Department of Electronic & Telecommunication Engineering, University of Moratuwa, Sri Lanka.

Automated Aquaculture Tank Monitoring System
Design Documentation

H. M. G. N. I. Herath	210216F
H. M. V. Vidumini	210669U

Submitted in partial fulfillment of the requirements for the module
EN 2160 Electronic Design Realization

21st July 2024

Contents

1 Comprehensive Design Details

1.1 Introduction

This report presents a comprehensive analysis and design documentation for an advanced aquaculture monitoring and feeding system. The system integrates various sensors and electronic components to ensure optimal water quality and automate feeding processes in aquaculture environments. The aim is to provide a reliable, durable, and user-friendly solution that enhances the efficiency and effectiveness of fish tank management.

1.2 Project overview

Our project involves the design and development of an advanced aquaculture monitoring and feeding system. The system is engineered to provide real-time monitoring of crucial water parameters and automate the feeding process in fish tanks, thereby enhancing the efficiency and health management of aquaculture environments.

1.3 The primary objectives of the aquaculture tank monitoring device

The primary objectives of the device are:

- **Real-time Monitoring:** Provide continuous and accurate monitoring of pH, Total Dissolved Solids (TDS), and water temperature in fish tanks.
- **Alert System:** Notify users promptly through alarms and warning messages when monitored parameters fall outside specified ranges.
- **User Interface:** Offer an intuitive interface with LCD display and pushbuttons for easy navigation and parameter adjustment.
- **Feeding Automation:** Automate fish feeding schedules based on user-defined times or allow manual feeding control through the interface.
- **Durability and Reliability:** Ensure robust construction using materials like ABS plastic and metal stands to withstand aquatic environments.
- **Ease of Installation:** Facilitate simple installation on tank rims using hooks and adjustable screws, ensuring stability and optimal sensor placement.
- **Enhanced Aquaculture Management:** Support efficient and effective management of fish tanks to promote healthier aquatic environments and growth.

These objectives aim to enhance user experience, streamline aquarium maintenance, and contribute to the overall well-being of aquatic life through advanced monitoring and feeding automation capabilities.

1.4 Sensor Specifications

In our aquaculture tank monitoring system, accurate and reliable data collection is critical for maintaining optimal water quality conditions. The system uses 3 sensors to monitor the key parameters

- pH
- Total Dissolved Solids (TDS)
- water temperature

This section provides detailed specifications of the sensors integrated into the system. Each sensor is chosen for its precision, reliability, and suitability for continuous monitoring in an aquaculture environment.

1.4.1 pH Sensor

The pH level of water is a crucial parameter in aquaculture as it directly affects the health and growth of aquatic organisms. Maintaining an optimal pH range is essential for ensuring the metabolic processes of fish and other aquatic life are functioning properly. Deviations from the ideal pH range can lead to stress, disease, and even mortality. Therefore, monitoring pH levels continuously helps in maintaining a stable and healthy environment, improving the overall productivity and sustainability of the aquaculture operation.
The chosen sensor and its specifications are as follows:

Figure 1: pH Sensor E-201-C

Model: pH Sensor E-201-C

- **Measurement Range:** 0 - 14 pH
- **Accuracy:** ±0.01 pH
- **Resolution:** 0.01 pH
- **Operating Temperature:** 0°C to 50°C
- **Response Time:** ¡= 10 seconds for 95% of final value
- **Output Type:** Analog Voltage
- **Supply Voltage:** 3.3V to 5V
- **Calibration:** Two-point (4.0, 7.0, and 10.0 pH)
- **Connector:** BNC connector for pH probe

1.4.2 TDS Sensor

Total Dissolved Solids (TDS) measure the combined content of all inorganic and organic substances contained in a liquid, which are present in a molecular, ionized, or micro-granular suspended form. In aquaculture, maintaining appropriate TDS levels is essential for the health of aquatic organisms as it affects water hardness, salinity, and the overall ionic balance. High TDS levels can indicate poor water quality and the presence of harmful substances, whereas low levels can lead to nutrient deficiencies. Continuous monitoring of TDS helps in maintaining optimal water quality and preventing potential health issues in the aquaculture environment.
The chosen sensor and its specifications are as follows:

Figure 2: Gravity TDS Meter V1.0

Model: Gravity TDS Meter V1.0

- **Measurement Range:** 0 - 1000 ppm
- **Accuracy:** ±10% full scale
- **Resolution:** 1 ppm
- **Operating Temperature:** 0°C to 40°C
- **Response Time:** 10 seconds
- **Output Type:** Analog Voltage
- **Supply Voltage:** 3.3V to 5V
- **Calibration:** One-point (700 ppm)
- **Connector:** 3-pin JST connector

1.4.3 Water Temperature Sensor

Water temperature is a critical factor in aquaculture as it influences the metabolic rates, growth, and reproductive cycles of aquatic organisms. Each species has a preferred temperature range, and deviations from this range can cause stress, reduce immune response, and impact overall health. Additionally, temperature affects the solubility of oxygen and the efficacy of other water quality parameters. Continuous temperature monitoring allows for the maintenance of optimal conditions, ensuring a healthy and productive aquaculture environment.
The chosen sensor and its specifications are as follows:

Figure 3: DS18B20 Digital Thermometer

Model: DS18B20 Digital Thermometer

- **Measurement Range:** -55°C to +125°C
- **Accuracy:** ±0.5°C (from -10°C to +85°C)
- **Resolution:** Programmable 9 to 12 bits
- **Response Time:** ¡= 750 milliseconds
- **Output Type:** Digital (1-Wire Protocol)
- **Supply Voltage:** 3.0V to 5.5V
- **Connector:** 3-pin (VCC, GND, Data)

1.5 Microcontroller specifications

The Atmel® picoPower® ATmega328P microcontroller is an excellent choice for this project due to its low-power CMOS architecture and high performance. This 8-bit microcontroller, based on the AVR® enhanced RISC architecture, executes powerful instructions in a single clock cycle, achieving throughputs close to 1MIPS per MHz. This allows for optimization between power consumption and processing speed, making it ideal for sensor applications.
The below is a summary of key features which are extracted from ATMega328P datasheet.

Key Features

- **Advanced RISC Architecture:**
  - 131 powerful instructions, mostly single clock cycle execution.
  - Up to 20 MIPS throughput at 20MHz.
- **Memory:**
  - 32KB In-System Self-Programmable Flash memory.
  - 1KB EEPROM.
  - 2KB SRAM.
- **Peripheral Features:**
  - 8-channel 10-bit ADC.
  - Six PWM channels.
  - Two 8-bit Timer/Counters and one 16-bit Timer/Counter.
  - Master/Slave SPI Serial Interface.
  - I2C compatible 2-wire Serial Interface.
- **Power Consumption:**
  - Active Mode: 0.2mA at 1MHz, 1.8V.
  - Power-down Mode: 0.1μA.
  - Power-save Mode: 0.75μA (including 32kHz RTC).
- **Operating Voltage:**
  - 1.8 - 5.5V.
- **Operating Temperature Range:**
  - -40°C to 105°C.

