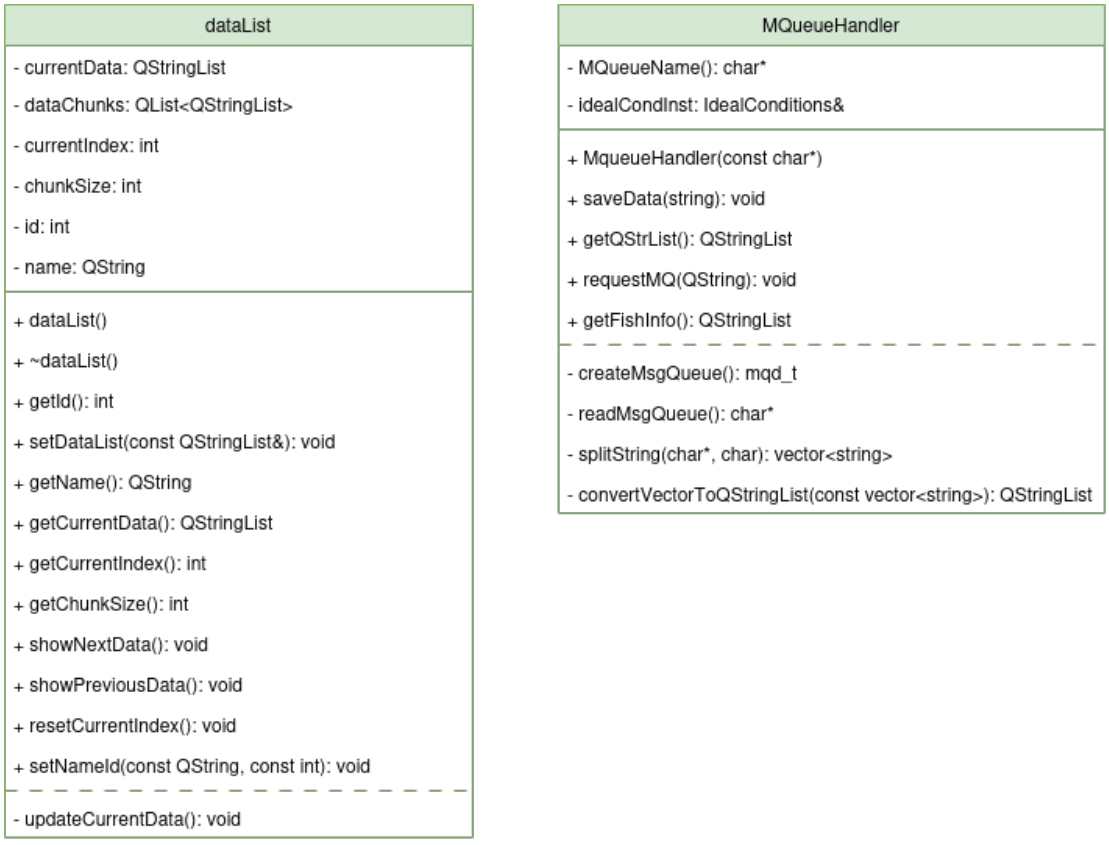

Figure 12.6: DataList and Message Queue Handler classes diagram.

12.1.6 Database Class

The aquarium control project uses a database to store most of the data. In order to communicate with the database, the system has one database class (Fig.12.7) which is a wrapper of the base functions of *sqlite3* and uses SQL statements to execute the desired query, that can only be queries to write something to the database but also to return something from it.

Figure 12.7: Database class diagram.

12.2 Database Entity-Relationship Diagram

The database implemented has three entities: *fishList*, *aquariumFish* and *sensorsData*.

The *fishList* table contains an huge variety of fish species and the respective information about their ideal environmental conditions. The *aquariumFish* table contains all the fish species currently habitating the aquarium and the respective quantities. The *sensorsData* table contains the last 3 days sensors readings of the respective element and the corresponding time it was read, in order to analyze the data in a graphical way.

To better clarify the relationship between the database entities, an Entity-Relationship Diagram was designed (Fig.12.8).

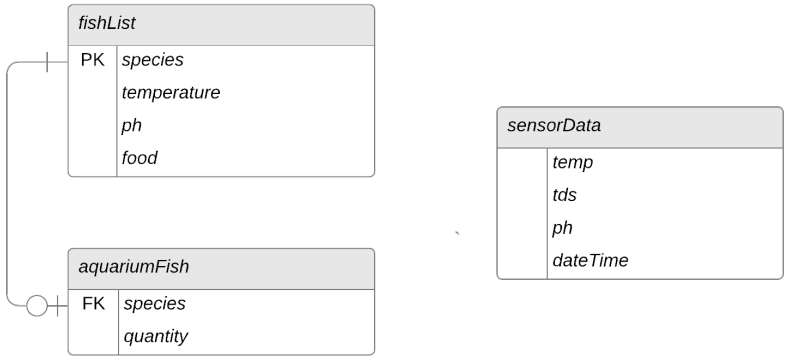

Figure 12.8: Database Entity-Relationship diagram.

12.3 Thread Synchronization

As the system runs, threads must maintain constant communication with one another to signal specific events, share data, allow user input handing to occur concurrently with other operations and more. To implement such communication without conflicts, synchronization methods are required.

The system utilizes three well-known synchronization methods: condition variables to suspend a thread execution until another thread signals it about a specific occurrence, mutex to guarantee that no more than one thread can access the same resource simultaneously and message queues to transfer data between the threads and the daemon, providing full access to it.

The following table (Table 12.1) enumerates all the condition variables and mutexes implemented in the system. The system will also use the signal SIGALRM.

Condition Variables	Mutex
condRS	mutexRS
condMotor	mutexMotor
condWH	mutexWH
condLight	mutexLight
condTime	mutexTime

Table 12.1: Condition Variables and Mutex Table.

12.4 Inter Thread/Process Communication

Since all the system components are all described, a thread interaction diagram (Fig.12.9) can be designed to better demonstrate all the tasks functioning and interaction.

The communications between the threads and the daemon are made via Message Queue, since the daemon is a second plan process that can't communicate directly with the main system.

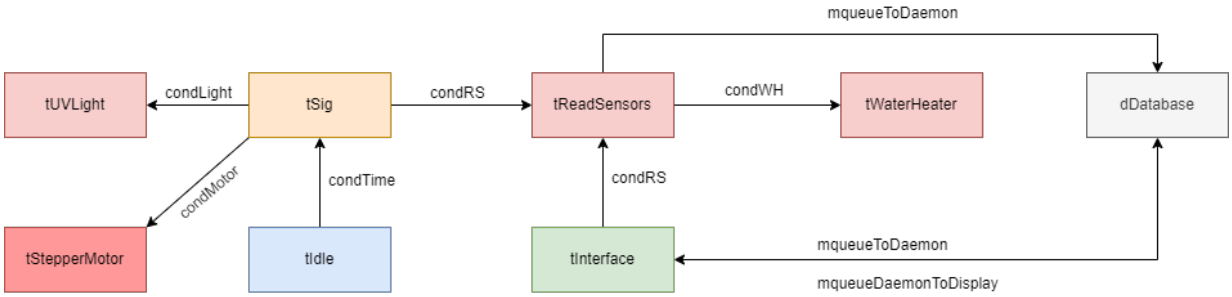

Figure 12.9: Threads and Daemon Interaction Diagram.

12.5 Threads Priorities

The subsequent figure (Fig.12.10) depicts the threads' priorities.

The higher priority threads are *tServoMotor*, *tWaterHeater* and *tUVLight* because they are responsible for feeding the fish, heat the water and activate the UV light. The *tReadSensors*, and *tSig* threads are the second highest priority, as the *tReadSensors* is responsible for reading the sensors and the *tSig* thread processes whether the UV Light or the servo motor should be activated.

In the second lowest priority is the *tInterface* thread, given that it focused on user interaction with the system. The lowest priority thread is the *tIdle* thread that is responsible for periodically waiting for a SIGALRM signal and sending a condition to trigger the execution of the *tSig* thread.

12.6 Flowcharts

To exhibit every thread in some detail, the thread's behavior will be presented as flowcharts to better observe the execution flow.

In order to simplify the representation of the condition variables which are formed by three main functions, two distinct functions (*waitCond* and *sendSignal*) were created, as is visible in the following

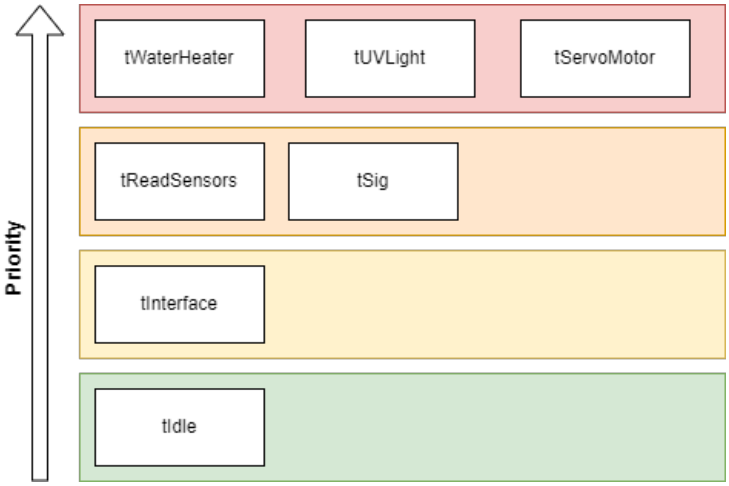

Figure 12.10: Threads Priorities.

flowchart (Fig.12.11).

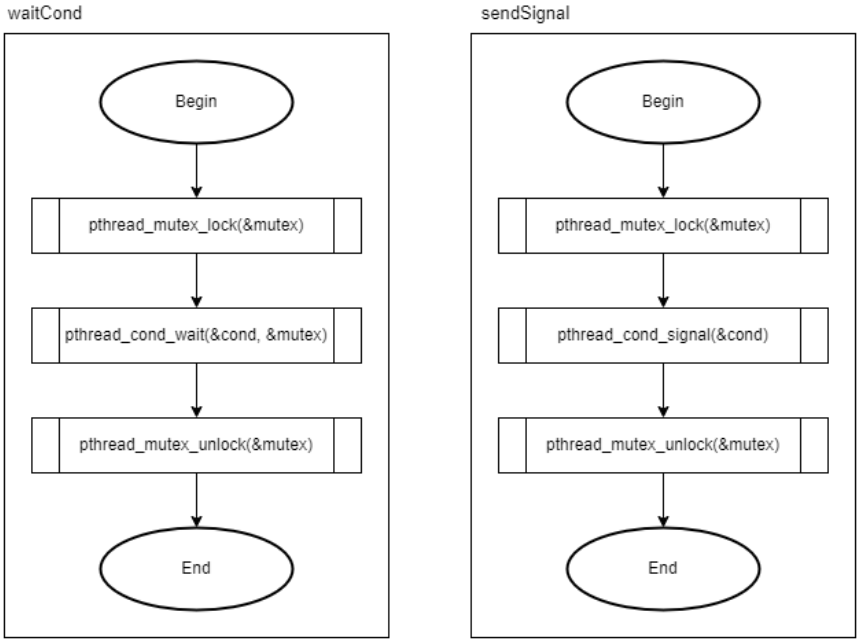

Figure 12.11: waitCond and sendSignal Flowcharts.

These functions share similar execution paths. Both start attempting to obtain access to a particular condition variable by locking the respective mutex. After securing it, the function either suspends the thread waiting for a condition variable to be set (*waitCond*) or sets the condition variable through a signal (*sendSignal*). Upon completion, both functions release the previously locked mutex and return to the calling thread.

12.6.1 tReadSensors's functions

After an initial configuration, the *tReadSensors* waits for the condition variable *condRS* to be set. Once it is set, all three sensors start measuring the pH, temperature and TDS consecutively and the measured values are added to a buffer; right after the buffer is full, the best value of each sensor is calculated and properly converted into the right unit. The read values are then sent to the database by Message Queue. After that the read values are compared with the system ideal values to verify if the water heater or a display warning is needed. In the end, if the water heater needs to be turned on or off, a *condWH* is sent to the *tWaterHeater* in order to actuate according to the state(Fig.12.12).

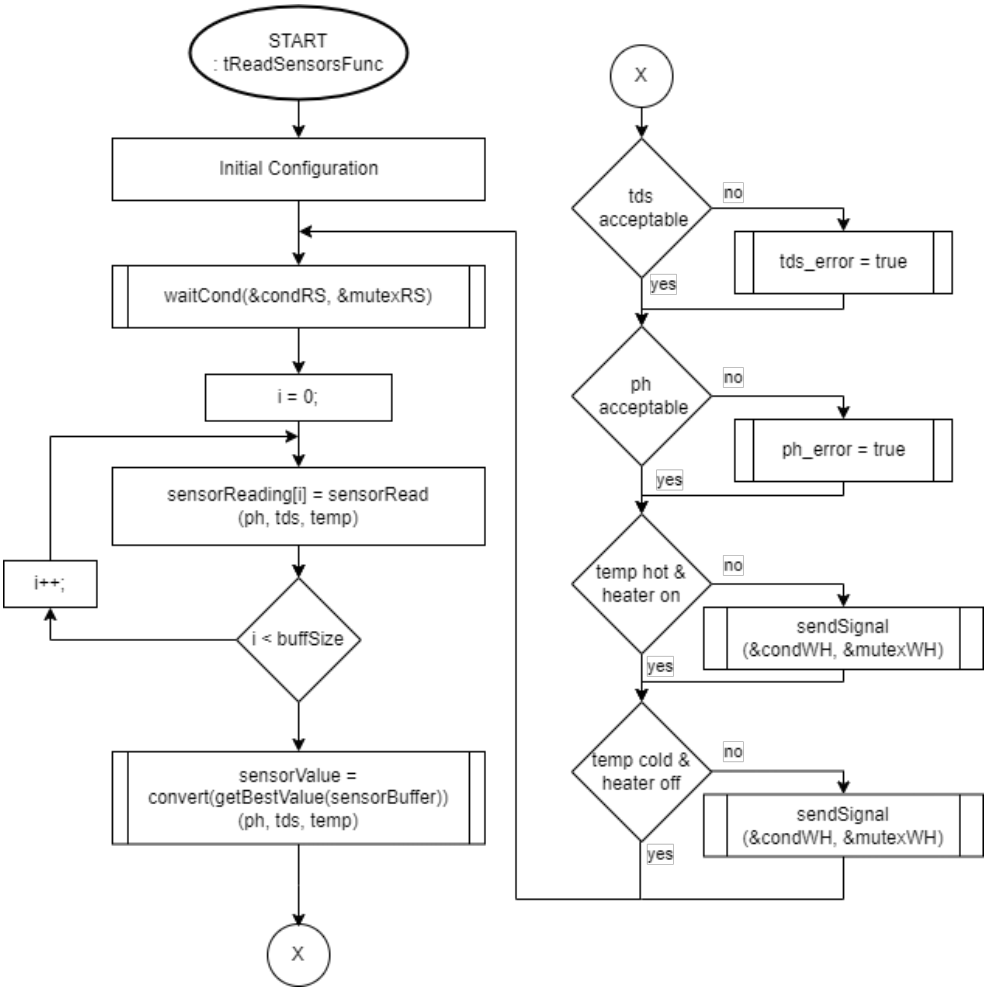

Figure 12.12: tReadSensors Thread Flowchart.

12.6.2 Actuators Threads's functions

tServoMotorFunc, *tWaterHeaterFunc* and *tUVLightFunc*, whose flowcharts are presented below (Fig.12.13), are the threads responsible for managing the actuators' actions. Their functioning is simple : after an initial configuration ,the thread is suspended until the respective condition variable (*condMotor*, *condWH* or *condLight*) is triggered; once it happens, in the *tServoMotorFunc* a function is called to rotate the servo

motor as many times as necessary in order to feed the fish, in the *tWaterHeaterFunc* according to the state the water heater is activated or deactivated and in the *tUVLightFunc* the UV light is turned on and a timer starts to count down the time until it turns off again.

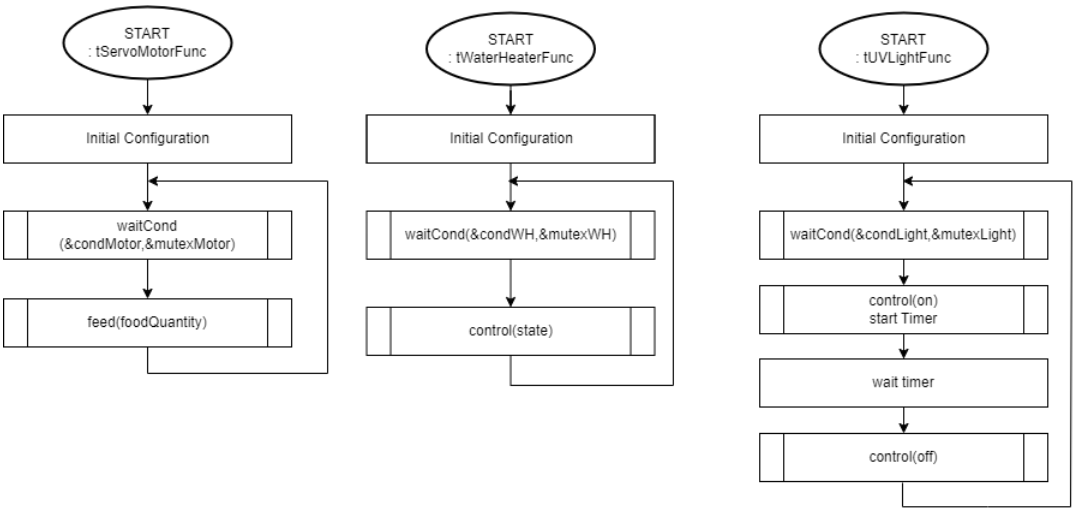

Figure 12.13: Actuators Threads Flowcharts.

12.6.3 tIdle's functions

The *tIdle* thread is responsible for maintaining the system in an "inactive" state until action is required. The following figure represents the respective flowchart (Fig.12.14).

The thread remains still unless a SIGALRM signal is detected, which trigger the condition variable *condTime* so a sensor's reading can be requested and the UV light, water heater or servo motor activation if necessary.

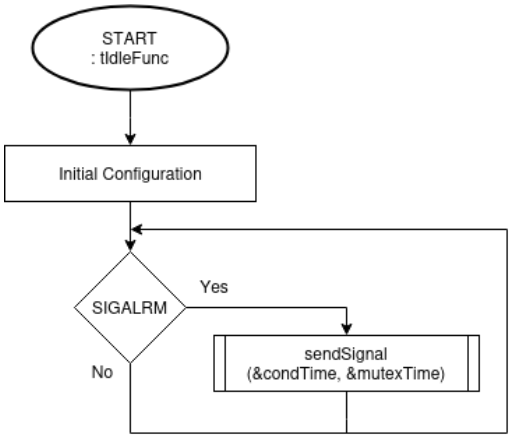

Figure 12.14: tIdle Thread Flowchart.