1.6 Circuit Analysis for Electronic Sub-Assemblies

The electronic design for our comprehensive monitoring and management system consists of 7 main electronic sub-assemblies. These sub-assemblies work together to monitor and maintain optimal conditions in fish tanks and pools used in both ornamental fish breeding and fish farming for human consumption. The sub-assemblies include,

1. Power Regulator
2. Microcontroller Unit
3. pH Sensor Module and Interface
4. Water Temperature Sensor Module and Interface
5. TDS Sensor Module and Interface
6. User Interface
7. Motor Drive and Switch

Each sub-assembly plays a critical role in ensuring the system's functionality and reliability. This document provides detailed descriptions of each sub-assembly, including their specifications, components, and functionality within the system.

1.6.1 Power Regulator

The power regulator converts the input voltage from 12V to a stable 5V using the LM7805CT power regulator.

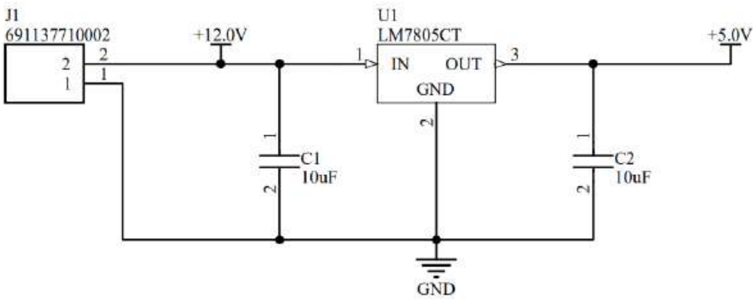


Figure 4: Voltage Regulator Circuit

Components and Sections

1. **Power Input:** The 12V DC input is connected to the circuit via the input connector J1.
2. **Input Filtering:** Capacitor C1, connected across the input voltage, filters out any noise or spikes from the input power supply. This ensures that the voltage regulator receives a clean 12V DC input.
3. **Voltage Regulation:** The LM7805CT voltage regulator receives the filtered 12V input at its IN pin (Pin 1). It regulates this input down to a stable 5V DC, which is provided at its OUT pin (Pin 3). The ground pin (Pin 2) is connected to the common ground of the circuit.

4. **Output Filtering:** Capacitor C2, connected across the output voltage, smooths out any residual noise or ripple in the 5V output, providing a clean and stable 5V DC to the load.

In summary, the power regulator sub-assembly is designed to take a 12V DC input and provide a stable 5V DC output using an LM7805CT voltage regulator. The capacitors (C1 and C2) are crucial for filtering and stabilizing the input and output voltages, respectively. Therefore, this ensures that all components in the system receive a consistent and reliable power supply, which is essential for their optimal operation.

The microcontroller unit serves as the brain of the system, orchestrating various tasks through its sub-units: power supply filtering, oscillator circuit, in-system programming, reset circuit, and peripheral I/O connections. Each sub-unit plays a critical role in ensuring the functionality and stability of the system.

1.6.2 Microcontroller Unit

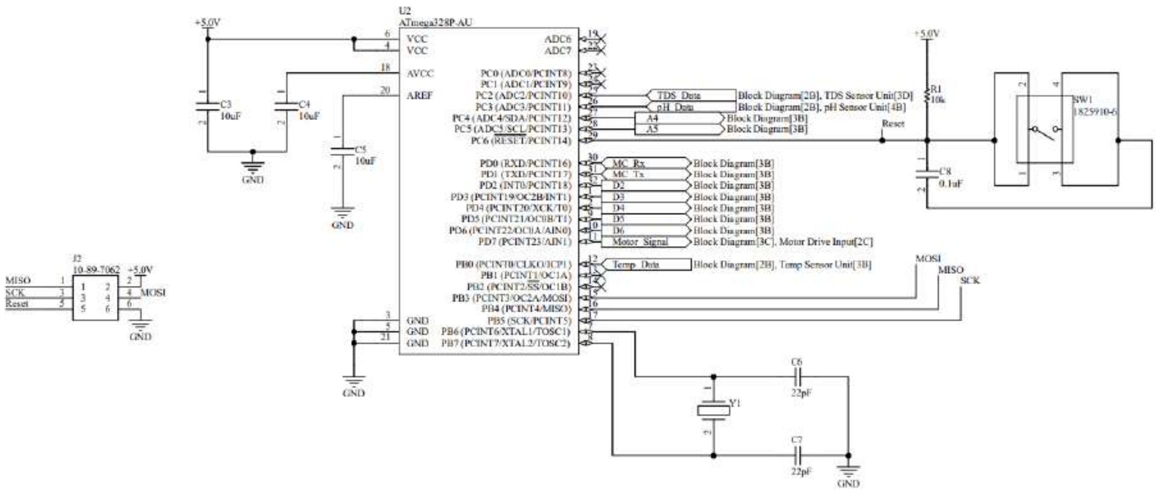


Figure 5: Microcontroller Unit Circuit

1. **Power Supply Filtering**
   - **Components:** C3, C4, C5 (10µF Capacitors)
   - **Function:** These capacitors filter the 5V power supply to provide a stable voltage to the microcontroller and other connected components, minimizing voltage fluctuations and noise.

   Components and Sections

2. **In-System Programming (ISP)**
   - **Component:** J2 (Programming Header)
   - **Function:** This header is used for programming the microcontroller via ISP, allowing firmware updates and debugging without removing the microcontroller from the circuit.

3. **Oscillator Circuit**
   - **Components:** Y1 (Crystal Oscillator), C6, C7 (22pF Capacitors)
   - **Function:** The crystal oscillator (Y1) provides the clock signal necessary for the microcontroller's operations. The capacitors (C6 and C7) stabilize the oscillations of the crystal, ensuring accurate and reliable timing.

4. **Reset Circuit**
   - **Components:** R1 (10k Resistor), C8 (0.1µF Capacitor), SW1 (Reset Switch)
   - **Function:**

   - **R1:** The pull-up resistor keeps the reset pin high under normal conditions.
   - **C8:** The capacitor debounces the reset switch to prevent false triggering.
   - **SW1:** The reset switch allows for manually resetting the microcontroller, reinitializing the system as needed.

5. **Peripheral Connections**

   - **Analog and Digital I/O Pins:**
     - **ADC6, ADC7 (Pins 18, 20):** Used for receiving data from the TDS and pH sensors.
     - **Digital Pins (PC0 - PC6, PD0 - PD7, PB0 - PB7):** Connected to various sensors, the motor driver, and user interface elements, facilitating communication and control.

   - **Serial Communication Pins:**
     - **TX, RX (PD0, PD1):** These pins are designated for serial communication, enabling data exchange with other devices.
     - **SPI (PB3 - PB5):** These pins are used for in-system programming and communication with other SPI devices, supporting data transfer and device interfacing.

The microcontroller unit is integral to the system's operation, coordinating tasks through well-structured sub-units. These include power supply filtering for voltage stability, an oscillator circuit for precise timing, ISP for easy programming, a reset circuit for system reinitialization, and various peripheral connections for interfacing with sensors and other components.

1.6.3 pH Sensor Module and Interface

The pH sensor module interfaces with the pH sensor to obtain pH readings, process the signal, and transmit it to the microcontroller. This module includes operational amplifiers (op-amps), voltage references, and other necessary components for measuring and processing pH levels from a pH sensor.

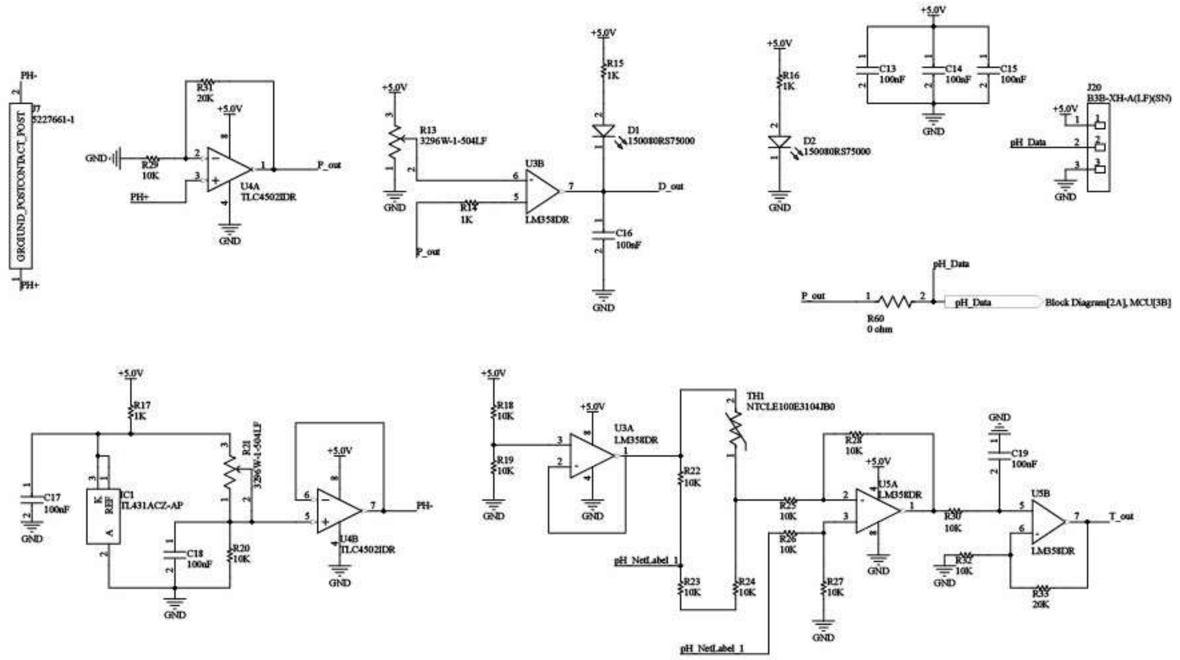


Figure 6: pH sensor module

Components and Functions

1. **pH Sensor Input**
   - **Function:** Connects the pH sensor to the module, allowing the analog signal proportional to the pH level to be processed.
   - **Component:** Connector (J7)
2. **Signal Conditioning**
   - **Function:** Buffers and stabilizes the pH signal from the sensor to isolate it from the rest of the circuit.
   - **Components:**
     - Op-Amp: U4A (TLC4502IDR)
     - Resistors: R14, R15, R16
     - Capacitors: C13, C14, C15
3. **Signal Amplification**
   - **Function:** Amplifies the buffered pH signal to a level suitable for further processing.
   - **Components:**
     - Op-Amp: U3B (LM358DR)
     - Resistors: R17, R18
     - Capacitors: C16, C17
4. **Voltage Reference**
   - **Function:** Provides a stable reference voltage for the op-amps to ensure accurate measurements.

- **Components:**
  - Voltage Reference: U1 (TL431ACZ-AP)
  - Resistors: R19, R20
  - Capacitors: C18, C19

5. **Output Signal Conditioning**

- **Function:** Further conditions the amplified signal, making it suitable for the ADC in the microcontroller.
- **Components:**
  - Op-Amp: U5A (LM358DR)
  - Resistors: R22, R23, R24
  - Capacitors: C19
  - Diodes: D1, D2

6. **Temperature Compensation**

- **Function:** Compensates the pH signal for temperature variations to ensure accuracy across different temperatures.
- **Components:**
  - Thermistor: TH1 (NTCLE100E3104JB0)
  - Op-Amp: U3A (LM358DR)
  - Resistors: R21, R22, R23, R24
  - Capacitors: C19

7. **Backup Connector**

- **Function:** Provides the conditioned pH signal to the microcontroller for further processing and analysis.
- **Components:**
  - Connector: J2
  - Op-Amp: U5B (LM358DR)

This pH sensor module is designed to interface a pH sensor with a microcontroller. It conditions the analog signal from the pH sensor, amplifies it, compensates for temperature variations, and provides a stable, noise-free output signal to the microcontroller. The use of op-amps, voltage references, and passive components ensures accurate and reliable pH measurements.

1.6.4 Water Temperature Sensor Module and Interface

The digital water temperature connected to the device operates using the sub-assembly given below.

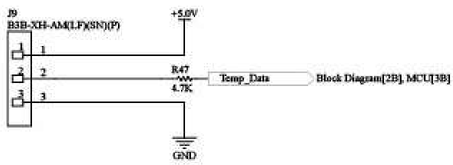


Figure 7: Water Temperature Sensor Module

Functionality: This module interfaces a temperature sensor with a microcontroller, providing the necessary power and signal conditioning to ensure accurate temperature data is communicated to the microcontroller.

Components:

- **Connector:** J9 - B3B-XH-AM(LF)(SN)(P)
- **Pull-up Resistor:** R47
- **Temperature Data Line:** Temp_Data