12.6.4 tSig's functions

The *tSig* thread, in Figure 12.15, after an initial configuration, waits for the trigger of the condition variable *condTime* in a suspended state. Once it leaves the suspended state, it compares the light time and the feed time with the current time: if the current time matches the feed time, the condition variable *condMotor* is triggered, so the stepper motor activates; if the current time matches the light time, the condition variable *condLight* is triggered so the UV light is turned on. In the end, the condition variable *condRS* is triggered to read the sensors and the thread returns to the suspended state.

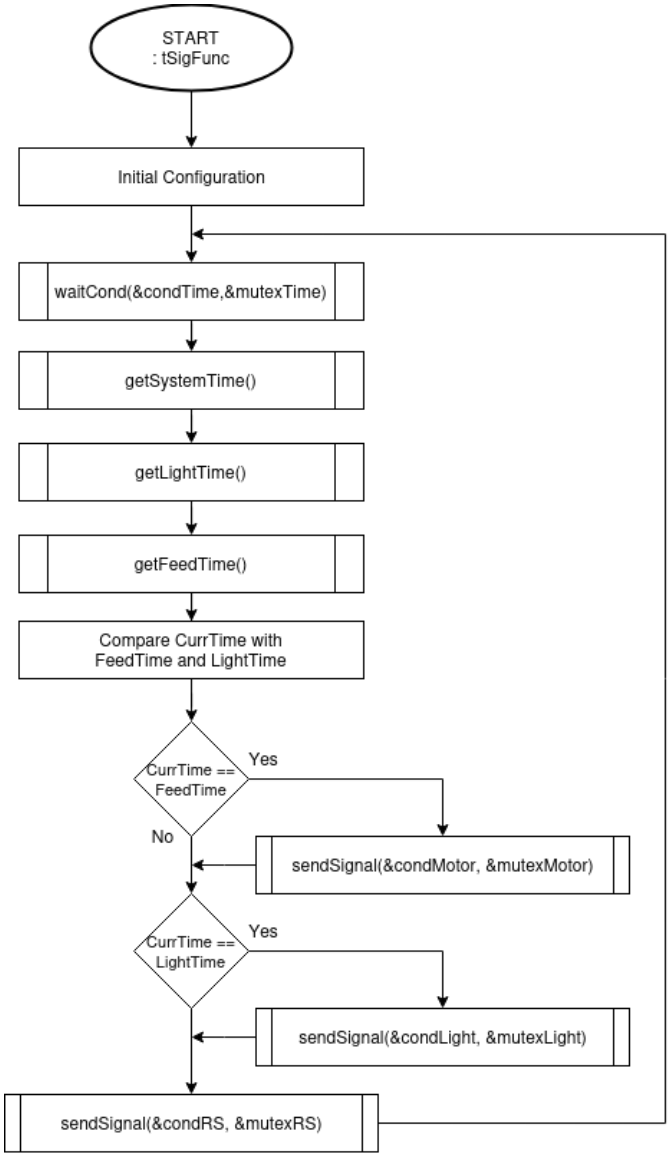

Figure 12.15: tSig Thread Flowchart.

12.6.5 dDatabaseManage Daemon's functions

The *dDatabaseManage* daemon is responsible for managing the database. Once a message queue is detected, its content is converted to SQL commands, the same commands are executed to the database and a new message queue is sent back with database data. After that, it waits again for a new message queue (Fig.12.16).

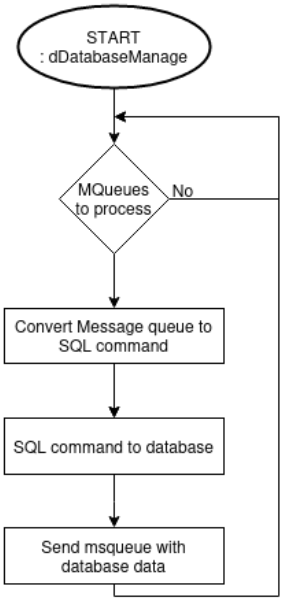

Figure 12.16: dDatabaseManage Daemon Flowchart.

12.7 GUI Layout

As previously stated, the interface between the system and the user is a touch screen display. Therefore, it's necessary to design the GUI so it is simple, intuitive, attractive and catchy. The main screen, represented by Figure 12.17, presents the sensor measurements values and three lines on the corner to open the "Options" menu tab.

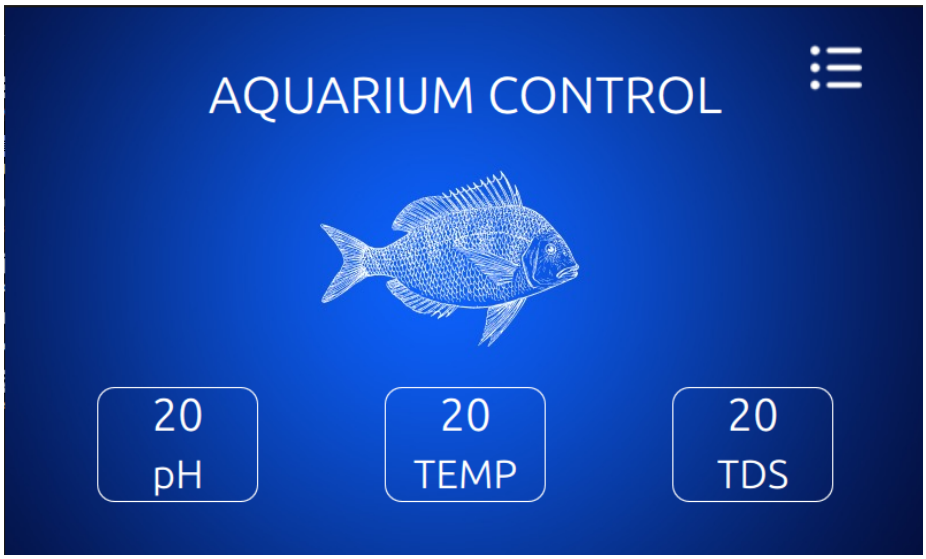

Figure 12.17: Main Screen Layout.

As represented previously, the main screen will only change if the TDS sensor or PH sensor readings present values outside the desired limits, and this will show a warning message to change the water in the aquarium (Fig 12.18).

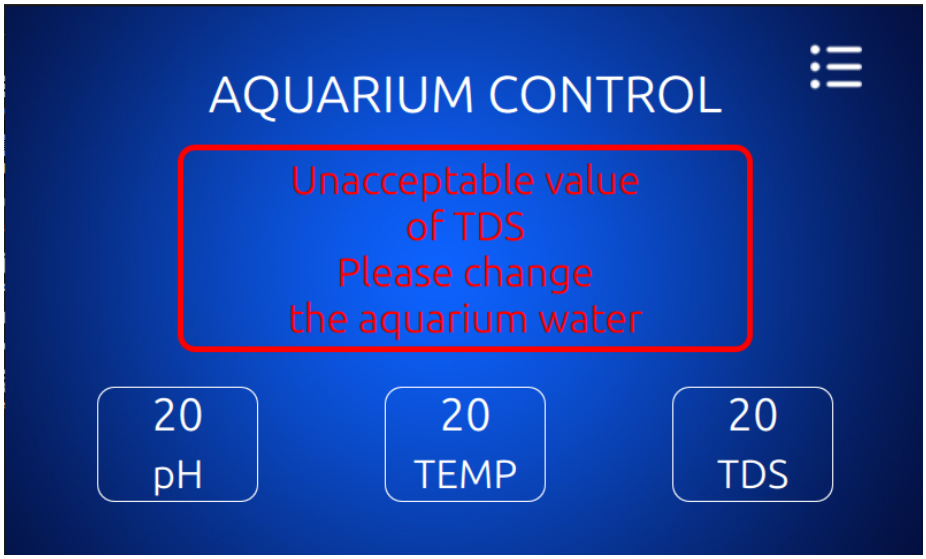

Figure 12.18: Main Screen Layout Warning.

The "Options" menu tab in Figure 12.19 contains four different buttons for the user to use: Add Fish, Remove Fish, System Control, Analysis and in the right top corner the aquarium current ideal conditions.

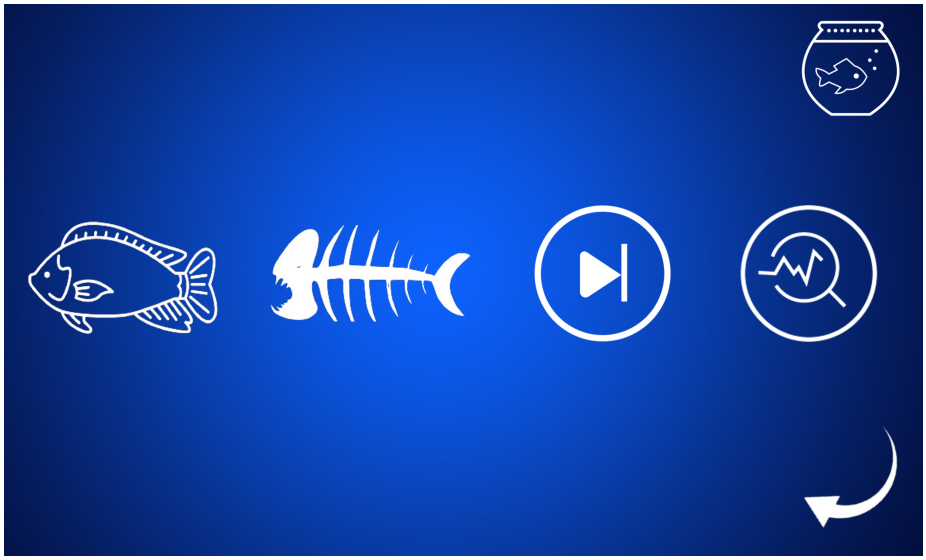

Figure 12.19: Options Menu Layout.

After pressing the "Add Fish" option, as can be seen in Figure 12.20, it is possible to view the species present in the database and select one of them to add to the system, according to the fish that the user is introducing in the aquarium.

Figure 12.20: Add Fish Interface.

Once the species to be added has been selected, its conditions will appear on the screen and if it can be added taking into account the current conditions of the aquarium, a "confirm" button will appear on the screen, otherwise this button will not appear, as can be seen in Figure 12.21.

Figure 12.21: Remove Fish Interface.

The "Remove Fish" screen interface (Fig.12.22) is similar to the "Add Fish". However the species that are presented are the ones currently habitating the aquarium and by selecting one of them, the species is removed from the database and the system.

Figure 12.22: Remove Fish Interface.

Once the option "System Control" is selected, a button to turn off the system is showed. Pressing this button causes the system to begin the shutdown process (Fig.12.23).

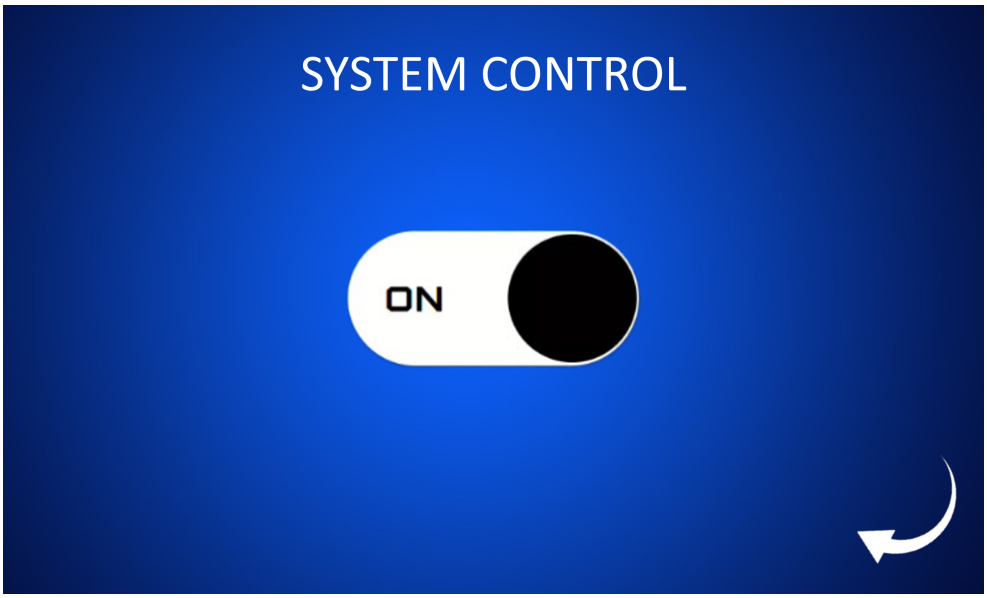

Figure 12.23: OFF Interface.

On the aquarium's current ideal conditions screen, it is possible to see the values for temperature, pH, TDS, food quantity, lighting time and feeding time (Fig.12.24).

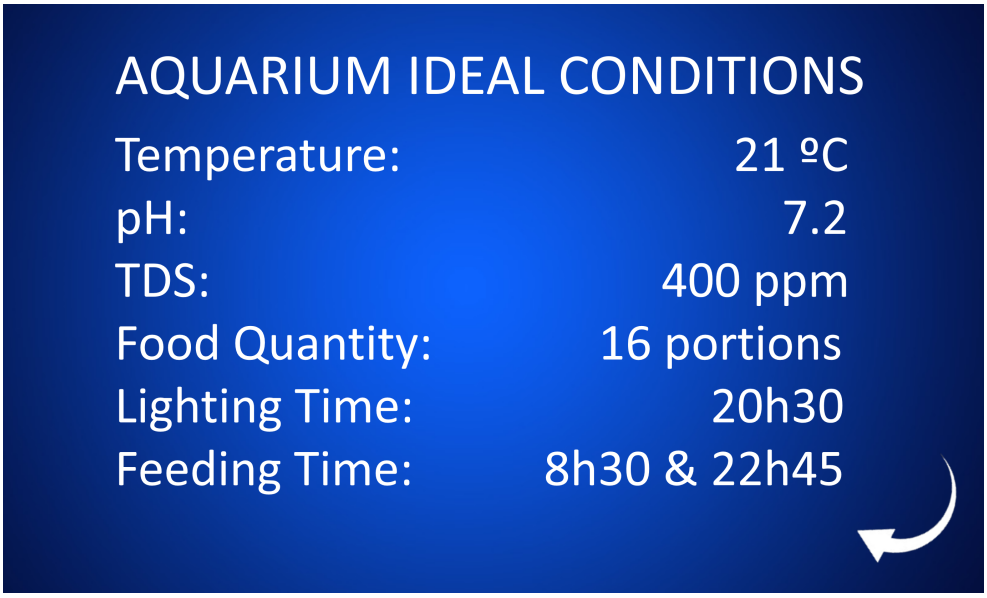

Figure 12.24: Ideal Conditions Interface.

On the aquarium's current ideal conditions screen, it is possible to see the values for temperature, pH, TDS, food quantity, lighting time and feeding time (Fig.12.25).

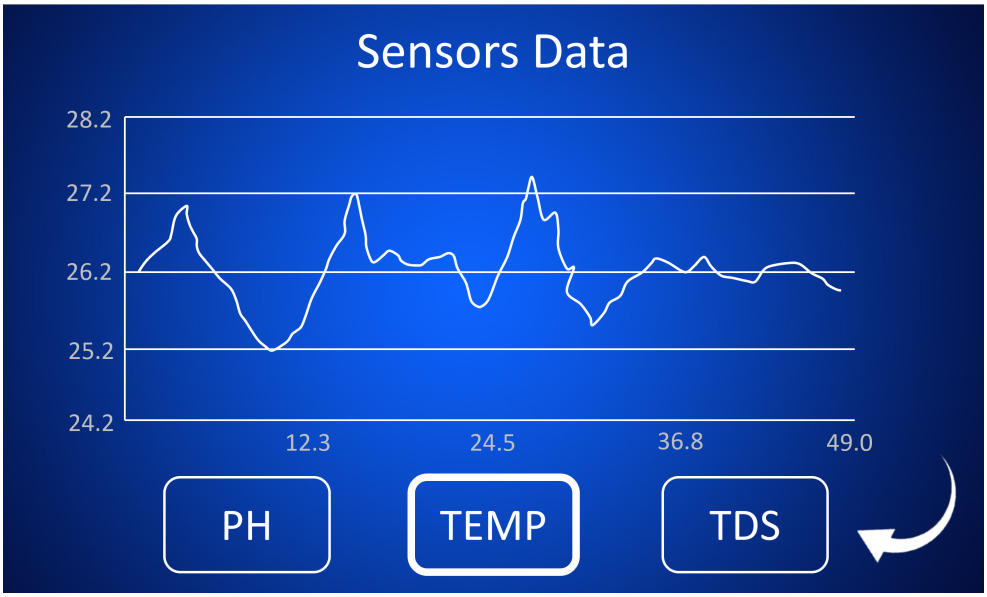

Figure 12.25: Sensors Graphs Interface.

13 | Boot Process Specification

The initialization process of an embedded system is an essential part of the correct functioning of the system in question and it follows a sequential logic. In Aquarium Control, the system boots as shown in Figure 26.1, starting with the hardware control initialization and ending with the software initialization.

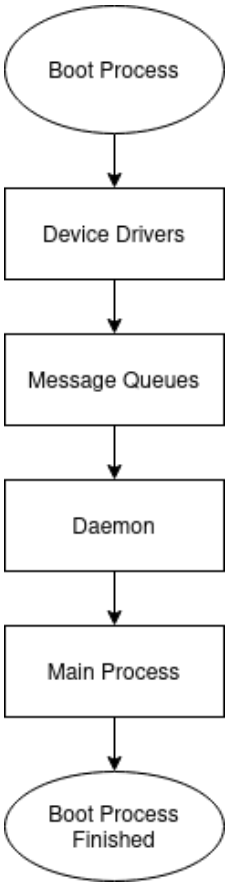

Figure 13.1: Boot System Startup.

14 | System Shutdown

The system shutdown process follows a sequence of system commands. Initially, the drivers "pwm" and "relay" are removed using the rmmod command, ensuring the clean termination of these drivers. Subsequently, the /dev/mqueue directory is unmounted, followed by the removal of the directory using the rm command. Finally, the system is turned off using the poweroff command, completing the shutdown procedure.