1.6.5 TDS Sensor Module and Interface

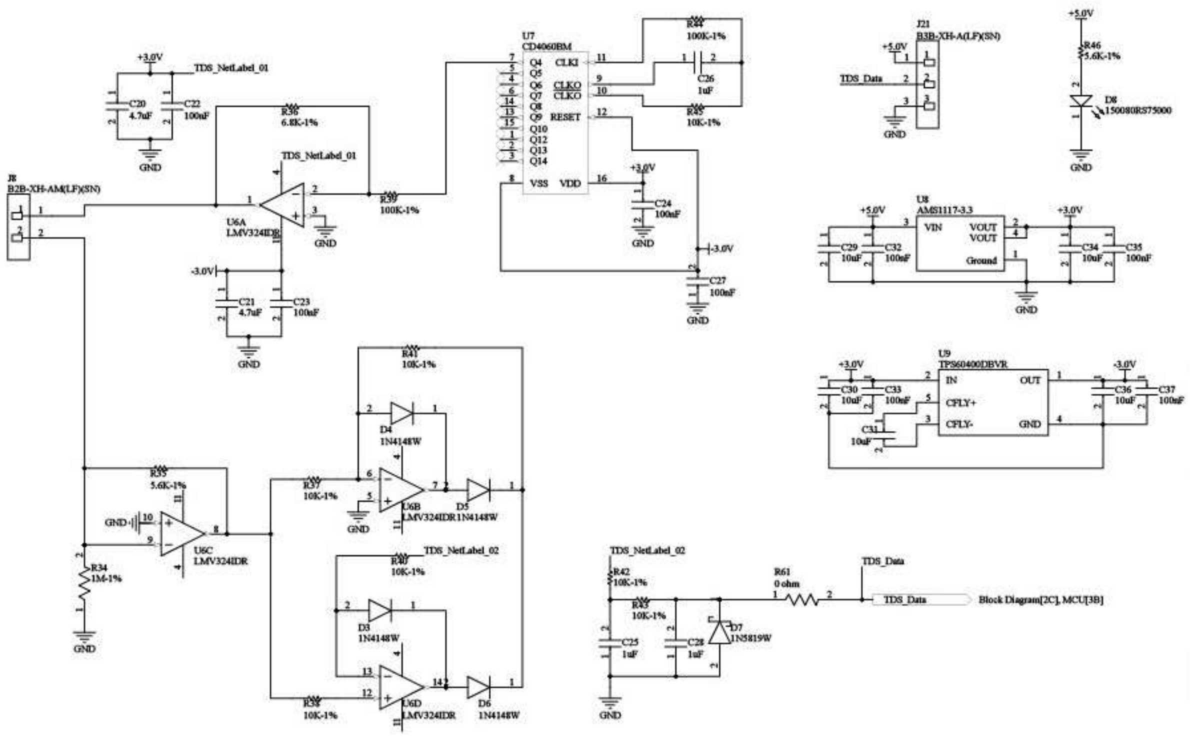


Figure 8: TDS Sensor module

TDS Sensor Module and Interface provides an interface to the TDS sensor to connect with the microcontroller. It includes various operational amplifiers (op-amps), voltage regulators, and other components necessary for measuring and processing TDS levels from a TDS sensor.

Components and Sections

1. **Power Supply**

   Functionality: Provides stable voltage supply to various parts of the circuit.

   Components:

   - **Voltage Regulators:**
     - **AMS1117-3.3 (U8):** Regulates the input 5V to a stable 3.3V output.
     - **TPS60400DBVR (U9):** Provides a negative voltage (-3V) from a positive 3.3V input.
   - **Capacitors:**
     - **C28, C29, C30:** Decoupling capacitors for U8.
     - **C31, C32, C33, C34, C35, C36, C37:** Charge pumping and output stabilization capacitors for U9.

2. **TDS Sensor Input**

   Functionality: Receives the input signal from the TDS sensor.
   Components:

   - **Connectors**

  - **J8:** Primary input for the TDS sensor.
  - **J21:** Secondary connection point, for calibration of the sensor and backup purposes.

3. **Signal Conditioning**

   Functionality: Buffers and amplifies the input signal from the TDS sensor.

   Components:

   - **Op-Amp (U6A - LMV324IDR):** Buffers and amplifies the TDS sensor signal.
   - **Resistors:**
     - **R39, R40, R41, R42:** Set gain and biasing for U6A.
   - **Capacitors:**
     - **C20, C21, C22, C23:** Decoupling and noise reduction.
   - **Voltage Reference (U7 - CJ4006DM):** Provides a stable reference voltage.
   - **Resistors:**
     - **R43, R44, R45, R46:** Set output voltage for U7.
   - **Capacitors:**
     - **C24, C25, C26, C27:** Stabilize the reference voltage.

4. **Signal Amplification**

   Functionality: Further amplifies and conditions the TDS signal.

   Components:

   - **Op-Amps (U6B, U6C, U6D - LMV324DR):** Amplify and condition the TDS signal.
   - **Resistors:**
     - **R34, R35, R36, R37, R38, R47, R48, R49, R50:** Set gain and biasing for amplification stages.
   - **Capacitors:**
     - **C19, C38, C39, C40, C41, C42, C43, C44:** Filtering and decoupling.
   - **Diodes:**
     - **D1, D2, D3, D4, D5, D6:** Protect op-amps from voltage spikes.

5. **Output Signal Conditioning**

   Functionality: Buffers the final TDS signal before sending it to the microcontroller.

   Components:

   - **Op-Amp (U6D - LMV324DR):** Buffers the final TDS signal.
   - **Resistors:**
     - **R60, R61:** Set output level and feedback.
   - **Capacitor:**
     - **C25:** Stabilizes the output signal.

6. **Output to Microcontroller**

   Functionality: Provides the conditioned TDS signal to the microcontroller for further processing.

   Components:

   - **Connector (J21):** Delivers the TDS_Data signal to the microcontroller.

This circuit is designed to interface a TDS sensor with a microcontroller. It conditions the analog signal from the TDS sensor, amplifies it, and provides a stable, noise-free output signal to the microcontroller. The use of op-amps, voltage regulators, and passive components ensures accurate and reliable TDS measurements. The negative voltage generator and multiple stages of amplification and filtering are crucial for achieving high precision in TDS readings.

1.6.6 User Interface

The user interface circuit interfaces the buttons, display, and the WiFi module to the microcontroller, allowing for user interaction and connectivity. It includes connectors and necessary components for stable and reliable operation.

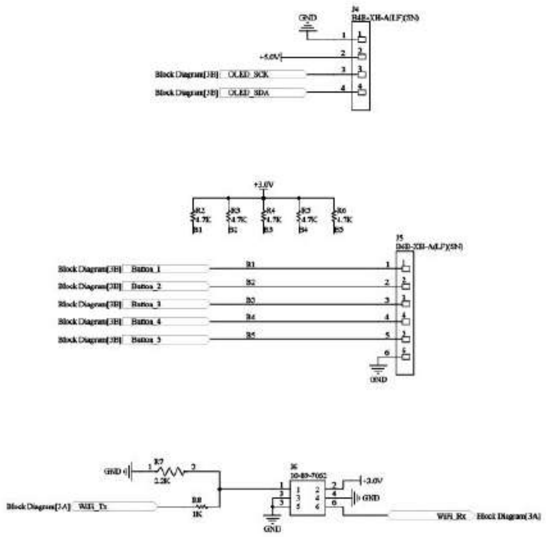


Figure 9: User Interface

Functionality and Components

1. **Display:** The display module connects through the J4 connector, allowing the microcontroller to send data and commands for visual output.
    - **Connector (J4):**
        - Provides the interface for the display module to connect to the circuit.
        - Ensures proper data and power connections between the microcontroller and the display.
2. **Buttons Panel:** The buttons connect through the J5 connector. When a button is pressed, the corresponding input signal changes state, which is detected by the microcontroller.
   The pull-up resistors (R2, R3, R4, R5, R6) ensure stable input signals and reduce noise.
    - **Connector (J5):**
        - Provides the interface for the buttons panel to connect to the circuit.
    - **Resistors (R2, R3, R4, R5, R6):**
        - Pull-up resistors ensure that each button input is at a known voltage level when not pressed.
        - Provides debounce functionality to avoid multiple triggers from a single button press.
3. **WiFi Module Interface:** The WiFi module connects through the J6 connector, enabling wireless communication.
   Resistor R7 ensures the signals between the microcontroller and the WiFi module are stable and within proper voltage levels.
    - **Connector (J6):**
        - Provides the interface for the WiFi module to connect to the circuit.
    - **Resistor (R7):**
        - Ensures the proper biasing and signal integrity for the WiFi module interface.