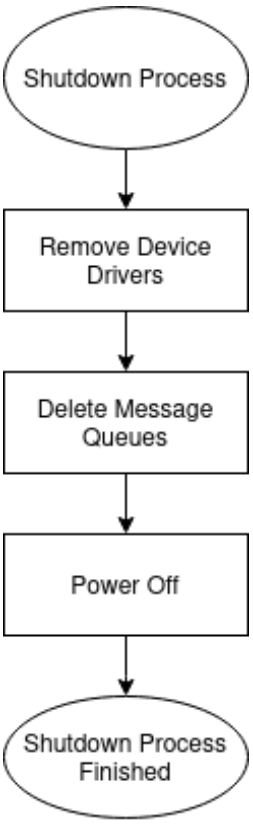

Figure 14.1: Shutdown Process.

```
void MainClass::shutdown() {
    system("rmmod pwm") ;
    system("rmmod relay");
    system("umount /dev/mqueue");
    system("rm -r /dev/mqueue");
    system("poweroff");
}
```

15 | Errors Handling Specification

15.1 Test Cases

Daemon Test Cases

Test	Description	Expected Result	Real Result
Receive Message Queues	Read Message Queues Received	Message Queue data correct	
Send Message Queues	Send a message queue	Message queue present in the /mqueue folder	
Deamon Execution	Check if daemon is executing	Daemon working in background	

Table 15.1: Power Supply Test Cases.

Sensors Test Cases

Test	Description	Expected Result	Real Result
Read Temperature	Verify if a correct temperature value is obtained	Correct temperature value obtained	
Read pH	Verify if a correct pH value is obtained	Correct pH value obtained	
Read TDS	Check if the TDS value is correct	Correct TDS value obtained	

Table 15.2: Sensor Test Cases.

Actuators Test Cases

Test	Description	Expected Result	Real Result
Actuate Servo Motor	Test Servo Motor rotation	Servo motor start rotating	
Actuate Water Heater	Test Water Heater heating process	Water heater start heating	
Actuate UV Light	Test Light activation	Activation of Light	
Actuate Relay Module	Test relay output activation	Transition between NO and NC	

Table 15.3: Sensor Test Cases.

Touchscreen Test Cases

Test	Description	Expected Result	Real Result
DSI Connection	Connect via DSI touchscreen	Display Image	
Input Response	Test the user input	Input is detected	
Display Update	Send data to display	Values change in the interface	

Table 15.4: Touch Screen Display Test Cases.

Database Test Cases

Test	Description	Expected Result	Real Result
Insert Data	Insert data into the database	Data inserted with sucess	
Remove data	Remove data from the database	Data removed with sucess	
Get Fish Information	Receive a specific fish information	Fish information received	
Get All Species	Receive all the species	All the species received	
Get Aquarium Species	Receive Aquarium species	Aquarium species received	
Sensors Data	Insert sensors values	Sensors values into database	

Table 15.5: Database Test Cases.

Power Supply Test Cases

Test	Description	Expected Result	Real Result
Power Raspberry	Connect supply to the raspberry	Raspberry turn On	
Power Sensors	Connect supply to the sensors	Sensors Leds On	
Power Actuators	Connect supply to the actuators	Actuators Leds On	

Table 15.6: Power Supply Test Cases.

GUI Test Cases

Test	Description	Expected Result	Real Result
Add Fish	Add a fish into the Aquarium	Fish added to database	
Remove Fish	Remove a fish from the Aquarium	Fish added to database	
Sensors Read	Read sensor values	Sensor Values displayed	
Sensors Graphs	Display sensor graph values	Sensor graph values displayed	
Ideal Conditions	Display ideal conditions	Show ideal conditions	
Fish Conditions	Display selected fish conditions	Show selected fish conditions	
Display Species to Add	Display species presented in the database	Show all the species	
Display Species to Remove	Display species presented in the aquarium	Show the species in the aquarium	
Shutdown Button	Turn off the system	The system turns off	

Table 15.7: GUI Test Cases.

16 | Hardware Implementation

16.1 Structure

The project structure was designed to hide all existing cables and connections between sensors, actuators and modules. The user is only presented with a touchscreen with all the necessary features and cables to connect the sensors and actuators inside the aquarium to be used (Figures **??** & **??**).

The materials used in the structure were wood and chipboard.

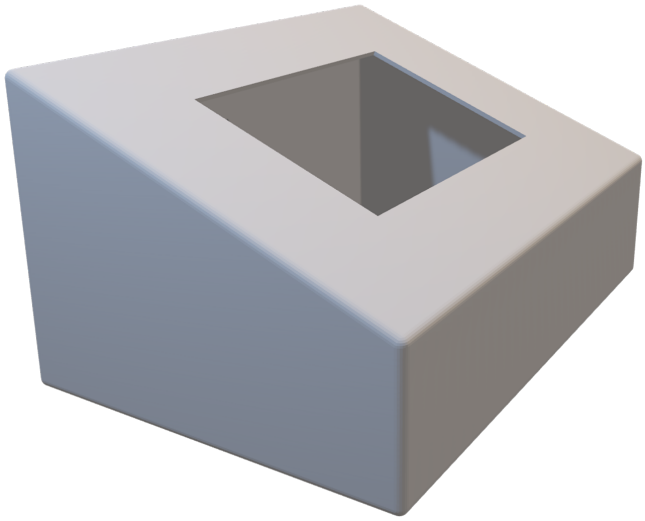

Figure 16.1: Structure Front View.

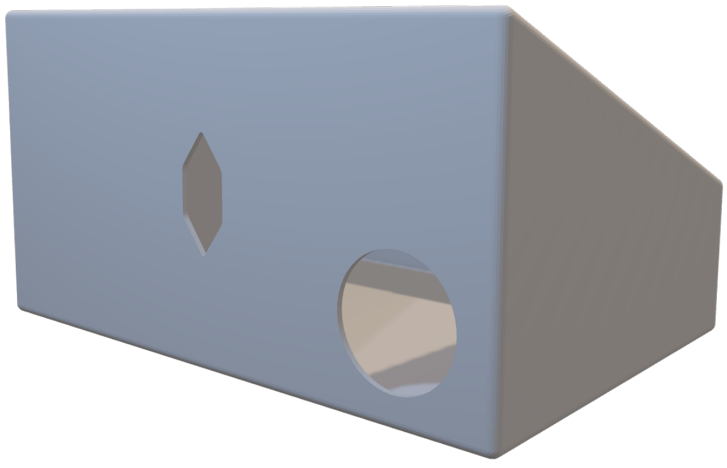

Figure 16.2: Structure Back View.

16.2 Feeder

For the feeder, it was 3D printed using an open source design [23] that matched the project needs. There were only small changes in its application as in our case we are using a Raspberry Pi and distanced from the servo motor that is used to feed the fish (Figure 16.3).

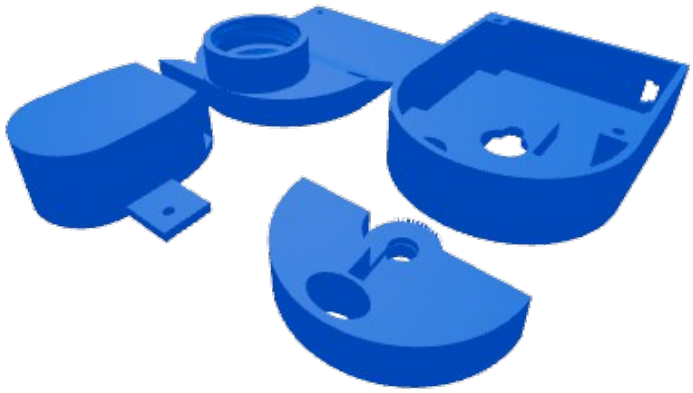

Figure 16.3: Feeder.

16.3 PCB

Taking into account that the aquarium project uses an RTC, an ADS1115, 3 sensors and 3 actuators, there was a need to create a PCB in order to power everything and simplify connections to them. In this way, the PCB represented in Figure 16.4 was created.

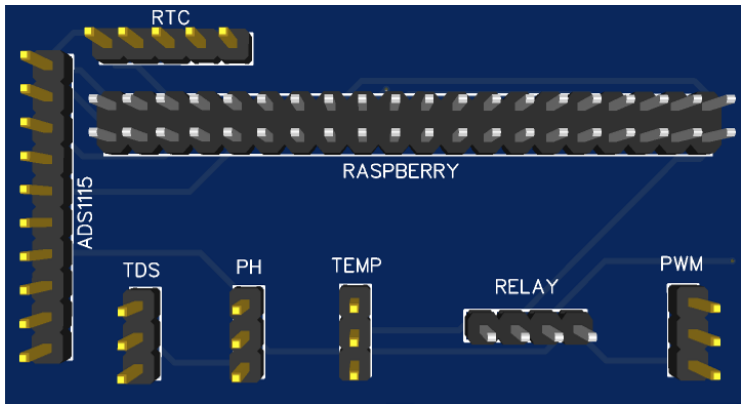

Figure 16.4: PCB 3D View.

17 | System Configuration

17.1 Buildroot

As a constrain and necessity to create a custom linux image to run our system in the raspberry pi environment, buildroot was used. Firstly, it was necessary to navigate to the buildroot installation path, open the terminal and run the command **$ make raspberrypi4 defconfig**.

After having performed the previously mentioned make, in order to configure the image according to the project's needs, the **$ make menuconfig** command is used, thus presenting a graphical interface in the terminal in order to facilitate the selection of packages (Figure 17.1).

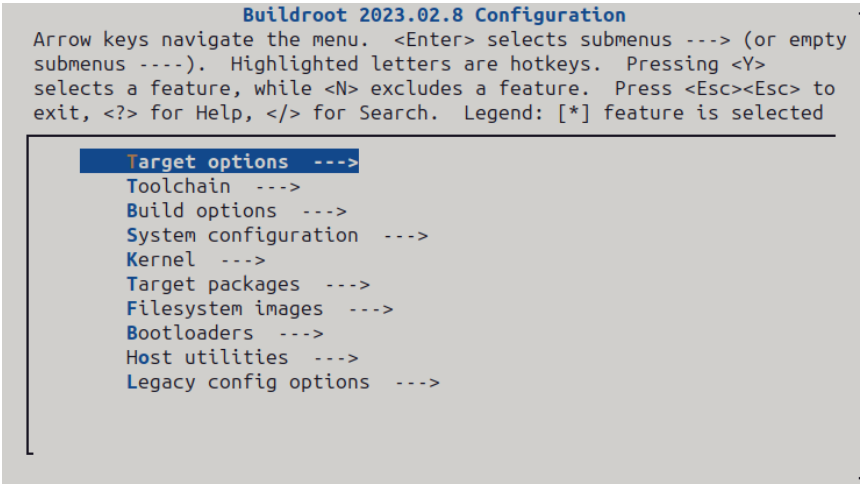

Figure 17.1: Buildroot Main Menu.

In the toolchain menu, glibc was used as C library (default in this buildroot version) and C++ support was enabled (Figure 17.2).

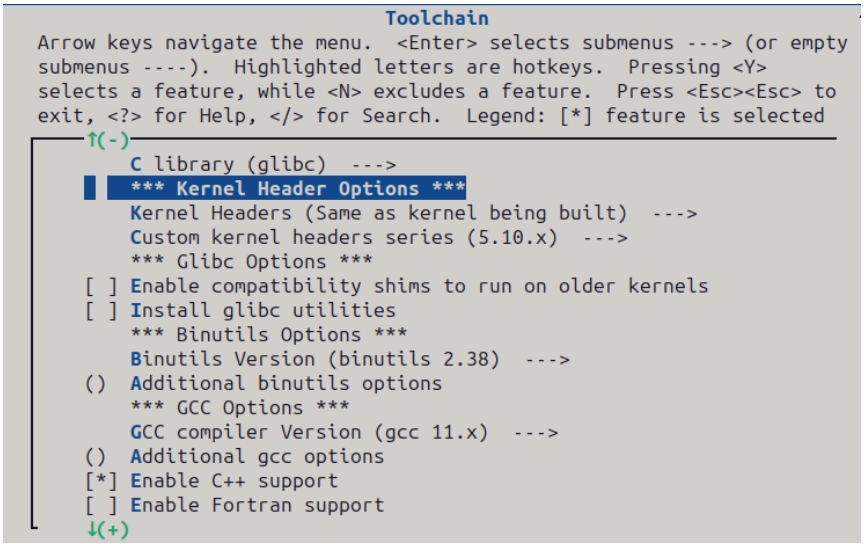

Figure 17.2: Toolchain Menu.

In the System configuration menu, root login with password was enabled.(Figure 17.3).

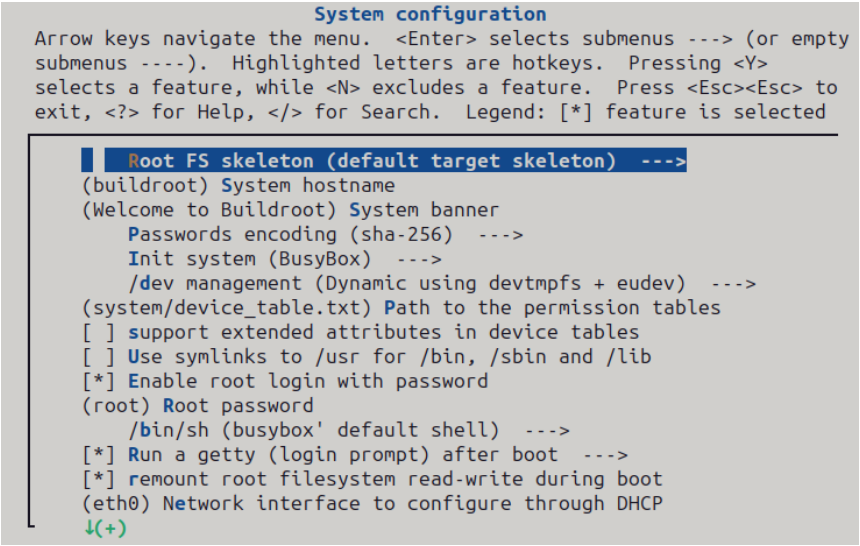

Figure 17.3: System Configuration Menu.

In the Filesystem images menu, the size of the image was increased to 400M to ensure there is space for all packages and also tar the root filesystem was enable with compress method bzip2 (Figure 17.4).

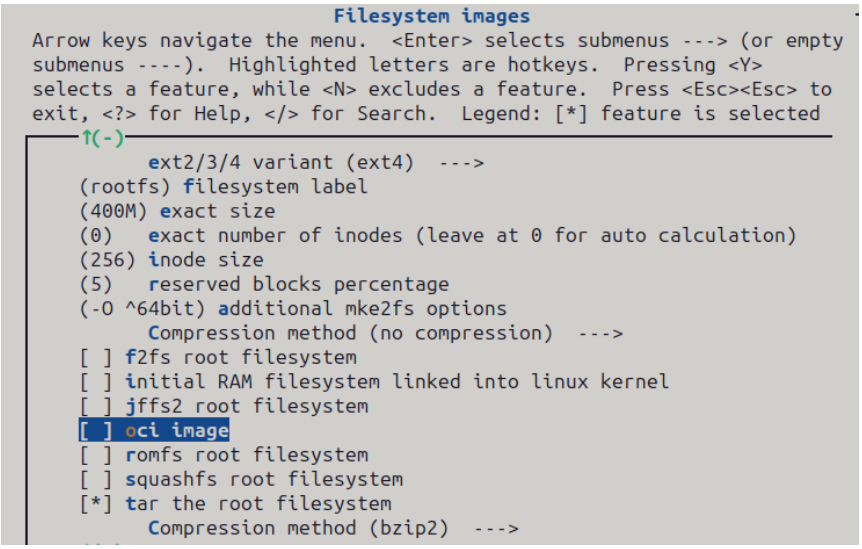

Figure 17.4: Filesystem Images Menu.

To use qt5 as a programming interface for the raspberry pi and to create a graphical environment for communication between the user and the system, the configurations present in 17.5 and 17.6 were used.

The default graphical platform should be "linuxfb" since it will serve as the foundational platform for the final solution created with Qt. Key modules, such as Qt5quickcontrols2, qt5declarative, and qtquick, are crucial due to their extensive libraries that support the running application built with QML and C++.

Target Packages » Graphic libraries and applications » Qt5

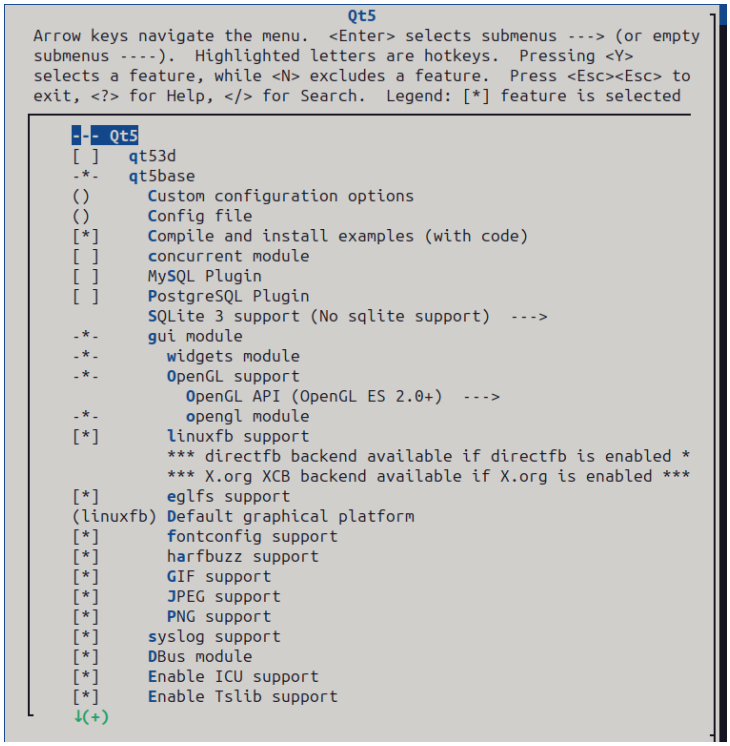

Figure 17.5: Qt5 Configuration Menu.