Additionally, it should be mentioned that the buzzer for the alarm system which is also a part of the user interface is to be connected to pin 12, after the microcontroller is programmed.

This user interface circuit facilitates the interaction between the user and the system via buttons and display, and enables wireless communication through the WiFi module. The use of connectors and resistors ensures stable and reliable connections and signal integrity.

1.6.7 Motor Drive and Switch

The motor driver part controlls the motor mechanism and provides and interface to the motor in the feeder.

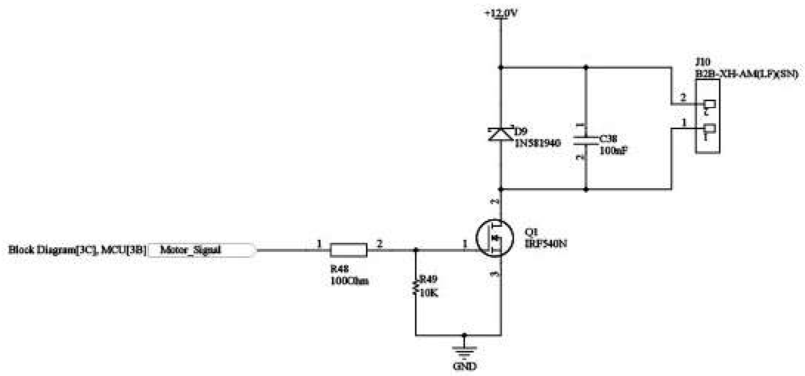


Figure 10: Motor Drive and switch

Components and Connections

1. **MOSFET (Q1 - IRF540N):**
   - **Function:** Acts as a switch to control the motor. The gate of the MOSFET receives the control signal (*Motor_Signal*), which determines whether the MOSFET is on or off.
   - **Gate Resistor (R48 - 100 ohms):** Limits the inrush current to the gate to protect the MOSFET and control the switching speed.
   - **Pull-down Resistor (R49 - 10k ohms):** Ensures that the MOSFET stays off when there is no input signal, preventing accidental activation.
2. **Diode (D9 - 1N5819):**
   - **Function:** Provides a path for the current when the motor is turned off, protecting the MOSFET from voltage spikes caused by the inductive load (motor).
   - **Protection:** Ensures the longevity of the MOSFET by preventing back EMF from damaging it.
3. **Connector (J10):**
   - **Function:** Connects the motor to the circuit.
   - **Pin 1:** Connected to the drain of the MOSFET.
   - **Pin 2:** Connected to the +12V supply.
4. **Capacitor (C38 - 100uF):**
   - **Function:** Provides decoupling and helps to smooth out voltage fluctuations and noise from the power supply.
   - **Stabilization:** Ensures stable operation of the motor by reducing voltage spikes and ripples.

5. **Control Signal (*Motor_Signal*):**
   - **Source:** Comes from the microcontroller unit (MCU) and is applied to the gate of the MOSFET through the gate resistor.
   - **Function:** Determines the on/off state of the MOSFET, thereby controlling the motor.

Operation

- **When *Motor_Signal* is High:**
  - The gate of the MOSFET (Q1) receives a high signal, turning the MOSFET on.
  - Current flows from the +12V supply through the motor (connected to J10), the MOSFET, and to ground, powering the motor.
- **When *Motor_Signal* is Low:**
  - The gate of the MOSFET (Q1) receives a low signal, turning the MOSFET off.
  - No current flows through the motor, thus turning it off.
- **Protection Mechanism:**
  - The diode (D9) protects against voltage spikes by providing a path for the inductive kickback current.
  - The capacitor (C38) stabilizes the supply voltage, ensuring smooth operation of the motor.

In conclusion, this motor drive circuit utilizes a MOSFET (IRF540N) to control a motor using a digital signal (*Motor_Signal*) from a microcontroller. The design includes protective components such as a diode to prevent voltage spikes and a capacitor to smooth power supply fluctuations. This simple yet effective circuit ensures reliable motor control and protection of the components.

1.7 PCB Specifications

1.7.1 Trace Width Calculations

Trace width is a crucial parameter in PCB design, influencing the board's electrical performance and thermal management. Proper trace width ensures that the traces can handle the expected current without excessive heating or voltage drop, thereby maintaining the reliability and functionality of the PCB.

Calculation Methodology:

The trace width for different nets in a PCB is calculated based on the maximum current the trace needs to carry, the permissible temperature rise, and the PCB material properties. The IPC-2221 standard provides guidelines for determining the appropriate trace width for given current and temperature rise requirements.

The formula used for calculating the trace width is derived from the IPC-2221 standards:

$$W = \frac{I}{k \cdot (T_r)^{0.44} \cdot (A)^{0.725}}$$

where:
- W is the trace width (in mils)
- I is the current (in amperes)
- k is a constant (0.024 for external layers, 0.048 for internal layers)
- T_r is the temperature rise (in °C)
- A is the cross-sectional area (in mils^2)

Design Requirements:

The calculations are done using these assumptions and facts:

- **Ambient Temperature:** 25°C
- **Temperature Rise:** 10°C
- **Copper Thickness:** 1 oz/ft^2

The PCB design includes the following trace widths for different nets:

1. **55 mil** for **Powering the Motor**
2. **15 mil** for **Power Traces**
3. **10 mil** for **Signal Traces**

Calculations:

1. **55 mil** for **Powering the Motor**
   Considering
   - a maximum trace length: 4000mil
   - maximum current: 1.5A

   Calculated trace width:
   - maximum: 53 mil
   - minimum: 18 mil

   Therefore trace width of **55 mil** is chosen considering other possibilities as well.

2. **15 mil** for **Power Traces** Considering

   - a maximum trace length: 9000 mil
   - maximum current: 200 mA

   Calculated trace width:

   - maximum: 3.5 mil
   - minimum: 1.5 mil

   Therefore trace width of **15 mil** is chosen considering other possibilities as well.

3. **10 mil** for **Signal Traces** Considering

   - a maximum trace length: 10 000 mil
   - maximum current: 50 mA

   Calculated trace width:

   - maximum: 0.2 mil
   - minimum: 0.5 mil

   Therefore trace width of **10 mil** is chosen considering other possibilities as well.

Using larger trace widths than strictly required by the calculated values offers several advantages in PCB design.

- Firstly, it enhances reliability by reducing resistance and thereby minimizing voltage drops, especially crucial for high-current paths like power lines to motors or critical components.
- Secondly, larger traces can handle higher currents safely without excessive heating, which extends the lifespan of the PCB and reduces the risk of thermal damage.
- Additionally, during manufacturing and assembly processes, having slightly oversized traces can simplify production by reducing the sensitivity to slight variations in fabrication tolerances or soldering quality.

Overall, while there may be a marginal increase in material cost or PCB size, the benefits in terms of reliability, longevity, and ease of manufacturing often justify using larger trace widths when feasible in electronic designs.