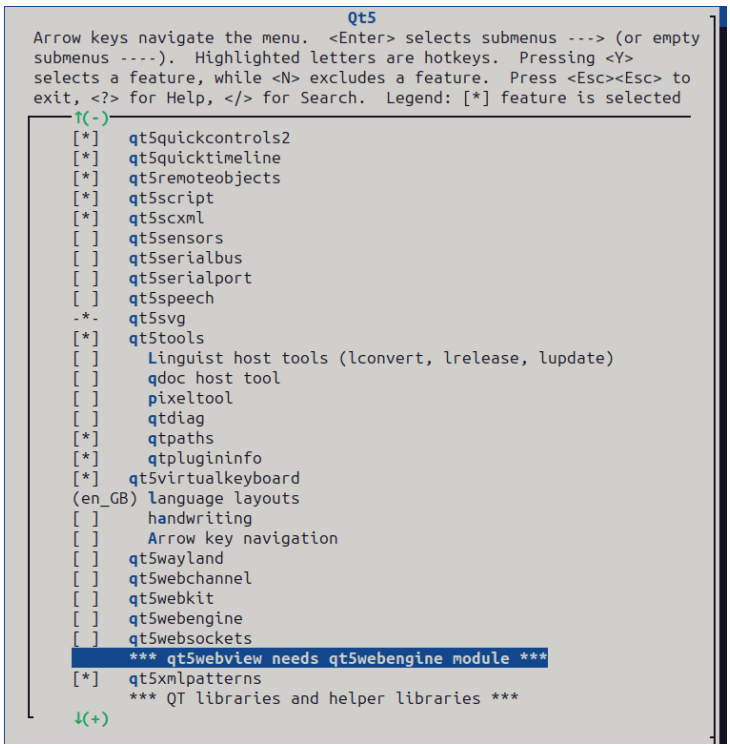

Figure 17.6: Qt5 Configuration Menu.

To activate the touch functionality of the selected screen, it is necessary to enable the libinput library. This library is designed to manage input devices in Wayland, offering a generic input driver support (Figure 17.7)

Target Packages » Libraries » Hardware Handling

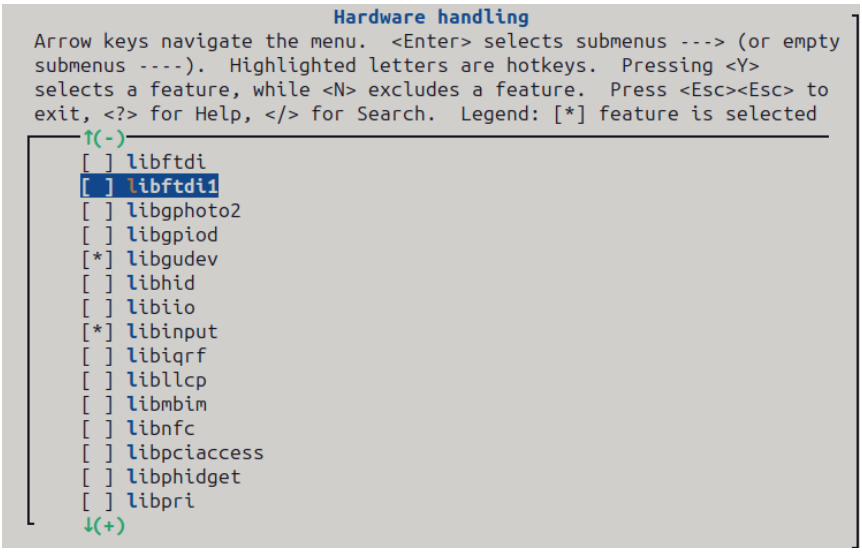

Figure 17.7: Hardware Handling Menu.

To communicate with the Raspberry Pi over Ethernet, activate DHCP for IP address assignment and enable the lightweight dropbear package, optimizing system resources for SSH protocol use on embedded devices (Figure 17.8).

Target Packages » Networking applications

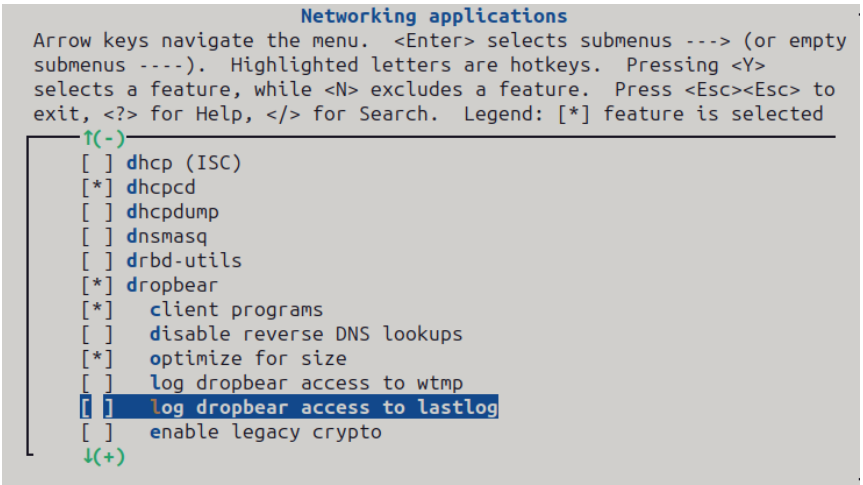

Figure 17.8: Networking applications Menu.

For Qt5 development on Raspberry Pi, the use of OpenSSH and Rsync is crucial, with OpenSSH ensuring secure communication for remote file deployment, and Rsync simplifying file transfers by syncing only the essential changes (Figures 17.9 and 17.10).

Target Packages » Networking applications

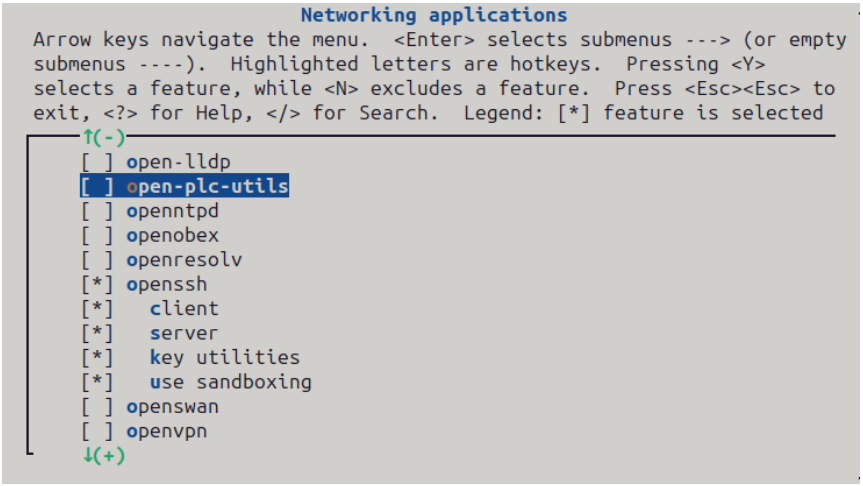

Figure 17.9: Networking applications Menu.

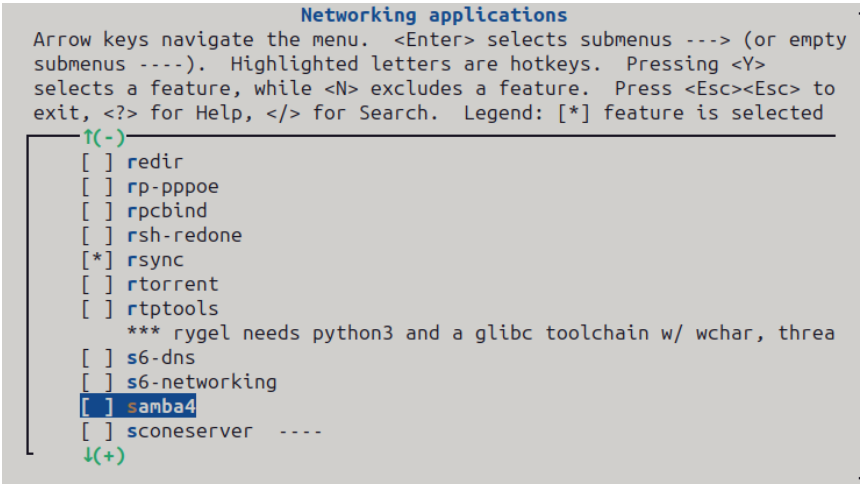

Figure 17.10: Networking applications Menu.

In order to use the sqlite3 database for this project, it was also necessary to add the packages in buildroot corresponding to it (Figures 17.12)

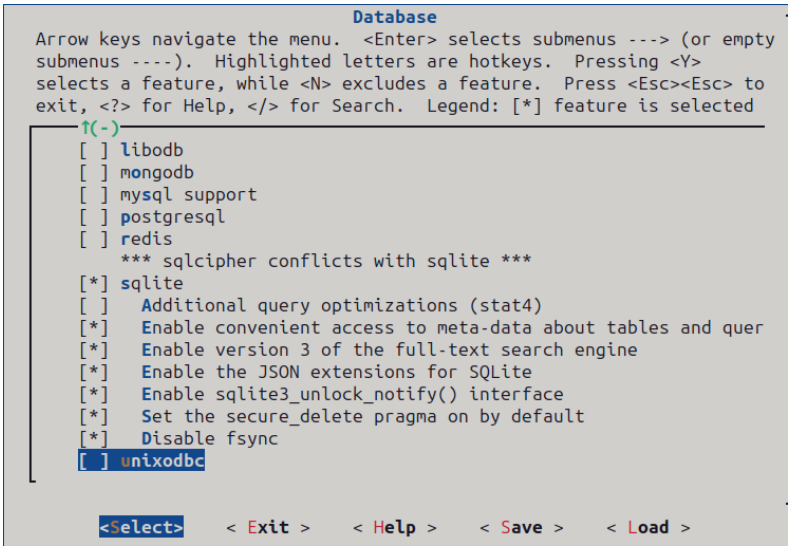

Figure 17.11: Sqlite3 Configuration Menu.

Image Generation

After all of the packages have been selected run **$ make** to create the custom image.

The image can be flashed to the sdcard using the **$ sudo dd if=output/images/sdcard.img of=/dev/sda** command.

17.2 Qt5

The first step is to create a device, to do this go to **tools » options » devices**, create a new generic linux device and define the raspberry host name and username. Note: host name can be found using **$ arp -a** command in the terminal (Figure 17.12).

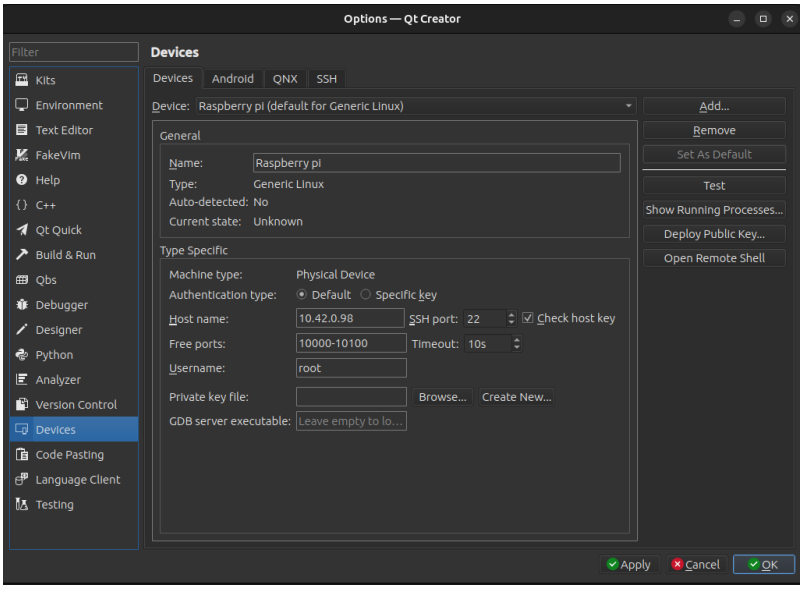

Figure 17.12: Qt5 Devices Menu.

After all the device configuration it is possible to use the ”test” button to verify if the connection worked successful 17.13).

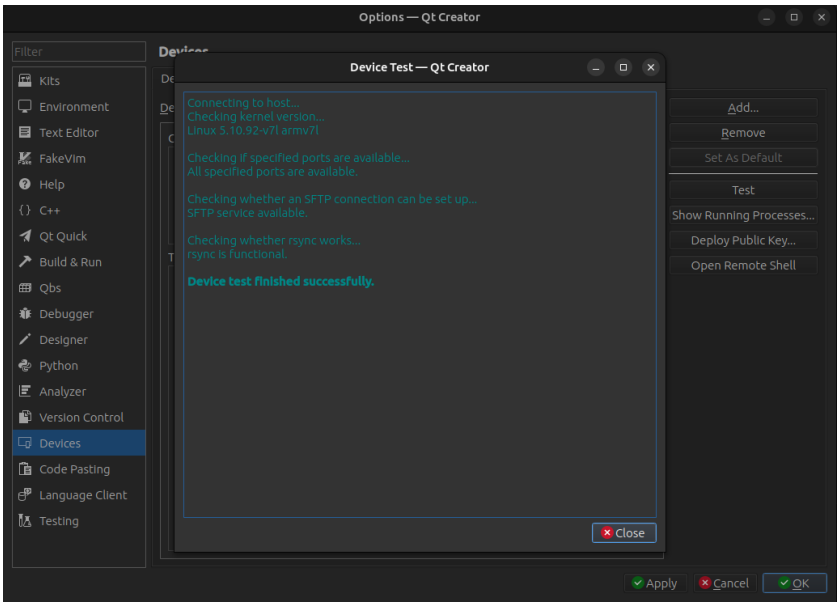

Figure 17.13: Qt5 Test.

In the kits tab, you need to configure the debugger. To do this, you first need to run the command **$ sudo apt-get install gdb-multiarch** and then manually add a debugger and define the multiarch path 17.14).

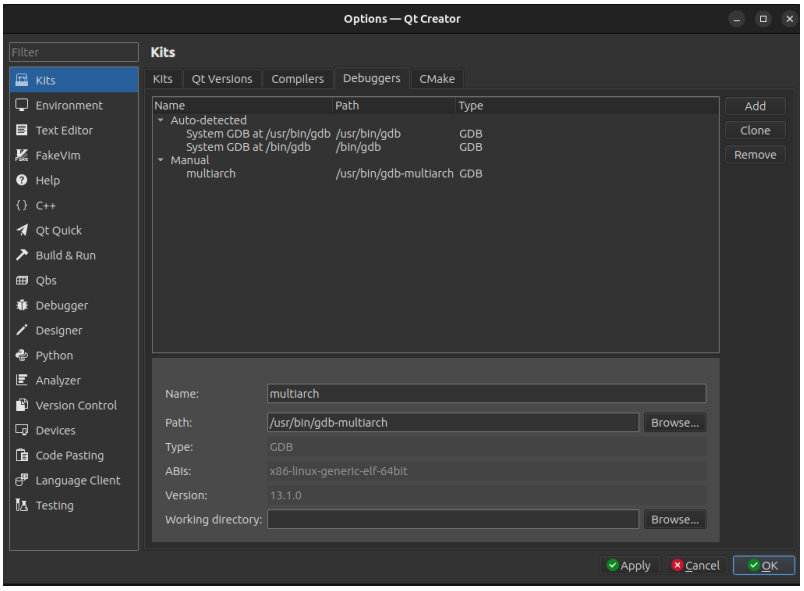

Figure 17.14: Qt5 Debugger.

In order to compile the project for the Raspberry Pi, the compiler path was defined as: *buildroot-2023.02.8 /output/host/bin/arm-buildroot-linux-gnueabihf-gcc* for the C compiler 17.15).

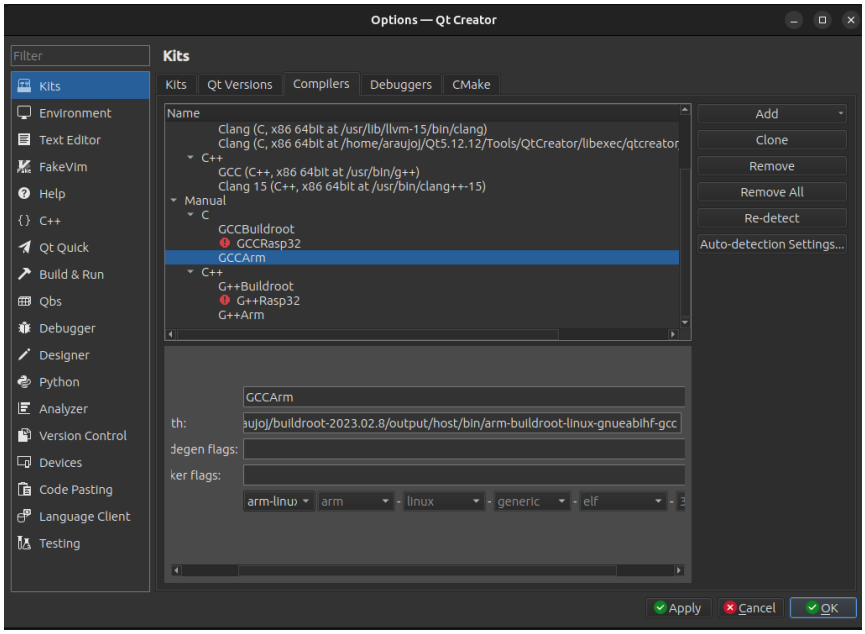

Figure 17.15: Qt5 C Compiler.

Also, as the project will use c++ programming, the following path: *buildroot-2023.02.8/output/host/bin/arm-buildroot-linux-gnueabihf-g++* was used as the c++ compiler 17.16).

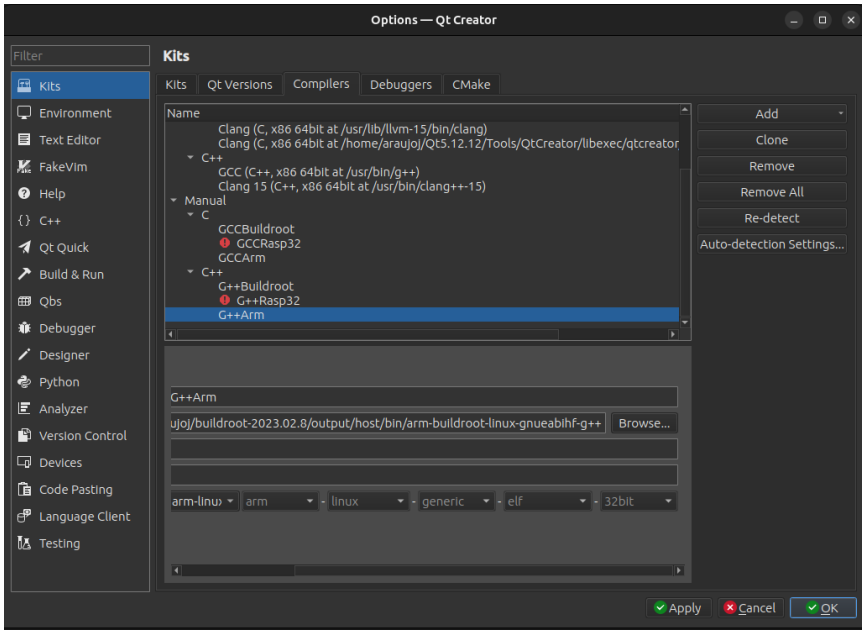

Figure 17.16: Qt5 C++ Compiler.