1.7.2 Power Calculations

We refered to data sheets and identified the below voltage and current requirements

- **pH Sensor:**
  - Current drawn: 20mA
  - Voltage: 5V
- **Digital Temperature Sensor:**
  - Current drawn: 6mA
  - Voltage: 5V
- **TDS Sensor:**
  - Current drawn: 6mA
  - Voltage: 5V
- **Microcontroller:**
  - Active mode: 1.5mA
  - Sleep mode: 1uA
- **Wi-Fi Module:**
  - Active mode: 200mA
  - Voltage: 3.3V
- **LCD Display:**
  - Current drawn: 1.65mA
  - Voltage: 5V
- **Buzzer:**
  - Current drawn: 10mA
  - Voltage: 5V
- **DC Motor:**
  - Current drawn: 300mA
  - Voltage: 12V

After reviewing the current requirements for each component in our system, the trace width calculations were conducted to ensure robust PCB design. The highest current draw identified was from the DC Motor at approximately 300mA. Considering a safety margin, the trace width was calculated for 450mA to account for variations and ensure reliability. Using the IPC-2221 standard and assuming 1oz copper thickness, the calculated trace width for the DC Motor was approximately 6.81 mils (0.173 mm), rounded to the nearest standard practice of 10 mils (0.254 mm) for power lines. This approach ensures adequate current carrying capacity and supports the overall reliability of the PCB design for our aquaculture monitoring system.

1.7.3 Design Rule Violation Report

The following table shows the design rules violation report for the PCB design. The report indicates that there are no violations, ensuring that the design adheres to all specified constraints.

Rule Violations	**Count**
Clearance Constraint (Gap=5mil) (All),(All)	0
Short-Circuit Constraint (Allowed=No) (All),(All)	0
Un-Routed Net Constraint ((All))	0
Modified Polygon (Allow modified: No), (Allow shelved: No)	0
Width Constraint (Min=0.05mil) (Max=55mil) (Preferred=5mil) (All)	0
Power Plane Connect Rule(Direct Connect)(Expansion=20mil) (Conductor Width=10mil) (Air Gap=10mil) (Entries=4) (InPadClass('PowerPads'))	0
Power Plane Connect Rule(Relief Connect)(Expansion=11.811mil) (Conductor Width=5mil) (Air Gap=5mil) (Entries=4) (All)	0
Minimum Annular Ring (Minimum=3mil) (All)	0
Hole Size Constraint (Min=11.811mil) (Max=248.031mil) (All)	0
Hole To Hole Clearance (Gap=9.842mil) (All),(All)	0
Minimum Solder Mask Sliver (Gap=0mil) (All),(All)	0
Silk To Solder Mask (Clearance=5mil) (IsPad),(All)	0
Silk to Silk (Clearance=0mil) (All),(All)	0
Net Antennae (Tolerance=0mil) (All)	0
Board Clearance Constraint (Gap=0mil) (All)	0
Height Constraint (Min=0mil) (Max=71497.938mil) (Preferred=500mil) (All)	0
Total	**0**

Table 1: Design Rules Violation Report

1.8 PCB Design

The documents containing layout and artwork for the PCB are as follows:

1.8.1 3D Views			
Sub Topic Number	**Figure Number**	**Caption**	**Page**
1	Figure 11	3D view - Top Layer	23
2	Figure 12	3D view - Bottom Layer	24

Table 2: 3D Views

1.8.2 Final Artwork Drawings			
Sub Topic Number	**Figure Number**	**Caption**	**Page**
1	Figure 13	Top-Layer	25
2	Figure 14	Bottom Layer	26
3	Figure 15	Top Silkscreen Overlay	27
4	Figure 16	Mechanical Drawing	28
5	Figure 17	Top Designator	29
6	Figure 18	Top Assembly	30
7	Figure 19	Top Component Outline	31
8	Figure 20	Top 3D Body	32
9	Figure 21	Top Courtyard	33
10	Figure 22	Top Component Centre	34
11	Figure 23	Top Pad Master	35
12	Figure 24	Bottom Pad Master	36
13	Figure 25	Drill Drawing for Top-Bottom Layer	37
14	Figure 26	Drill Guide for Top-Bottom Layer	38

Table 3: Final Artwork Drawings

1.8.3 Gerber Files			
Sub Topic Number	**Figure Number**	**Caption**	**Page**
1	Figure 27	Copper Signal Bottom/Top	39
2	Figure 28	Legend Top	40
4	Figure 29	Paste Top	40
5	Figure 30	Profile	41
7	Figure 32	Soldermask Top/Bottom	42

Table 4: Gerber Files

1.8.1 3D Views

The 3D views provides a three-dimensional visualization of the bottom side of the PCB with all its components. This view allows for a thorough inspection of the bottom side, checking the placement and orientation of components, and ensuring proper clearance and fit. Like the top view, it is also valuable for confirming the design and for presentations. These are crucial for visualizing the final product before manufacturing, helping to identify any potential issues in component placement or board design.

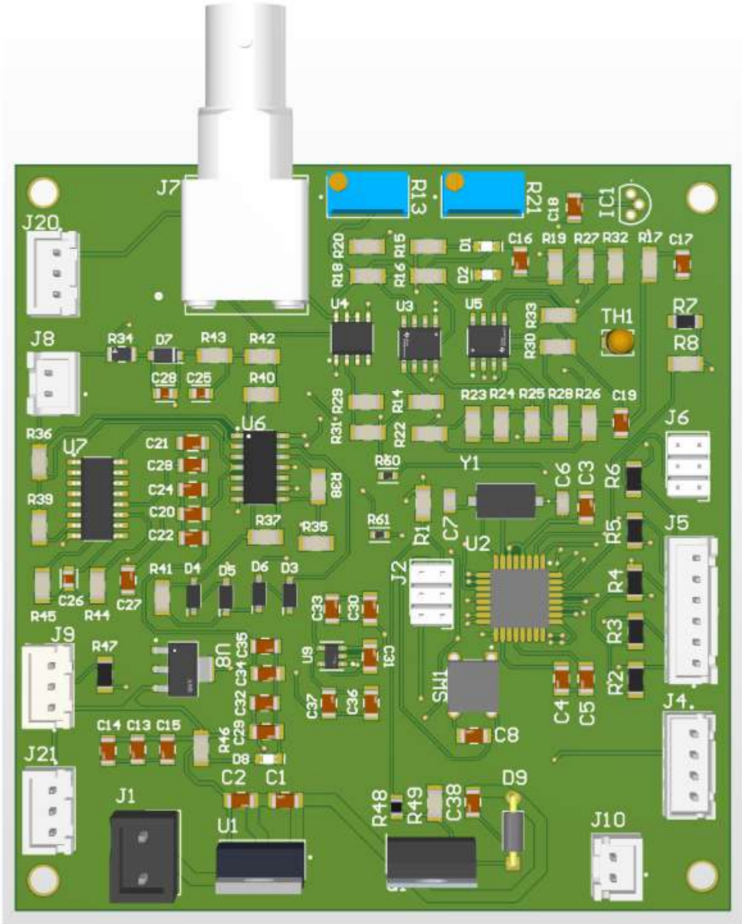


Figure 11: 3D view - Top Layer

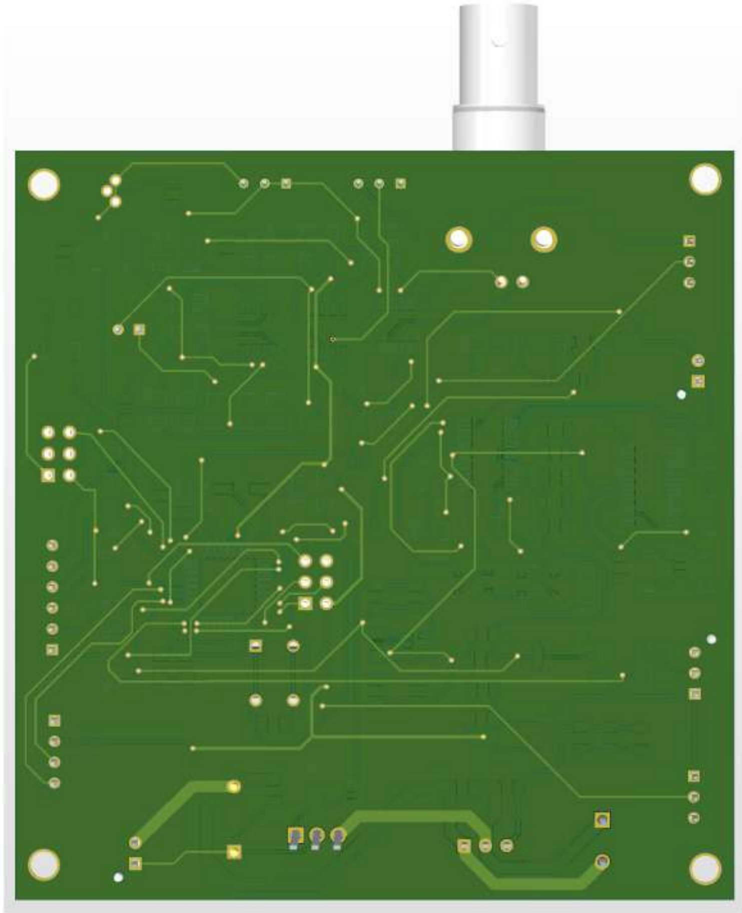


Figure 12: 3D view - Bottom Layer

1.8.2 Final Artwork Drawings

1. **Top Layer**

   The top layer of the PCB contains the copper traces and pads on the top side of the board. It shows the electrical connections between the components placed on the top side of the PCB.

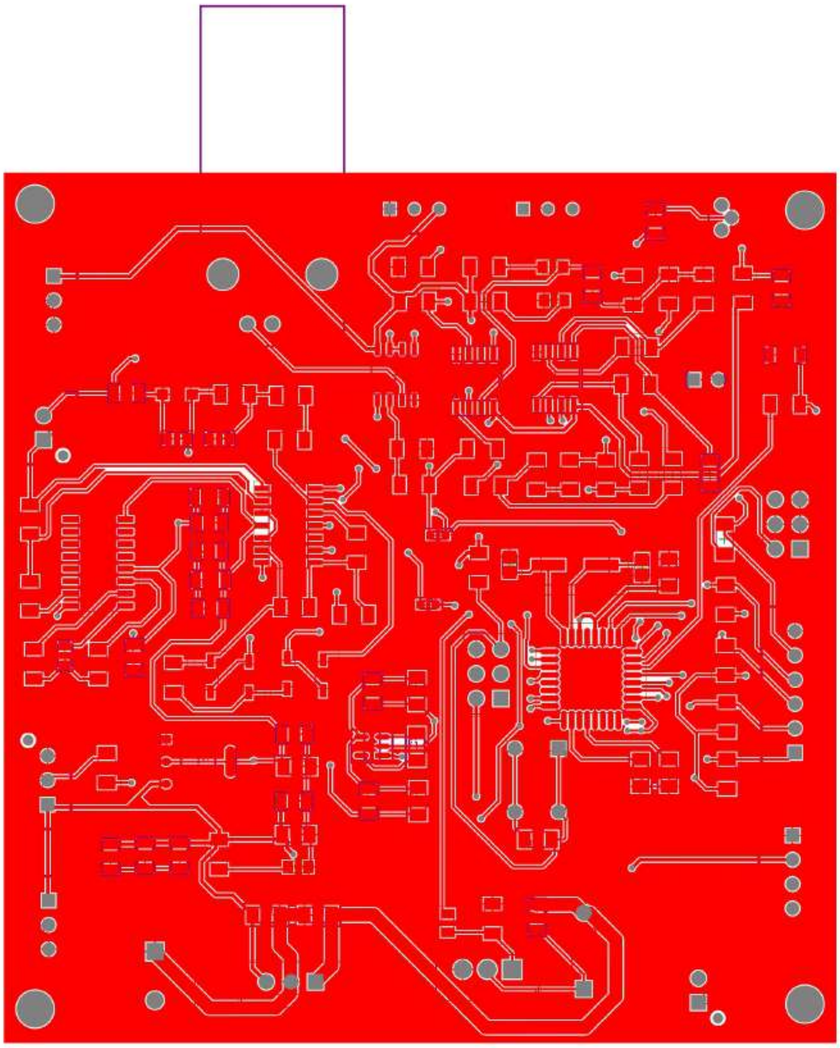


Figure 13: Top-Layer

2. **Bottom Layer**

   The bottom layer of the PCB contains the copper traces and pads on the bottom side of the board. It shows the electrical connections between the components placed on the bottom side of the PCB.