In the qt versions tab, it is necessary to define the path for the qt version in this case: *buildroot-2023.02.8/output/host/bin/qmake* 17.17).

Finally, in the kits add a new one and use all the configurations made previously, including the device

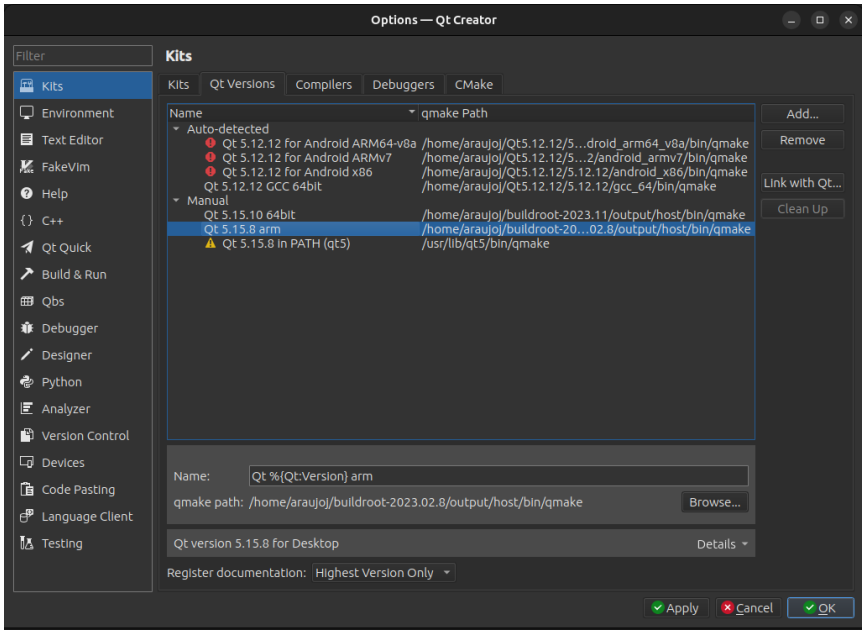

Figure 17.17: Qt5 Versions.

type, the compilers, the debugger and the qt version. Once this is done, QT is ready to start working with the Raspberry Pi 17.18).

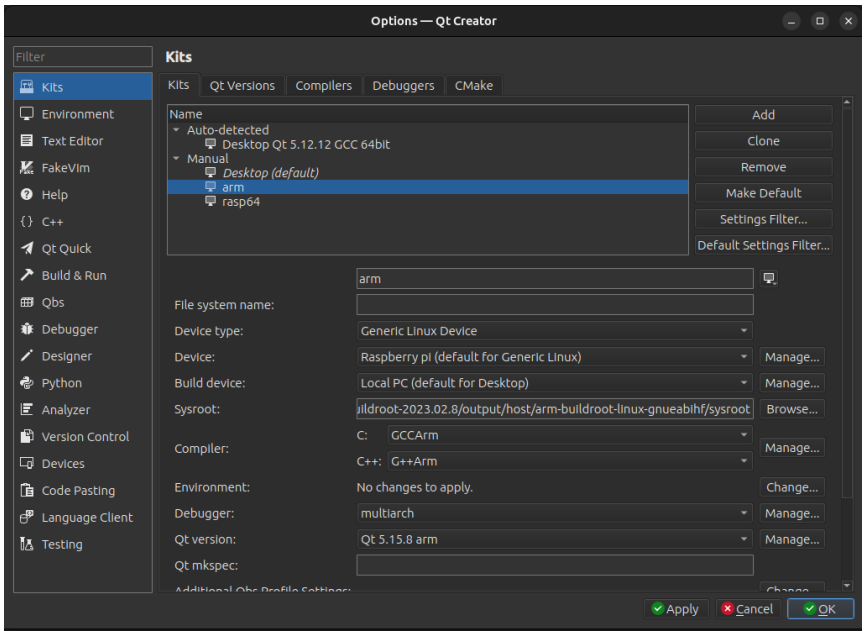

Figure 17.18: Qt5 Kits.

18 | System Initialization

The Raspberry Pi boot sequence entails the initialization and configuration of the Linux system, guided by user-defined parameters stored in the SD card. Some adjustments have been implemented to guarantee the presence of essential drivers, ensuring optimal functionality for the system.

18.1 Config.txt

Unlike regular computers that use a BIOS, Raspberry Pi relies on a config file named "config.txt" for settings. During the buildroot image creation, default lines were added to this file and should be kept. Additional lines like *dtoverlay=w1-gpio* and *dtparam=i2c1=on* were appended to enable the W1 protocol for sensors and activate I2C, allowing sensor integration in the project.

```

start_file=start4.elf
fixup_file=fixup4.dat

kernel=zImage

# To use an external initramfs file
#initramfs rootfs.cpio.gz

# Disable overscan assuming the display supports displaying the full resolution
# If the text shown on the screen disappears off the edge, comment this out
disable_overscan=1

# How much memory in MB to assign to the GPU on Pi models having
# 256, 512 or 1024 MB total memory
gpu_mem_256=100
gpu_mem_512=100
gpu_mem_1024=100

# fixes rpi (3B, 3B+, 3A+, 4B and Zero W) ttyAMA0 serial console
dtoverlay=miniuart-bt

# enable autoprobing of Bluetooth driver without need of hciattach/btattach
dtparam=krnbt=on

#enable uart communication
enable_uart=1

#enable w1 bus
dtoverlay=w1-gpio

#enable i2c bus
dtparam=i2c1=on
```

18.2 Init Script

As the system boots up, essential configurations for the project are initiated through a dedicated bash script named *ScriptAQCtrl.sh*. Located in */etc/init.d* on the Raspberry Pi, this script not only launches the necessary drivers for sensors but also synchronizes the system time with the RTC (Real-Time Clock) in the project, ensuring precise timekeeping and integration of sensor functionalities. At the end, it initiates the daemon and the main process for the project to function as intended.

```
#!/bin/bash

# path to daemon executable
daemonEXE=/etc/Aquarium/daemonRasp

# path to qt application executable
qtAppEXE=/etc/Aquarium/aquariumAPP

# i2c modules
modprobe i2c-dev
modprobe i2c-bcm2835

# one wire config
modprobe w1-gpio
modprobe w1-therm

# rtc config
modprobe rtc-ds1307
echo ds1307 0x68 >> /sys/class/i2c-adapter/i2c-1/new_device
hwclock -s

# drivers
insmod /etc/Aquarium/DDrivers/relay.ko
insmod /etc/Aquarium/DDrivers/pwm.ko

# message queue
mkdir /dev/mqueue
mount -t mqueue none /dev/mqueue

# execute daemon in background
exec $daemonEXE &

# wait 0.5 sec so everything is properly initialized
sleep 0.5

# executes qt application
exec $qtAppEXE
```

19 | Device Drivers

The project incorporates two device drivers: one for managing the servo motor and another for controlling the relay module. These drivers were developed to simplify hardware interaction without requiring in-depth familiarity with its specific details.

Communication between user space programs and the Linux kernel is facilitated through system calls, including *open()*, *close()*, *read()* and *write()*. The primary interface between a device driver and the Linux kernel consists of a standardized set of entry points and a file operations structure.

19.1 Relay Driver

In order to be able to activate the UV light and water heater of our system, there was a need to create a relay driver as previously mentioned. This driver has the functionality to activate and deactivate the output of two gpio pins, thus using *write()* as a bridge for the user, and it is not necessary to implement *read()* in this case, but as the functions are in pairs it exists with a empty code.

relayDD.c

```
ssize_t relay_device_write(struct file *pfile, const char __user *pbuff, size_t len,
↪  loff_t *off) { //tirar o static
        struct GpioRegisters *pdev;
        uint8_t pin = 0;
        uint8_t state = 0;

        pr_alert("%s: called (%u)\n",__FUNCTION__,len);

        if(unlikely(pfile->private_data == NULL))
                return -EFAULT;

        pdev = (struct GpioRegisters *)pfile->private_data;

        pin = (pbuff[0] - '0') * 10 + (pbuff[1] - '0');
        state = pbuff[2] - '0';

    SetGPIOOutputValue(pdev, pin, state);

        return len;
}

/*(...)*/

static struct file_operations relayDevice_fops = {
        .owner = THIS_MODULE,
        .write = relay_device_write,
        .read = relay_device_read,
        .release = relay_device_close,
        .open = relay_device_open,
};

static int __init relayModule_init(void) {
        int ret;
        struct device *dev_ret;

        printk("init started");
```

```
36
37          pr_alert("%s: called\n",__FUNCTION__);
38
39          if ((ret = alloc_chrdev_region(&relayDevice_majorminor, 0, 1, DEVICE_NAME)) <
    ↪  0) {
40                  return ret;
41          }
42
43          if (IS_ERR(relayDevice_class = class_create(THIS_MODULE, CLASS_NAME))) {
44                  unregister_chrdev_region(relayDevice_majorminor, 1);
45                  return PTR_ERR(relayDevice_class);
46          }
47          if (IS_ERR(dev_ret = device_create(relayDevice_class, NULL,
    ↪  relayDevice_majorminor, NULL, DEVICE_NAME))) {
48                  class_destroy(relayDevice_class);
49                  unregister_chrdev_region(relayDevice_majorminor, 1);
50                  return PTR_ERR(dev_ret);
51          }
52
53          cdev_init(&c_dev, &relayDevice_fops);
54          c_dev.owner = THIS_MODULE;
55          if ((ret = cdev_add(&c_dev, relayDevice_majorminor, 1)) < 0) {
56                  printk(KERN_NOTICE "Error %d adding device", ret);
57                  device_destroy(relayDevice_class, relayDevice_majorminor);
58                  class_destroy(relayDevice_class);
59                  unregister_chrdev_region(relayDevice_majorminor, 1);
60                  return ret;
61          }
62
63
64          s_pGpioRegisters = (struct GpioRegisters *)ioremap(GPIO_BASE, sizeof(struct
    ↪  GpioRegisters));
65
66          pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pGpioRegisters);
67
68          SetGPIOFunction(s_pGpioRegisters, 23, 0b001); //Output
69          SetGPIOFunction(s_pGpioRegisters, 24, 0b001); //Output
70          SetGPIOOutputValue(s_pGpioRegisters, 23, 1);
71          SetGPIOOutputValue(s_pGpioRegisters, 24, 1);
72          printk("init ended");
73
74          return 0;
75  }
76
77  static void __exit relayModule_exit(void) {
78
79          pr_alert("%s: called\n",__FUNCTION__);
80
81          SetGPIOFunction(s_pGpioRegisters, 23, 0b000);
82          SetGPIOFunction(s_pGpioRegisters, 24, 0b000);
83          iounmap(s_pGpioRegisters);
84          cdev_del(&c_dev);
85          device_destroy(relayDevice_class, relayDevice_majorminor);
86          class_destroy(relayDevice_class);
87          unregister_chrdev_region(relayDevice_majorminor, 1);
```

```
88 }
89
90 module_init(relayModule_init);
91 module_exit(relayModule_exit);
```

utils.c

In this section of code, the necessary functions are implemented, one of them to define the operating mode of the gpio pin and another to set or reset the bit, depending on whether we want to turn on or off an actuator.

```
1  void SetGPIOFunction(struct GpioRegisters *s_pGpioRegisters, int GPIO, int
   ↪  functionCode) {
2          int registerIndex = GPIO / 10;
3          int bit = (GPIO % 10) * 3;
4
5          unsigned oldValue = s_pGpioRegisters->GPFSEL[registerIndex];
6          unsigned mask = 0b111 << bit;
7
8          pr_alert("%s: register index is %d\n",__FUNCTION__,registerIndex);
9          pr_alert("%s: mask is 0x%x\n",__FUNCTION__,mask);
10         pr_alert("%s: update value is 0x%x\n",__FUNCTION__,((functionCode << bit) &
   ↪  mask));
11
12         s_pGpioRegisters->GPFSEL[registerIndex] = (oldValue & ~mask) | ((functionCode
   ↪  << bit) & mask);
13 }
14
15 void SetGPIOOutputValue(struct GpioRegisters *s_pGpioRegisters, int GPIO, bool
   ↪  outputValue) {
16
17         pr_alert("%s: register value is 0x%x\n",__FUNCTION__,(1<<(GPIO %32)));
18
19         if (outputValue)
20                 s_pGpioRegisters->GPSET[GPIO / 32] = (1 << (GPIO % 32));
21         else
22                 s_pGpioRegisters->GPCLR[GPIO / 32] = (1 << (GPIO % 32));
23 }
```

19.2 PWM Driver

In order to be able to feed the fish in the aquarium, there is a servo motor responsible for this function. Therefore, it was necessary to create a pwm driver since this is essential in controlling servo motors.

It should be noted that this driver is only made for a pwm0_0 on gpio12.

pwmDD.c

This first section represents the code implemented for a pwm driver, with an *init* where all physical registers are mapped to virtual registers for later use, an *exit* to deallocate this previously mapped space and a *write* so the programmer can use the driver at a more abstract level.

```
ssize_t pwm_device_write(struct file *pfile, const char __user *pbuff, size_t len, loff_t *off) {
        struct GpioRegisters *pdev;
        unsigned int freq = 0;
        unsigned int duty = 0;

        pr_alert("%s: called (%u)\n",__FUNCTION__,len);

        if(unlikely(pfile->private_data == NULL))
                return -EFAULT;

        pdev = (struct GpioRegisters *)pfile->private_data;

        freq = (pbuff[0] - '0') * 10000 + (pbuff[1] - '0') * 1000 + (pbuff[2] - '0') * 100 + (pbuff[3] - '0') * 10 + (pbuff[4] - '0');
        duty = (pbuff[5] - '0') * 100 + (pbuff[6] - '0') * 10 + (pbuff[7] - '0');

        ConfigPwmSignal(freq,duty);

        return len;
}

static struct file_operations pwmDevice_fops = {
        .owner = THIS_MODULE,
        .write = pwm_device_write,
        .read = pwm_device_read,
        .release = pwm_device_close,
        .open = pwm_device_open,
};

static int __init pwmModule_init(void) {
        int ret;
        struct device *dev_ret;

        printk("init started");

        pr_alert("%s: called\n",__FUNCTION__);

        if ((ret = alloc_chrdev_region(&pwmDevice_majorminor, 0, 1, DEVICE_NAME)) < 0) {
                return ret;
        }

        if (IS_ERR(pwmDevice_class = class_create(THIS_MODULE, CLASS_NAME))) {
                unregister_chrdev_region(pwmDevice_majorminor, 1);
                return PTR_ERR(pwmDevice_class);
        }
        if (IS_ERR(dev_ret = device_create(pwmDevice_class, NULL, pwmDevice_majorminor, NULL, DEVICE_NAME))) {
                class_destroy(pwmDevice_class);
                unregister_chrdev_region(pwmDevice_majorminor, 1);
                return PTR_ERR(dev_ret);
        }

        cdev_init(&c_dev, &pwmDevice_fops);
```

```
52          c_dev.owner = THIS_MODULE;
53          if ((ret = cdev_add(&c_dev, pwmDevice_majorminor, 1)) < 0) {
54                  printk(KERN_NOTICE "Error %d adding device", ret);
55                  device_destroy(pwmDevice_class, pwmDevice_majorminor);
56                  class_destroy(pwmDevice_class);
57                  unregister_chrdev_region(pwmDevice_majorminor, 1);
58                  return ret;
59          }
60
61          s_pGpioRegisters = (struct GpioRegisters *)ioremap(GPIO_BASE, sizeof(struct
    ↪  GpioRegisters));
62          pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pGpioRegisters);
63          s_pPwmCtlRegisters = (volatile unsigned int *)ioremap(PWM_CTL, sizeof(unsigned
    ↪  int));
64          pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pPwmCtlRegisters);
65          s_pPwmStaRegisters = (volatile unsigned int *)ioremap(PWM_STA, sizeof(unsigned
    ↪  int));
66          pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pPwmStaRegisters);
67          s_pPwmRng1Registers = (volatile unsigned int *)ioremap(PWM_RNG1,
    ↪  sizeof(unsigned int));
68          pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pPwmRng1Registers);
69          s_pPwmDat1Registers = (volatile unsigned int *)ioremap(PWM_DAT1,
    ↪  sizeof(unsigned int));
70          pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pPwmDat1Registers);
71          s_pPwmClkRegisters = (volatile unsigned int *)ioremap(CLK_BASE,
    ↪  sizeof(unsigned int));
72          pr_alert("map to virtual adresse: 0x%x\n", (unsigned)s_pPwmClkRegisters);
73
74          SetGPIOFunction(s_pGpioRegisters, GPIO12, GPIO_ALTFUNC0);
75
76      ConfigPwmChannel();
77      SetPwmDivisor(BCM2711_PWM_CLOCK_DIVIDER_16); //CLK_DIV
78      ConfigPwmSignal(50, 50);
79
80          printk("init ended");
81
82          return 0;
83  }
84
85  static void __exit pwmModule_exit(void) {
86
87          pr_alert("%s: called\n",__FUNCTION__);
88
89          SetGPIOFunction(s_pGpioRegisters, GPIO12, GPIO_INPUT);
90          *(s_pPwmClkCtlRegisters) = 0x5A000020; /**< Kill the clock*/
91          *(s_pPwmClkDivRegisters) = 0x5A000000;
92          clearbit(*s_pPwmCtlRegisters, PWEN1);        /**< Disable PWM transmission*/
93
94          iounmap(s_pGpioRegisters);
95          iounmap(s_pPwmCtlRegisters);
96          iounmap(s_pPwmStaRegisters);
97          iounmap(s_pPwmRng1Registers);
98          iounmap(s_pPwmDat1Registers);
99          iounmap(s_pPwmClkRegisters);
100         cdev_del(&c_dev);
```

```
101         device_destroy(pwmDevice_class, pwmDevice_majorminor);
102         class_destroy(pwmDevice_class);
103         unregister_chrdev_region(pwmDevice_majorminor, 1);
104 }
105
106 module_init(pwmModule_init);
107 module_exit(pwmModule_exit);
```

utils.h

The *utils.h* header file contains all the definitions and variables necessary for programming this driver, thus keeping the code in a more structured way.

```
1  #define setbit(ptr,pos) ptr |= (1 << pos)
2  #define clearbit(ptr,pos) ptr &= ~(1 << pos)
3  #define togglebit(ptr,pos) ptr^=(1<<pos)
4  #define isbitset(ptr,pos) ((ptr>>pos) & 0x1)
5  #define isbitclear(ptr,pos) !((ptr>>pos) & 0x1)
6
7  #define BCM2711_PERI_BASE   0xfe000000
8  #define GPIO_BASE   (BCM2711_PERI_BASE + 0x200000)
9  #define PWM_BASE    (BCM2711_PERI_BASE + 0x20C000)  /**< PWM0*/
10 #define CLK_BASE    (BCM2711_PERI_BASE + 0x101000)
11 #define BCM2711_PWMCLK_CNTL 40
12 #define BCM2711_PWMCLK_DIV  41
13 #define BCM2711_PWM_CLOCK_DIVIDER_16 16
14
15 /* Defines for PWM Clock, word offsets (ie 4 byte multiples) */
16 #define BCM2711_PWMCLK_CNTL     40
17 #define BCM2711_PWMCLK_DIV      41
18 #define BCM2711_PWM_PASSWRD     (0x5A << 24)
19 #define BCM2711_PWM_CNTL_KILL 5
20 #define BCM2711_PWM_CNTL_ENAB 4
21 #define BCM2711_PWM_CNTL_SRC  0
22 #define BCM2711_PWM_DIV_DIVI  12
23 #define CLK_SRC_OSCILLATOR 1
24
25 #define GPIO12 12
26
27 struct GpioRegisters
28 {
29         uint32_t GPFSEL[6];
30         uint32_t Reserved1;
31         uint32_t GPSET[2];
32         uint32_t Reserved2;
33         uint32_t GPCLR[2];
34 };
35
36 typedef enum {
37
    ↪  GPIO_INPUT=0,
38      GPIO_OUTPUT= 1,
39      GPIO_ALTFUNC5= 2,
```

```
    GPIO_ALTFUNC4= 3,
    GPIO_ALTFUNC0= 4,
    GPIO_ALTFUNC1= 5,
    GPIO_ALTFUNC2= 6,
    GPIO_ALTFUNC3= 7,
} GPIOMODE;


#define PWEN1   0 /**< Channel 1 enable*/
#define MODE1   1 /**< Channel 1 mode*/
#define RPTL1   2 /**< Channel 1 Repeat Last Data*/
#define SBIT1   3 /**< Channel 1 Silence Bit*/
#define POLA1   4 /**< Channel 1 Polarity*/
#define USEF1   5 /**< Channel 1 Use FIFO*/
#define CLRF    6 /**< Clear FIFO*/
#define MSEN1   7 /**< Channel 1 M/S Enable*/

#define STA1 9
```

utils.c

The present *utils.c* has the functions necessary for pwm programming. The *SetPwmDivisor* function, as the name indicates, as the input divider divides the oscillator, the *ConfigPwmChannel* function configures the channel's operating mode, in this case M/S transmission is used and finally the *ConfigPwmSignal* function has as parameters a frequency and a duty cycle and according to these received values, it configures the waveform of the generated signal.

```
struct GpioRegisters *s_pGpioRegisters;
volatile unsigned int * s_pPwmCtlRegisters;
volatile unsigned int * s_pPwmStaRegisters;
volatile unsigned int * s_pPwmRng1Registers;
volatile unsigned int * s_pPwmDat1Registers;
volatile unsigned int * s_pPwmClkRegisters;
volatile unsigned int * s_pPwmClkCtlRegisters;
volatile unsigned int * s_pPwmClkDivRegisters;

volatile unsigned int PWM_CTL = PWM_BASE + 0x00;
volatile unsigned int PWM_STA = PWM_BASE + 0x04;
volatile unsigned int PWM_RNG1 = PWM_BASE + 0x10;
volatile unsigned int PWM_DAT1 = PWM_BASE + 0x14;

void SetPwmDivisor(unsigned int div)
{
        *(s_pPwmClkRegisters + BCM2711_PWMCLK_CNTL) = (BCM2711_PWM_PASSWRD | 1 <<
↪  BCM2711_PWM_CNTL_KILL); /**< Kill the clock*/

        clearbit(*s_pPwmCtlRegisters, PWEN1);        /**< Disable PWM transmission*/

        if ( div < 1 ) {
                div = 1;                                /**< Lowest divisor */
        } else if ( div >= 0x1000 ) {
                div = 0xFFF;                          /**< Highest divisor */
        }
```

```
26
27          *(s_pPwmClkRegisters + BCM2711_PWMCLK_DIV) = (BCM2711_PWM_PASSWRD | (div <<
    ↪  BCM2711_PWM_DIV_DIVI));  /**< Configure the Divisor */
28          *(s_pPwmClkRegisters + BCM2711_PWMCLK_CNTL) = (BCM2711_PWM_PASSWRD | (1 <<
    ↪  BCM2711_PWM_CNTL_ENAB) |
29                  (CLK_SRC_OSCILLATOR << BCM2711_PWM_CNTL_SRC)); /**< bits[3:0]:
    ↪  1-Source oscillator, bit[4]: Enable Clock Generator*/
30          setbit(*s_pPwmCtlRegisters, PWEN1);
31
32          printk("Configure PWM frequency clk");
33  }
34
35  void ConfigPwmChannel(){
36          *s_pPwmCtlRegisters = 0;
37      setbit(*s_pPwmCtlRegisters, MSEN1); // configure as M/S transmission
38      setbit(*s_pPwmCtlRegisters, CLRF);
39          printk("Configure PWM channels");
40  }
41
42  void ConfigPwmSignal(unsigned int freq, unsigned int duty){
43
44          unsigned int s, m;
45
46          if (freq > 20000)        // max 20kHz frequency
47                  freq = 20000;
48          if (freq < 10)
49                  freq = 10;
50
51          s = (54000000 / (freq * 16));  /**< freq = oscillator / (divisor * s); */
52
53          m = s*duty/100;                /**< m/s = duty cycle */
54
55          clearbit(*s_pPwmCtlRegisters, PWEN1); /**< Disable PWM transmission*/
56          *(s_pPwmRng1Registers) = s;
57          *(s_pPwmDat1Registers) = m;
58          if(*(s_pPwmDat1Registers + STA1) == 0){
59                  printk("channel1 Disable");
60          }
61          setbit(*s_pPwmCtlRegisters, PWEN1);  /**< Enable PWM transmission*/
62          printk("Configure PWM Signal form (m/n)");
63  }
```

20 | Daemon

All database management is executed in the daemon, and all the requests from the main system are performed via message queue and then read and interpreted. Below is the code of the daemon creation.

```
int createDaemon() {
    pid_t pid;  // process ID
    pid_t sid;  // session ID

    openlog("AqCtrlDaemon", LOG_PID, LOG_DAEMON);   // initialize syslog

    pid = fork();           // create a new process

    if (pid < 0){
        syslog(LOG_ERR, "fork Failed");
        exit(EXIT_FAILURE);
    }

    if (pid > 0)    // exit the parent process
        exit(EXIT_SUCCESS);

    sid = setsid();             // create a new session

    if (sid < 0){
        syslog(LOG_ERR, "setsid failed");
        exit(EXIT_FAILURE);
    }

    umask(0);   // set full permissions over newly created files

    // close the standard streams file descriptors and detach from the terminal
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);


    if (chdir("/") < 0){    // change to the root directory
        syslog(LOG_ERR, "chdir failed");
        exit(EXIT_FAILURE);
    }

    // START DAEMON CODE HERE ------------------------------
    syslog(LOG_INFO, "Daemon Created Successfully");
    daemonImp();
    // END DAEMON CODE HERE --------------------------------

    // close resources, cleanup and exit
    closelog();
    exit(EXIT_SUCCESS);
}
```

The *daemonImp* function is responsible for continuously reading the message queue "/mqueueToDaemon" since all requests to daemon are all done writting to this message queue and then parsed in the *HandleMQueue* function. In the beginning of the function all message queues used by the whole system are created.

```
/**
 * @brief main daemon function
 *
 * create all message queues for the program
 * reads the mq "/mqueueToDaemon" infinitey and handles its content
 */
void daemonImp() {
    mqInitALL();

    while(true) {
        char *msgcontent = readMsgQueue("/mqueueToDaemon");
        if (msgcontent != NULL) {
            HandleMQueue(msgcontent);
            syslog(LOG_INFO, "READ: %s", msgcontent);
        }
        else
            syslog(LOG_INFO, "NOT READ");
    }
}
```

The *HandleMQueue* analyzes the first word of the message queue which is the main command and performs corresponding actions.

```
/**
 * @brief handles messages from the message queue and performs corresponding actions.
 *
 * @param content the content of the message from the message queue.
 */
void HandleMQueue (char* content) {
    Database db;
    char* c_command = strtok(content, " ");
    std::string command = c_command;

    if (command == "ALLFISH") {
        std::vector<std::string> allSpeciesList = db.getSpeciesList("fishList");
        sendStringVector(allSpeciesList);
        return;
    }
    else if (command == "AQUARIUM") {
        std::vector<std::string> aqSpeciesList = db.getSpeciesList("aquariumFish");
        sendStringVector(aqSpeciesList);
        std::vector<std::string> aqSpeciesQtty = db.getAqQuantity();
        sendStringVector(aqSpeciesQtty);
        return;
    }
    else if (command == "INFO") {
        char* c_fishName = strtok (nullptr, "\0");
        std::string strFishName = c_fishName;
        Info(strFishName);
        return;
```

```
28      }
29      else if (command == "IDEAL") {
30          retrieveIdealCond();
31          return;
32      }
33      else if (command == "ADD") {
34          char* c_fishName = strtok (nullptr, "\0");
35          std::string strFishName = c_fishName;
36          db.add(strFishName);
37          return;
38      }
39      else if (command == "REMOVE") {
40          char* c_fishName = strtok (nullptr, "\0");
41          std::string strFishName = c_fishName;
42          db.remove(strFishName);
43          return;
44      }
45      else if (command == "DATA") {
46          char* c_sensor = strtok (nullptr, "\0");
47          std::string strSensor = c_sensor;
48          std::vector<std::string> sensorData = db.getSensorReadings(strSensor);
49          sendStringVector(sensorData);
50          return;
51      }
52      else if (command == "SAVEDATA") {
53          char* c_ph = strtok (nullptr, " ");
54          char* c_temp = strtok (nullptr, " ");
55          char* c_tds = strtok (nullptr, "\0");
56          float f_ph = std::strtof(c_ph, nullptr);
57          float f_temp = std::strtof(c_temp, nullptr);
58          float f_tds = std::strtof(c_tds, nullptr);
59          db.insertSensorData(f_ph, f_temp, f_tds);
60          return;
61      }
62      else {
63          syslog(LOG_INFO, "read unknow mq: %s", command.c_str());
64          return;
65      }
66  }
```

Beneath it's exhibited the header file of the database. The database was encapsulated in a class so other types of databases can be used besides SQLite.

```
1   /**
2    * @brief Represents parameters associated with a fish
3    */
4   struct FishParameters {
5       float food;         ///< Quantity of food suitable for the fish
6       float temperature;  ///< Water temperature suitable for the fish
7       float ph;           ///< pH level suitable for the fish
8   };
9
10  /**
11   * @brief Represents the ideal value of each measure for the aquarium
```

```
12  */
13  struct structIdealConditions {
14      float food;         ///< Ideal Quantity of food for all the fish in the Aquarium
15      float temperature;  ///< Ideal Temperature for the fish in the Aquarium
16      float ph;           ///< Ideal ph for the fish in the Aquarium
17  };
18
19  class Database
20  {
21  private:
22      sqlite3 *db;
23      sqlite3_stmt* stmt;
24      const std::string dbPath = "/etc/Aquarium/aqdb.db";
25  public:
26      Database();
27      ~Database();
28
29      /**
30       * @brief Opens a connection to the database
31       * @return A pointer to the SQLite connection if successful, nullptr on error
32       */
33      sqlite3* openDatabase();
34
35      /**
36       * @brief Closes the connection to the database
37       * @param db A pointer to the SQLite connection to be closed
38       */
39      void closeDatabase();
40
41      /**
42       * @brief Updates the quantity of a fish species in the aquarium database
43       *
44       * This function increases the quantity of the specified fish species in the
    ↪  aquariumFish table by 1
45       *
46       * @param species The name of the fish species
47       * @param db The SQLite database connection
48       * @return 0 on success, 1 on failure
49       */
50
51      int updateFishQuantity(const std::string& species);
52
53      /**
54       * @brief Inserts a new fish species into the aquarium database if it doesn't
    ↪  exist
55       * @param species The name of the fish species
56       * @param db The SQLite database connection
57       * @return 0 on success, 1 on failure
58       */
59      int insertFishIfNotExists(const std::string& species);
60
61      /**
62       * @brief Executes an SQLite query
63       * @param db A pointer to the SQLite connection
64       * @param sql The SQL query to be executed
```

```
65          * @param stmt A pointer to the prepared SQLite statement
66          * @return 0 if the execution is successful, 1 on error
67          */
68         int executeQuery(const std::string& sql);
69
70         /**
71          * @brief Adds a new fish species to the database or increments its quantity if
    ↪  already exists
72          * @param species Name of the fish species to add
73          * @return 0 if successful, 1 if an error occurs
74          */
75         int add(const std::string& species);
76
77         /**
78          * @brief Decrements the quantity of a fish species in the database
79          * If the quantity becomes zero, the species is removed from the database
80          * @param species Name of the fish species to remove or decrement
81          * @return 0 if successful, 1 if an error occurs
82          */
83         int remove(const std::string& species);
84
85         /**
86          * @brief Retrieves a list of species from the specified table in the database
87          * @param tableName Name of the table containing species information
88          * @return A vector of strings containing the species names If an error occurs,
    ↪  the vector returns with the word "EMPTY"
89          */
90         std::vector<std::string> getSpeciesList(const std::string& tableName);
91
92         /**
93          * @brief Retrieves the quantity of each fish species in the aquarium
94          * @return A vector of strings containing the quantity of each species If an error
    ↪  occurs, the vector returns with "0"
95          */
96         std::vector<std::string> getAqQuantity();
97
98         /**
99          * @brief Retrieves fish parameters for a specific species from the database
100         * This function gets a species name as input and queries the database
101         * to retrieve the associated temperature, pH, and food parameters
102         * @param species The name of the fish species
103         * @return FishParameters structure containing temperature, pH, and food
    ↪  parameters
104         */
105        FishParameters getFishParameters(const std::string& species);
106
107        /**
108         * @brief Retrieves the ideal conditions for all fish in the aquarium
109         * @return structIdealConditions structure containing the ideal quantity of food,
    ↪  temperature, and pH
110         */
111        structIdealConditions retrieveFishIdealCond();
112
113        /**
114         * @brief Callback function used in SQL queries
```

```
115         * @param data User data passed to the callback
116         * @param argc Number of columns in the result set
117         * @param argv Array of strings representing column values
118         * @param colNames Array of strings representing column names
119         * @return 0 to continue processing rows
120         */
121        static int callback(void *data, int argc, char **argv, char **colNames);
122
123        /**
124         * @brief Inserts sensor data (PH, Temp, TDS) into the database
125         * @param ph PH value
126         * @param temp Temperature value
127         * @param tds TDS (Total Dissolved Solids) value
128         */
129        void insertSensorData(float ph, float temp, float tds);
130
131        /**
132         * @brief Retrieves sensor readings from the database based on the sensor type
133         * @param sensorType Type of sensor for which readings are retrieved ("PH",
    ↪  "Temp", "TDS")
134         * @return A vector of strings containing the sensor requested readings
135         */
136        std::vector<std::string> getSensorReadings(const std::string& sensorType);
137
138        /**
139         * @brief Removes old sensor data from the database
140         * Old data is defined as data recorded more than 72 hours ago
141         */
142        void removeOldData();
143    };
```

21 | Classes Implementation

This project was developed using the C++ programming language to utilize object-oriented programming, as it was a project constrain. Consequently, the implementation includes 12 classes, each serving a specific purpose.

21.1 MainClass

The Main Class is the class that stands out the most because it contains most of the objects from the other classes present in the project. This also has functions for creating mutexes, condition variables, threads and thread functions, thus being the class responsible for all multithreading of the project.

This class inherits the QObject class, a class that is important in the interaction between C++ and QML, and as you can see in the code below, there are Q_INVOKABLE functions necessary for use in QML.

```
class MainClass : public QObject  {
    Q_OBJECT
public:
    MainClass();
    ~MainClass();
    void RunAC();
    void createMutexes();
    void destroyMutexes();
    void createConds();
    void destroyConds();
    void attrThreads();
    void createThreads();
    void joinThreads();

public /*slots*/:
    Q_INVOKABLE void requestReading();
    Q_INVOKABLE QStringList requestValues();
    Q_INVOKABLE int verifyError();
    Q_INVOKABLE void shutdown();

private:
    /*
     * @brief Threads Functions
    */
    static void *tUVLightFunc(void *);
    static void *tSigFunc(void *);
    static void *tReadSensorsFunc(void *);
    static void *tWaterHeaterFunc(void *);
    static void *tServoMotorFunc(void *);
    static void *tIdleFunc(void *);
    static void* tInterfaceFunc(void*);

    static void handleSig(int signal);
    /**
     * @brief Get the system current time
     * @return The system current time in minutes
     */
    static unsigned int getSystemTime();

    /**
     * @brief Receives an array of values, order them in ascending order and returns
↪   the middle value
```

```
42          * @param sensorArray An array with the sensors reading values
43          * @param size Size of the array
44          * @return The best value
45          */
46         static float getBestValue(float sensorArray[], int size);
47
48     private:
49         IdealConditions& idealCond = IdealConditions::getInstance();
50
51         GPIOActuator heater{23};
52         GPIOActuator light{24};
53         ServoActuator servo;
54         TDS tds;
55         PH ph;
56         Temp temp{"28-3ce1e3806c3f"};
57
58         pthread_t tUVLight, tSig, tReadSensors, tWaterHeater, tServoMotor, tIdle,
    ↪  tInterface;
59         pthread_attr_t attr0, attr1, attr2, attr3;
60         sched_param sched0, sched1, sched2, sched3;
61         pthread_mutex_t mutexRS, mutexMotor, mutexWH, mutexLight, mutexTime;
62         pthread_cond_t condRS, condMotor, condWH, condLight, condTime;
63
64         unsigned int sensorsReadTime;       /**< Time between sensors reading */
65         static unsigned int FlagSIGALRM;    /**< Flag to signal that the timer ended */
66
67         float realValueTds;
68         float realValuePh;
69         float realValueTemp;
70
71         bool tds_error; /**< True if the tds value is unusual, false otherwise */
72         bool ph_error;  /**< True if the ph value is unusual, false otherwise */
73     };
```

21.2 Sensors

The Sensors class was implemented as the parent class of the *Temp*, *TDS* and *PH* classes that will inherit it. This class is composed of an *ADS1115* class and has all the variables and functions necessary for its use.

It presents set and get functions widely used in object-oriented programming, it also has a virtual function *readSensor* that the implementation in this class will be used for readings from sensors with i2c bus while *Temp* class, which uses one wire, will be implemented differently. It should be noted that there is a purely virtual function *convertSensor* so the subclasses will have to implement it and it boils down to converting the value received by the sensor to a real measurement value.

```
1  class Sensors : public QObject  {
2      Q_OBJECT
3  protected:
4      float voltagevalue;  /**< The voltage value read from the sensor */
5      float realValue;     /**< The real value converted from the raw sensor reading */
6      int i2cChannel;      /**< The I2C channel used for communication */
7      ADS1115 ads1115;     /**< The ADS1115 object for analog-to-digital conversion */
```

```
8
9   public /*slots*/:
10      /**
11       * @brief Gets the value of the sensor.
12       * @return The real value (value already converted) of the sensor.
13       */
14      Q_INVOKABLE float getRealValue();
15
16  public:
17      Sensors();
18      ~Sensors();
19
20      /**
21       * @brief Reads the sensor value
22       * @return True if the sensor reading was successful, false otherwise
23       */
24      virtual bool readSensor();
25
26      /**
27       * @brief Sets the voltage value of the sensor
28       * @param voltagevalue The voltage value to set
29       */
30      void setVoltageValue(float voltagevalue);
31
32      /**
33       * @brief Sets the real value of the sensor
34       * @param realValue The real value to set
35       */
36      void setRealValue(float realValue);
37
38      /**
39       * @brief Gets the voltage value of the sensor
40       * @return The voltage value of the sensor
41       */
42      float getVoltageValue();
43
44      /**
45       * @brief Converts the sensor reading to a real value
46       * @param rawValue The sensor reading value
47       * @return The converted value (real value)
48       */
49      virtual float convertValue(float rawValue) = 0;
50  };
```

21.3 ServoActuator

The ServoActuator class, as the name suggests, has the variables and functions to control the servo motor present in the project. This class has only one variable *dayTime* whose value represents the time of day when the fish in the aquarium were last fed, and can therefore take on the value of "morning" or "night" (enumerations).

The class includes set and get functions for a single variable named *dayTime*. Additionally, there is a *feed* function that, based on the provided food amount parameter, controls a servo motor to rotate as many times as needed to feed the fish in the aquarium. The class also defines two constants, *closeS* and *openS*,

representing duty cycle values required to operate the servo motor.

```
#define openS 7
#define closeS 9

enum dayTimeS {morning, night};

class ServoActuator {

private:
    dayTimeS dayTime; /**< Current time of the day */

public:
    ServoActuator();
    ~ServoActuator();
    /**
     * @brief Feed the actuator with a specified quantity of food
     * @param foodQuantity The quantity of food to feed
     * @return True if was successful, false otherwise
     */
    bool feed(int foodQuantity);

    /**
     * @brief Set the time of the day that fed occurred (morning / night)
     * @param dayTime The time of the day to set
     */
    void setDayTime(dayTimeS dayTime);

    /**
     * @brief Get the time of day when feeding has already occurred
     * @return The time of the day that feeding has occurred
     */
    dayTimeS getDayTime();
};
```

21.4 GPIOActuator

The GPIOActuator class was created in order to facilitate the control of gpio output pins, and in this case it will be used to control the ultraviolet light and the water heater. The class has two variables, a *state* that represents the current state of the actuator, which can be "on" or "off" and a variable *gpioPin* that defines the raspberry pin of each actuator. The constructor of this class has as a parameter the gpio pin intended for the created object and in terms of the class's functions, there is a set and a get for the state and a function *control* which depending on the parameter passed is responsible for calling or turn off the actuator in question.

```
enum OnOff {on, off};

class GPIOActuator {

private:
    OnOff state;        /**< Current state of the actuator */
    OnOff operation;    /**< Operation to be performed on the actuator*/
    uint8_t gpioPin;    /**< GPIO pin associated with the actuator */
```

```
9
10  public:
11      GPIOActuator(uint8_t gpioPin);
12      ~GPIOActuator();
13      /**
14       * @brief Controls the actuator based on the operation.
15       * @param operation The operation to be performed (on / off)
16       * @return True if the operation was successful, false otherwise
17       */
18      bool control(OnOff operation);
19
20      /**
21       * @brief Gets the current state of the actuator
22       * @return The current state
23       */
24      OnOff getState();
25
26      /**
27       * @brief Sets the state of the actuator.
28       * @param state The new state
29       */
30      void setState(OnOff state);
31  };
```

21.5 Temp, TDS & PH

As previously mentioned, the Temp, PH and TDS classes inherit the Sensors class.

The Temp class has a constructor where the sensor's one wire address is passed, which is concatenated with the path for communication with this same protocol and stored in the private variable *address*. Unlike the PH and TDS classes, this one has a *readSensor* implementation different from the one in the Sensors class, since the one wire protocol is different from the i2c, and presents the *convertValue* function that converts the received value to the real temperature value.

```
1   class Temp : public Sensors
2   {
3   private:
4       std::string address;
5
6   public:
7
8       Temp(std::string address);
9       ~Temp();
10      bool readSensor();
11      float convertValue(float rawValue);
12  };
```

In relation to the PH and TDS classes, they are similar, just differentiating the *convertValue* function implemented in each of them differently.

```
1   class TDS : public Sensors
2   {
3   public:
```

```
    TDS();
    ~TDS();

    float convertValue(float rawValue);
};

class PH : public Sensors
{
public:
    PH();
    ~PH();

    float convertValue(float rawValue);
};
```

21.6 IdealConditions

The IdealConditions class was implemented following the singleton design pattern, as there should be only one instance of this class representing ideal conditions in the project.

This class has all the variables relating to ideal conditions, such as ideal sensor values, feeding time, UV light turn on time and amount of food. Most of the class functions are sets and gets of private variables, and Q_INVOKABLE is used here so that they can also be used in QML for the graphical interface.

```
class IdealConditions : public QObject {
    Q_OBJECT
public:
    static IdealConditions& getInstance();  // singleton design pattern method
    ~IdealConditions();

    void updtNewFishParam(std::vector<std::string>);

    void setTDS(float);
    void setTemp(float);
    void setPh(float);
    float getLightDuration();

    Q_INVOKABLE void calculateIdeal();
    Q_INVOKABLE bool verifyAddition();
    Q_INVOKABLE float getFood();
    Q_INVOKABLE float getTDS();
    Q_INVOKABLE float getTemp();
    Q_INVOKABLE float getPh();
    Q_INVOKABLE float getLightTime();
    Q_INVOKABLE float getFeedTime(bool);

private:
    IdealConditions();
    IdealConditions(const IdealConditions&);                // Private copy
    ↪ constructor to prevent copying
    IdealConditions& operator=(const IdealConditions&);     // Private copy assignment
    ↪ operator to prevent assignment

    float tds;
```

```
29      float temp;
30      float ph;
31      float lightTime;
32      float feedTime[2];
33      float foodQuantity;
34      float lightTimeDuration;
35      float fishParams[3];
36
37      std::string mqNameR = "/mqueueDaemonToIdeal";
38      std::string mqNameW = "/mqueueToDaemon";
39  };
```

22 | Threads Implementation

In response to the need for simultaneous task execution, multithreading was implemented. To prevent potential race conditions from simultaneous thread access, parallelism mechanisms such as mutexes and condition variables were introduced.

Additionally, thread priorities were established to manage the sequence of execution, enhancing the efficiency of parallelized tasks. This approach not only facilitated concurrent task execution but also reduce the risk of conflicts, ensuring the development of a robust and well-coordinated multithreaded system.

22.1 tIdle

The tIdle thread is responsible for periodically sending a *condTime* signal, which activates the *tSig* thread. Its crucial function involves the use of a timer programmed with the *alarm* function. When the timer expires, it generates a *SIGALRM* signal, making the *tIdle* thread to send the *condTime* signal and reprogram the timer for subsequent intervals.

```
1  void *MainClass::tIdleFunc(void *aquarium) {
2      MainClass *aq = static_cast<MainClass*>(aquarium);
3
4      aq->FlagSIGALRM = 0;
5      aq->sensorsReadTime = 5*60;
6
7      signal(SIGALRM, handleSig);
8      alarm(1);
9
10     while (1) {
11         if (aq->FlagSIGALRM) {
12             pthread_mutex_lock(&aq->mutexTime);
13             pthread_cond_signal(&aq->condTime);
14             pthread_mutex_unlock(&aq->mutexTime);
15             alarm(aq->sensorsReadTime);
16             aq->FlagSIGALRM = 0;
17         }
18     }
19     pthread_exit(NULL);
20 }
```

22.2 tSig

Regarding the initial configuration of the *tSig* thread, it uses the *getSystemTime* function to receive the current system time, and depending on the fish feeding schedule, the *dayTime* variable is set to either "morning" or "night."

During the thread cycle execution, it awaits the arrival of *condTime* and upon receiving it, the system reaffirms the current time and, considering the conditions presented in the IdealConditions class and the states of the actuators decides whether signals are sent or not. If it is time to activate the fish feeder, a *condMotor* signal will be sent to the *tServoMotor*. On the other hand, if it is the designated time to turn on the UV light, a *condLight* signal will be dispatched to the *tUVLight* thread.

```
1  void *MainClass::tSigFunc(void *aquarium) {
2      MainClass * aq = static_cast<MainClass*>(aquarium);
3      unsigned int timeInMinutes = getSystemTime();
4      float feedTime [2];
5      feedTime[morning] = aq->idealCond.getFeedTime(morning);      // [0]:morning
```

```
6       feedTime[night] = aq->idealCond.getFeedTime(night);         // [1]:night
7
8       if (timeInMinutes > feedTime[morning]) {
9           aq->servo.setDayTime(morning);
10      } else {
11          aq->servo.setDayTime(night);
12      }
13
14      while(1) {
15          pthread_mutex_lock(&aq->mutexTime);
16          pthread_cond_wait(&aq->condTime, &aq->mutexTime);
17          pthread_mutex_unlock(&aq->mutexTime);
18
19          timeInMinutes = getSystemTime();
20
21          feedTime[morning] = aq->idealCond.getFeedTime(morning); // [0]:morning
22          feedTime[night] = aq->idealCond.getFeedTime(night);     // [1]:night
23
24          if((feedTime[morning] <= timeInMinutes && timeInMinutes <= feedTime[morning] +
    ↪  60   // check if it is time to feed
25              && aq->servo.getDayTime() == night) || (feedTime[night] <= timeInMinutes
26                 && timeInMinutes <= feedTime[night] + 60 && aq->servo.getDayTime() ==
    ↪  morning)) {
27              pthread_cond_signal(&aq->condMotor);    //condition to feed the tank
28          }
29          if (aq->idealCond.getLightTime() <= timeInMinutes && timeInMinutes <=
    ↪  (aq->idealCond.getLightTime() + 60) // // check time to activate the light
30              && aq->light.getState() == off) {
31              pthread_cond_signal(&aq->condLight);    //condition to activate UVlight
32          }
33          pthread_cond_signal(&aq->condRS);   //condition to read sensors values
34      }
35      pthread_exit(NULL);
36  }
```

22.3 tReadSensors

The *tReadSensors* thread assumes the responsibility of reading all sensors and taking appropriate actions based on these readings. Throughout its execution, it patiently awaits a *condRS* signal, sent by *tSig*. Upon receiving this signal, the thread proceeds to perform five readings of the sensors, selecting the most optimal measurement among them. These readings are then converted to real values and transmitted via a message queue to the daemon, which subsequently stores them in the database.

Following the acquisition of the best sensor readings, the values for TDS and pH are compared against their ideal values. If they surpass the desired thresholds, the variables *ph_error* and *tds_error* are adjusted to signal a warning to the user through the display.

The temperature values are also evaluated, and if the temperature is low, the thread initiates adjustments in the timer interval used by *tIdle* for more frequent measurements. Subsequently, a *condWH* signal is dispatched to activate the water heater and raise the water temperature. On the other hand, if the temperature exceeds the desired value, a condWH signal is sent, but with the heater set to the off state, effectively employing an on-off controller for the heater.

```
void *MainClass::tReadSensorsFunc(void *aquarium) {
    MainClass * aq = static_cast<MainClass*>(aquarium);

    const int readingBuffer = 5;
    float tdsReading[readingBuffer];
    float phReading[readingBuffer];
    float tempReading[readingBuffer];
    float idealTds, idealPh, idealTemp;
    OnOff heaterState;
    MQueueHandler SaveSensorData("/mqueueToDaemon");


    while(1) {
        pthread_mutex_lock(&aq->mutexRS);
        pthread_cond_wait(&aq->condRS, &aq->mutexRS);
        pthread_mutex_unlock(&aq->mutexRS);

        idealTds = aq->idealCond.getTDS();
        idealPh = aq->idealCond.getPh();
        idealTemp =  aq->idealCond.getTemp();
        heaterState = aq->heater.getState();


        int i = 0;
        for (i = 0; i < readingBuffer; ++i) { // read all the sensors the
↪   readingBuffer times

            if(!aq->tds.readSensor())
                std::cout << "Error Reading TDS sensor " << std::endl;
            if(!aq->ph.readSensor())
                std::cout << "Error Reading PH sensor " << std::endl;
            if(!aq->temp.readSensor())
                std::cout << "Error Reading Temp sensor " << std::endl;

            tdsReading[i]   = aq->tds.getVoltageValue();    // store the value of
↪   sensor readings
            phReading[i]    = aq->ph.getVoltageValue();     // in an array
            tempReading[i]  = aq->temp.getVoltageValue();
        }

        aq->realValuePh   = aq->ph.convertValue(getBestValue(phReading, i));    // get
↪   the best value of the array
        aq->realValueTemp = aq->temp.convertValue(getBestValue(tempReading, i));
        aq->realValueTds  = aq->tds.convertValue(getBestValue(tdsReading, i));

        std::string sensorData = "SAVEDATA " + std::to_string(aq->realValuePh)
                                 + " " + std::to_string(aq->realValueTemp)
                                 + " " + std::to_string(aq->realValueTds);
        SaveSensorData.saveData(sensorData); // produce a msqueue with the sensors
↪   values to store in the database

        if (aq->realValueTds > idealTds + thresholdTDS) { // tds exceeded acceptable
↪   limits
            aq->tds_error = true; // variable to produce a tds warning in the display
        }
```

```
51          else {
52              aq->tds_error = false;
53          }
54          if (aq->realValuePh > idealPh + thresholdPH || aq->realValuePh < idealPh -
    ↪  thresholdPH){ // pH exceeded acceptable limits
55              aq->ph_error = true;    // variable to produce a ph warning in the display
56          }
57          else {
58              aq->ph_error = false;
59          }
60          if (aq->realValueTemp > idealTemp + thresholdTemp && heaterState == on) {   //
    ↪  the temperature is higher than it should so the heater
61                                                                                      //
    ↪  must be turned off
62              aq->heater.setState(off);
63              pthread_mutex_lock(&aq->mutexWH);
64              pthread_cond_signal(&aq->condWH);
65              pthread_mutex_unlock(&aq->mutexWH);
66          }
67
68          if (aq->realValueTemp < idealTemp - thresholdTemp) {    // water is to cold
    ↪  according to the ideal Conditions
69
70              alarm(0);
71              alarm(30);  // change the timer to perform fast readings because water
    ↪  will be warming up
72
73              if (heaterState != on) {
74                  aq->heater.setState(on);
75                  pthread_mutex_lock(&aq->mutexWH);
76                  pthread_cond_signal(&aq->condWH);
77                  pthread_mutex_unlock(&aq->mutexWH);
78              }
79          }
80      }
81      pthread_exit(NULL);
82  }
```

22.4 tServoMotor

Regarding *tServoMotor*, it awaits a condMotor signal, which can be activated by *tSig*. Upon receiving this signal, a feed function is triggered with the required amount of food for the fish in the aquarium. This function operates the motor as many times as necessary to dispense the specified amount of food.

```
1
2   void *MainClass::tServoMotorFunc(void *aquarium) {
3       MainClass * aq = static_cast<MainClass*>(aquarium);
4
5       while(1) {
6           pthread_mutex_lock(&aq->mutexMotor);
7           pthread_cond_wait(&aq->condMotor, &aq->mutexMotor);
8           pthread_mutex_unlock(&aq->mutexMotor);
9
```

```
10          float foodQuantity = aq->idealCond.getFood();
11
12          aq->servo.feed(foodQuantity);   // rotates the motor the times of food needed
13
14      }
15      pthread_exit(NULL);
16  }
```

22.5 tUVLight

The *tUVLight* thread, like the ones previously mentioned, waits for a conLight, generated by the *tSig* thread in case it is time to turn on the light. After receiving the *condLight*, the time to keep the light on is stored in the *lightDuration* variable and a timer made with the *sleep* function and a *count* variable starts counting the time. When the time is equal to the *ligh Duration*, it turns off, putting the thread on hold again for the *condLight*.

```
1
2   void *MainClass::tUVLightFunc(void *aquarium) {
3       MainClass * aq = static_cast<MainClass*>(aquarium);
4
5       aq->light.control(off);
6       unsigned int lightDuration = aq->idealCond.getLightDuration();
7       bool flagUv = 0;
8       volatile int count = 0;
9       while(1) {
10
11          if(!flagUv){
12              pthread_mutex_lock(&aq->mutexLight);
13              pthread_cond_wait(&aq->condLight, &aq->mutexLight); //condition to
    ↪   activate the light
14              pthread_mutex_unlock(&aq->mutexLight);
15              flagUv = 1;
16              aq->light.control(on);
17              lightDuration = aq->idealCond.getLightDuration();
18          }
19          sleep(1);
20          count++;
21          if(count >= lightDuration*60){          // When the time for having the light
    ↪   on ends,
22              aq->light.control(off);             // this condition is valid
23              count = 0;
24              flagUv = 0;
25          }
26      }
27      pthread_exit(NULL);
28  }
```

22.6 tWaterHeater

The *tWaterHeater* thread just waits for the *condWH*, which is sent by *tReadSensors* and the control function is used with the state value, which before being sent the *condWH* signal was changed. This makes this thread turn on or off the aquarium's water heater.

```
1  void *MainClass::tWaterHeaterFunc(void *aquarium) {
2      MainClass * aq = static_cast<MainClass*>(aquarium);
3
4      aq->heater.control(off);
5      while(1) {
6          pthread_mutex_lock(&aq->mutexWH);
7          pthread_cond_wait(&aq->condWH, &aq->mutexWH);
8          pthread_mutex_unlock(&aq->mutexWH);
9
10         aq->heater.control(aq->heater.getState());  //according to the state, turn on
   ↪  or off the heater
11
12     }
13     pthread_exit(NULL);
14 }
15
```

22.7 tInterface

The interface of the application was also included in a thread called *tInterface* since it communicates with other threads of the system besides the daemon.

After configuring the Qt application, creating an engine to process the QML code and registering objects in the QML context to allow communication between the C++ code and the QML code, the code in the line 34 start the loop of events, waiting for screen touches and signals and keeping the application active.

```
1  void* MainClass::tInterfaceFunc(void* aquariumQt) {
2      MainClass *aqQt = static_cast<MainClass*>(aquariumQt);
3
4      QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
5
6      int a = 1;
7      QApplication app(a, nullptr);
8      QQmlApplicationEngine engine;
9
10     const QUrl url_1(QStringLiteral("qrc:/main.qml"));
11
12     // connect app with engine in a synchronous way
13     QObject::connect(&engine, &QQmlApplicationEngine::objectCreated, &app,
   ↪  [url_1](QObject *obj, const QUrl &objUrl) {
14             if (!obj && url_1 == objUrl)
15                 QCoreApplication::exit(-1);
16         }, Qt::QueuedConnection);
17
18     MQueueHandler SendToDaemon("/mqueueToDaemon");
19     MQueueHandler ReadFromDaemon("/mqueueDaemonToDisplay");
20     dataList all;
21     dataList aquarium;
22     IdealConditions& Ideal = IdealConditions::getInstance();
23
24     engine.rootContext()->setContextProperty("all", &all);
25     engine.rootContext()->setContextProperty("aquarium", &aquarium);
26     engine.rootContext()->setContextProperty("SendToDaemon", &SendToDaemon);
27     engine.rootContext()->setContextProperty("ReadFromDaemon", &ReadFromDaemon);
```

```
28      engine.rootContext()->setContextProperty("Ideal", &Ideal);
29      engine.rootContext()->setContextProperty("aq", QVariant::fromValue (aqQt));
30
31      engine.load(url_1);
32      app.exec();
33
34      pthread_exit(NULL);
35      return nullptr;
36  }
```

23 | Front-end Implementation

As previously stated, the front-end application is developed in Qt using QML language.

One of the most important QML types used is the StackView, which implemented a stack-based navigation mdel, allowing multiple pages to be connected. Below is the code responsible to change the exhibited page in the screen.

```
1       StackView {
2           id: stack
3           anchors.fill: parent
4           initialItem: page_main
5       }
6
7       Component {
8           id: page_main
9           Page_main{}
10      }
11      Component {
12          id: page_options
13          Page_options{}
14      }
15      Component {
16          id: page_addfish
17          Page_addfish{}
18      }
19      Component {
20          id: page_addfish_conf
21          Page_addfish_conf{}
22      }
23      Component {
24          id: page_removefish
25          Page_removefish{}
26      }
27      Component {
28          id: page_onoff
29          Page_onoff{}
30      }
31      Component {
32          id: page_read
33          Page_read{}
34      }
35      Component {
36          id: page_added
37          Page_added{}
38      }
39      Component {
40          id: page_removed
41          Page_removed{}
42      }
43      Component {
44          id: page_ideal
45          Page_ideal{}
46      }
47      Component {
48          id: page_graph
49          Page_graph{}
50      }
```

```
51
52      function changePage(page) {
53          switch (page) {
54          case "options":
55              stack.push(page_options)
56              break
57          case "addfish":
58              stack.push(page_addfish)
59              break
60          case "addfishconf":
61              stack.push(page_addfish_conf)
62              break
63          case "removefish":
64              stack.push(page_removefish)
65              break
66          case "onoff":
67              stack.push(page_onoff)
68              break
69          case "read":
70              stack.push(page_read)
71              break
72          case "addSuccess":
73              stack.push(page_added)
74              break
75          case "removeSuccess":
76              stack.push(page_removed)
77              break
78          case "ideal":
79              stack.push(page_ideal)
80              break
81          case "graph":
82              stack.push(page_graph)
83              break;
84          case "back":
85              stack.pop()
86              break
87          case "addDone":
88              stack.pop()
89              stack.pop()
90              stack.pop()
91              stack.pop()
92              break
93          case "removeDone":
94              stack.pop()
95              stack.pop()
96              stack.pop()
97              break
98          }
99      }
```

With the intent of displaying the fish species in the aquarium or all the different fish species in the whole database, the ListView QML type was extremely useful used along an object of the class *dataList*. Below is a passage of the code used to display all the fish species in the screen, divided into pages.

```
ListView {
    id: list
    x:150; y:90
    width: 500
    height: 310
    interactive: false // disable scrolling

    property int selectedCOLOR : -1
    property int selectedID

    model: ListModel {
        id: allModel
        ListElement { data: "" }

        Component.onCompleted: {
            SendToDaemon.requestMQ("ALLFISH")
            var allFishQList = ReadFromDaemon.getQStrList()
            all.setDataList(allFishQList)
            allModel.remove(0)
            var currData = all.getCurrentData();
            for (var i = 0; i < currData.length; ++i)
                allModel.append({ "data": currData[i] })
        }
    }
    delegate: Item {
        width: list.width
        height: 44

        Rectangle {
            border.color: "white"
            width: parent.width
            height: parent.height
            color: index === list.selectedCOLOR ? "light blue" : "transparent"

            Text {
                x: 15
                anchors.centerIn: parent
                font.pixelSize: 20
                color: "white"
                text: model.data
            }

            MouseArea {
                anchors.fill: parent
                onClicked: {
                    list.selectedCOLOR = index
                    confirm_button.visible = true
                    list.selectedID=index+(all.getCurrentIndex()*all.getChunkSize());
                    all.setNameId(model.data, list.selectedID)
                }
            }
        }
    }
}
```

Cooperatively with the ListView, buttons to change the chunks/pages of species are required. Instead of buttons, small images were used with the MouseArea QML type, so they can be ”clicked” and a corresponding action is executed. The code of the two buttons previously mentioned is in the following code snippet.

```
Image {
    id:up_arrow
    source: "images/up_arrow.png"
    x: 40; y: 185
    width: 70
    height: 70
    fillMode: Image.PreserveAspectCrop
    MouseArea {
        anchors.fill: parent
        onClicked: {
            list.selectedCOLOR = -1
            confirm_button.visible = false
            all.showPreviousData()
            allModel.clear()
            var currData = all.getCurrentData();
            for (var i = 0; i < currData.length; ++i) {
                allModel.append({ "data": currData[i] })
            }
        }
    }
}

Image {
    id:down_arrow
    source: "images/down_arrow.png"
    x: 40; y: 275
    width: 70
    height: 70
    fillMode: Image.PreserveAspectCrop
    MouseArea {
        anchors.fill: parent
        onClicked: {
            list.selectedCOLOR = -1
            confirm_button.visible = false
            all.showNextData()
            allModel.clear()
            var currData = all.getCurrentData();
            for (var i = 0; i < currData.length; ++i) {
                allModel.append({ "data": currData[i] })
            }
        }
    }
}
```

The system allows the read sensors data from the last 72 hours to be analyzed in graphical way. To be able to do that in an easy and simple way, the ChartView QML type was used, as can be seen in the code segment below. The function *showData* is called in the creation of the page and in the sensor selection through a buttons.

```
property var tdsData: []
property var phData: []
property var tempData: []
property var currentData: []

ChartView {
    id: sensorChart
    title: "Line"
    anchors.fill: parent
    antialiasing: true
    backgroundColor: "transparent"
    plotArea: Qt.rect(100,100,600,250)

    ValueAxis {
        id: axisY
        gridVisible: true
        tickCount: 5
        min: 0
        max: 1
    }
    ValueAxis {
        id: axisX
        min: 0
        max: 1
        gridVisible: false
        tickCount: 0
    }
}

function showData(sensor) {
    var line = sensorChart.createSeries(ChartView.SeriesTypeLine, "", axisX, axisY)
    var higherValue = sensor[0]
    var lowerValue = sensor[0]
    for (var i = 0; i < sensor.length; ++i) {
        line.append(i, sensor[i])
        if (sensor[i] > higherValue) {
            higherValue = sensor[i]
        }
        if(sensor[i] < lowerValue) {
            lowerValue = sensor[i]
        }
    }

    line.color = "white"
    axisX.min = 0
    axisX.max = sensor.length
    axisX.color = "white"
    axisY.min = lowerValue
    axisY.max = higherValue
    axisY.color = "white"
}
```

In the end, to implement a visually actrative method to shutdown the whole system, a switch was implemented using the Switch QML type.

```
Switch {
    id: switch_onoff
    x: 260; y:175
    width: 280
    height: 130
    hoverEnabled: true

    onCheckedChanged: aq.shutdown()

    indicator: Rectangle {
        width: 280
        height: 130
        radius: height / 2
        color: switch_onoff.checked ? "black" : "white"

        Behavior on color {
            ColorAnimation {duration: 600}
        }

        Text {
            x: 35
            anchors.verticalCenter: parent.verticalCenter
            font.pixelSize: 40
            color: "white"
            text: "OFF"
        }
        Text {
            x: 170
            anchors.verticalCenter: parent.verticalCenter
            font.pixelSize: 40
            color: "black"
            text: "ON"
        }

        Rectangle {
            x: switch_onoff.checked ? parent.width - width - 2 : 4
            width: parent.height - 6
            height: width
            radius: width / 2
            anchors.verticalCenter: parent.verticalCenter
            color: switch_onoff.checked ? "white" : "black"

            // smooth color and movement transitions
            Behavior on x {
                NumberAnimation { duration: 600 }
            }
            Behavior on color {
                ColorAnimation {duration: 600}
            }
        }
    }
}
```

24 | Validation

24.1 Test Cases

Daemon Test Cases

Test	Description	Expected Result	Real Result
Receive Message Queues	Read Message Queues Received	Message Queue data correct	Message Queue data correct
Send Message Queues	Send a message queue	Message queue present in the /mqueue folder	Message queue present in the /mqueue folder
Deamon Execution	Check if daemon is executing	Daemon working in background	Daemon working in background

Table 24.1: Power Supply Test Cases.

Sensors Test Cases

Test	Description	Expected Result	Real Result
Read Temperature	Verify if a correct temperature value is obtained	Correct temperature value obtained	Correct temperature value obtained
Read pH	Verify if a correct pH value is obtained	Correct pH value obtained	Correct pH value obtained
Read TDS	Check if the TDS value is correct	Correct TDS value obtained	Correct TDS value obtained

Table 24.2: Sensor Test Cases.

Actuators Test Cases

Test	Description	Expected Result	Real Result
Actuate Servo Motor	Test Servo Motor rotation	Servo motor start rotating	Servo motor start rotating
Actuate Water Heater	Test Water Heater heating process	Water heater start heating	Water heater start heating
Actuate UV Light	Test Light activation	Activation of Light	Activation of Light
Actuate Relay Module	Test relay output activation	Transition between NO and NC	Transition between NO and NC

Table 24.3: Sensor Test Cases.

Touchscreen Test Cases

Test	Description	Expected Result	Real Result
DSI Connection	Connect via DSI touchscreen	Display Image	Display Image
Input Response	Test the user input	Input is detected	Input is detected
Display Update	Send data to display	Values change in the interface	Values change in the interface

Table 24.4: Touch Screen Display Test Cases.

Database Test Cases

Test	Description	Expected Result	Real Result
Insert Data	Insert data into the database	Data inserted with sucess	Data inserted with sucess
Remove data	Remove data from the database	Data removed with sucess	Data removed with sucess
Get Fish Information	Receive a specific fish information	Fish information received	Fish information received
Get All Species	Receive all the species	All the species received	All the species received
Get Aquarium Species	Receive Aquarium species	Aquarium species received	Aquarium species received
Sensors Data	Insert sensors values	Sensors values into database	Sensors values into database

Table 24.5: Database Test Cases.

Power Supply Test Cases

Test	Description	Expected Result	Real Result
Power Raspberry	Connect supply to the raspberry	Raspberry turn On	Raspberry turn On
Power Sensors	Connect supply to the sensors	Sensors Leds On	Sensors Leds On
Power Actuators	Connect supply to the actuators	Actuators Leds On	Actuators Leds On

Table 24.6: Power Supply Test Cases.

GUI Test Cases

Test	Description	Expected Result	Real Result
Add Fish	Add a fish into the Aquarium	Fish added to database	Fish added to database
Remove Fish	Remove a fish from the Aquarium	Fish added to database	Fish removed from database
Sensors Read	Read sensor values	Sensor Values displayed	Sensor Values displayed
Sensors Graphs	Display sensor graph values	Sensor graph values displayed	Sensor graph values displayed
Ideal Conditions	Display ideal conditions	Show ideal conditions	Show ideal conditions
Fish Conditions	Display selected fish conditions	Show selected fish conditions	Show selected fish conditions
Display Species to Add	Display species presented in the database	Show all the species	Show all the species
Display Species to Remove	Display species presented in the aquarium	Show the species in the aquarium	Show the species in the aquarium
Shutdown Button	Turn off the system	The system turns off	The system turns off

Table 24.7: GUI Test Cases.

24.1.1

25 | Budget

Product	Quantity	Price
Raspberry Pi 4	1	65,00 €
Touch display 5” 800x400	1	42,90 €
pH Sensor E-201-C module v1.1	1	17,02 €
TZT TDS sensor meter	1	5,71 €
Servo motor	1	3,50 €
ADS1115	1	3,40 €
Ds3231 RTC module	1	2,55 €
Relay module	1	2,50 €
DS18B20	1	2,40 €
Water Heater	1	8,50 €
UV Light	1	9,10 €
Total		164,58 €

Table 25.1: Prototype Cost.

26 | Gantt Diagram

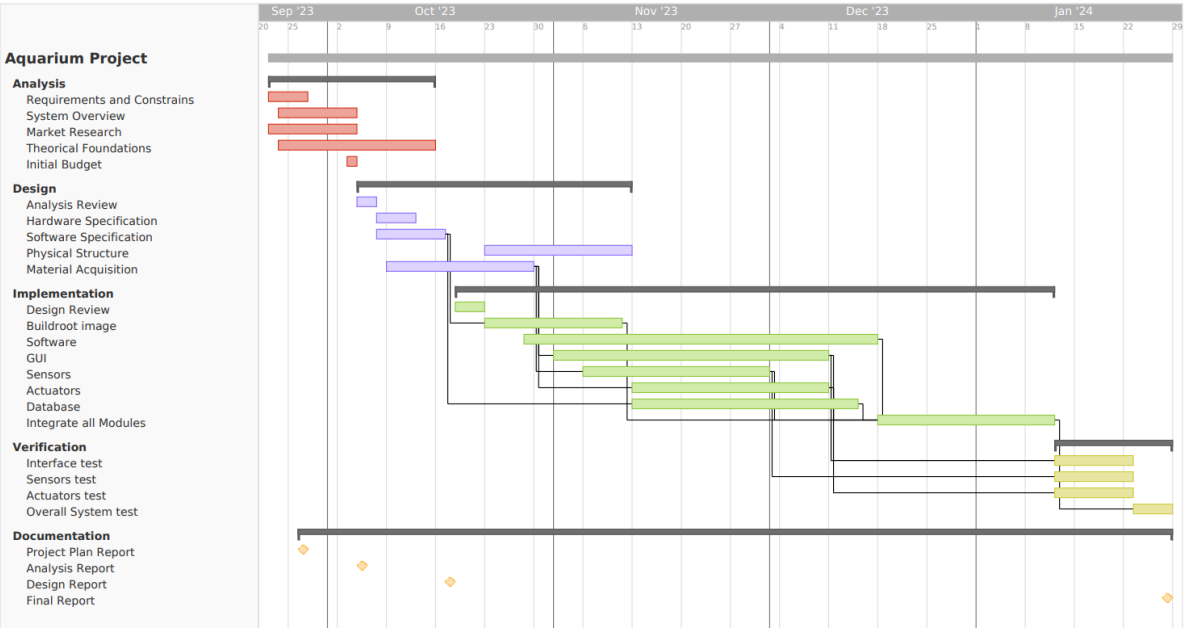

Figure 26.1: Gantt Diagram.

Bibliography

[1] https://www.coralvuehydros.com/control/control-4/

[2] https://www.seneye.com/store/seneye-home-device.html

[3] https://www.amazon.com/Yewhick-Controller-Saltwater-Aquaculture-Desalination/dp/B0C3TXYH6Q/ref=sr_1_4?crid=BM7JJWFLSS0G&keywords=aquarium+monitor&qid=1695683223&sprefix=aquarium+monitor%2Caps%2C360&sr=8-4

[4] https://www.ijraset.com/research-paper/smart-aquarium-monitoring-system-using-iot

[5] https://www.grandviewresearch.com/industry-analysis/reef-aquarium-market-report

[6] https://www.academia.edu/99380600/Smart_Aquarium_Monitoring_System_Using_IoT

[7] https://pt.aliexpress.com/item/4000038496514.html?spm=a2g0o.order_list.order_list_main.5.1f84caa466qmVU&gatewayAdapt=glo2bra

[8] https://pt.aliexpress.com/item/1005005687204657.html?spm=a2g0o.order_list.order_list_main.10.1f84caa466qmVU&gatewayAdapt=glo2bra

[9] Sandford, G. (2000). Manual Completo do Aquário. Livraria Civilização Editora.

[10] https://www.techtarget.com/searchsoftwarequality/definition/waterfall-model

[11] Kerrisk, M. (2010). The Linux Programming Interface. No Starch Press.

[12] https://en.wikipedia.org/wiki/Daemon_(computing)

[13] urlhttps://www.techtarget.com/searchnetworking/definition/mutex

[14] https://en.cppreference.com/w/cpp/thread/condition_variable

[15] Arpaci-Dusseau, R.H., & Arpaci-Dusseau, A.C. (2018). Operating Systems: Three Easy Pieces (Version 1.00). Arpaci-Dusseau Books.

[16] https://man7.org/linux/man-pages/man7/signal.7.html

[17] https://i2c.info/

[18] https://learn.sparkfun.com/tutorials/i2c/all

[19] https://www.analog.com/en/technical-articles/guide-to-1wire-communication.html

[20] https://www.engineersgarage.com/what-is-the-1-wire-protocol/

[21] https://en.wikipedia.org/wiki/Buildroot

[22] https://www.oracle.com/database/what-is-database/

[23] https://www.thingiverse.com/thing:2761061

[24] https://www.ti.com/lit/ds/symlink/ads1114.pdf

[25] https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

[26] https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf

[27] https://man7.org/linux/man-pages/man7/signal.7.html

[28] https://wiki.dfrobot.com/Gravity__Analog_TDS_Sensor___Meter_For_Arduino_SKU__SEN0244

[29] https://wiki.seeedstudio.com/Grove-PH-Sensor-kit/

[30] Lee Zhi Eng. (2016). Qt5 C++ GUI Programming Cookbook. Packt Publishing Ltd.