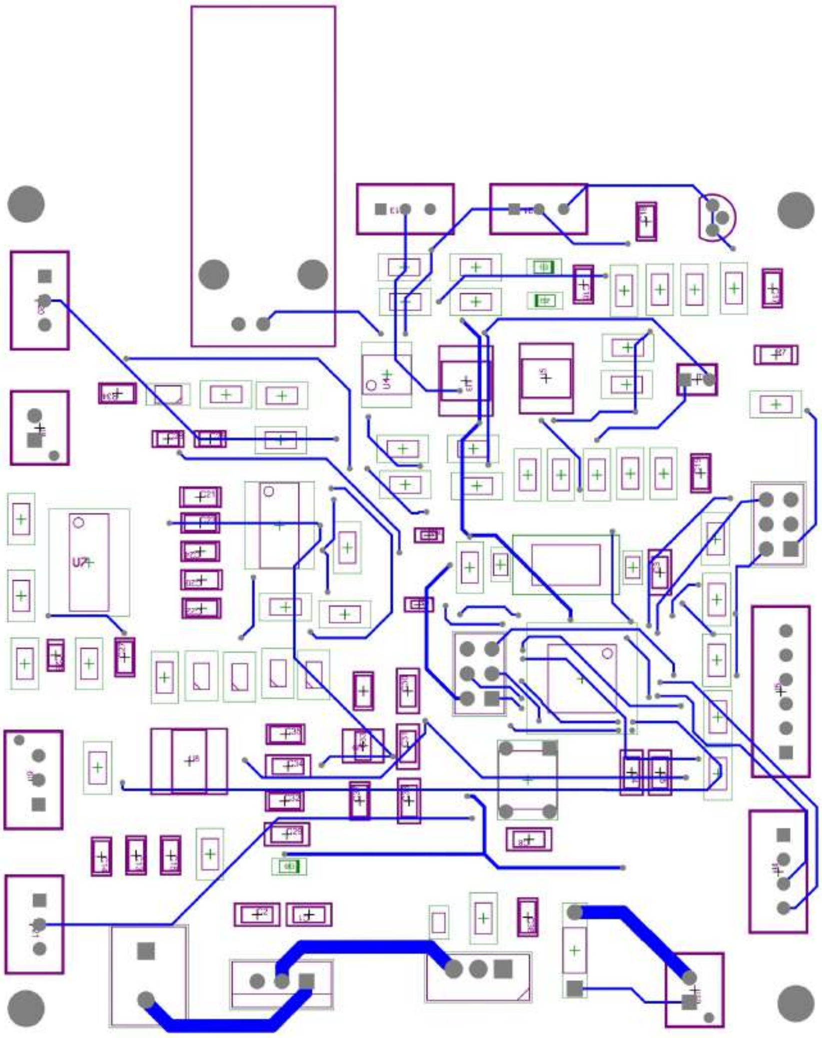


Figure 14: Bottom Layer

3. **Top Silkscreen Overlay**

The top silkscreen overlay includes component outlines, reference designators, and other labels printed on the top surface of the PCB. It aids in identifying components and their placement during assembly.

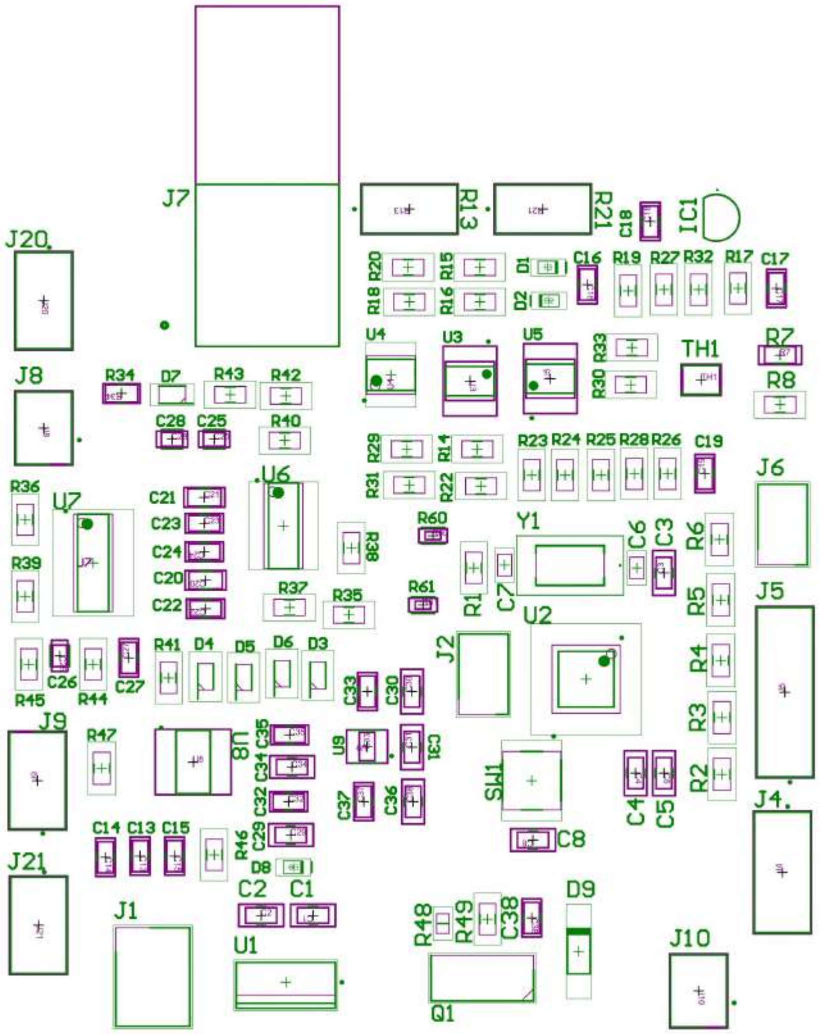


Figure 15: Top Silkscreen Overlay

4. **Mechanical1**

   The mechanical layer provides detailed information about the physical dimensions and shape of the PCB. It includes board outlines, mounting holes, and any cutouts required for the PCB to fit in its enclosure.

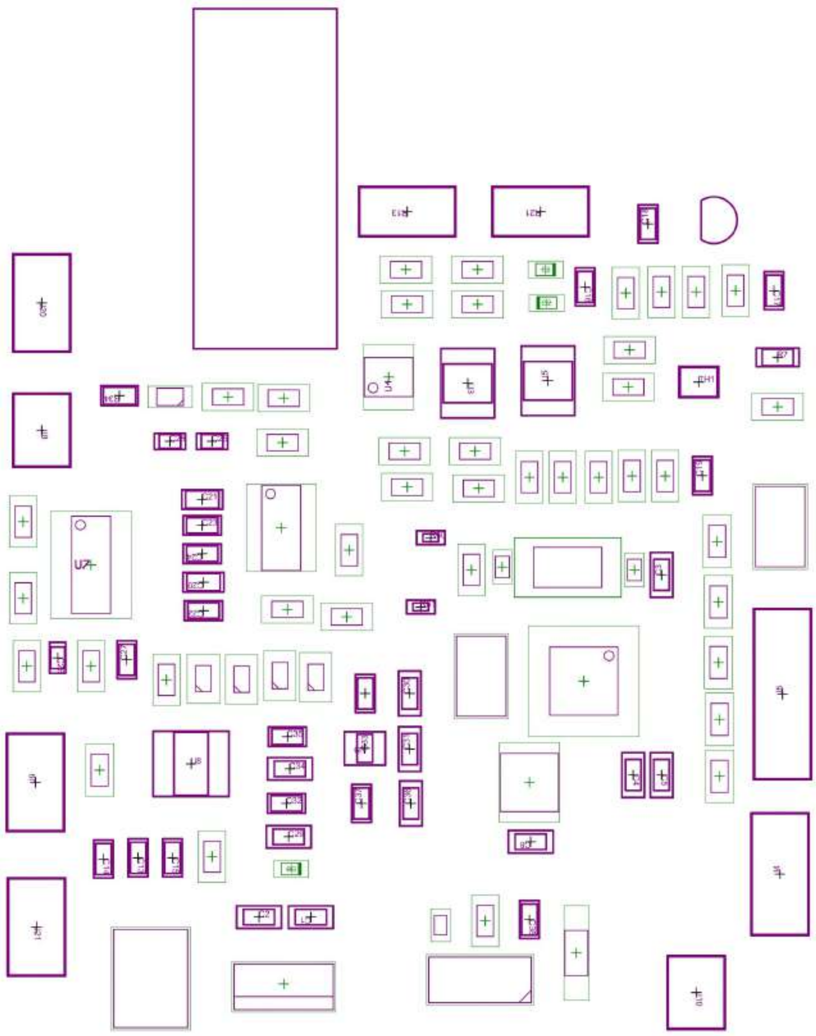


Figure 16: Mechanical Drawing

5. **Top Designator**

The top designator layer displays the reference designators for all components placed on the top side of the PCB. These designators help identify each component during assembly and troubleshooting.

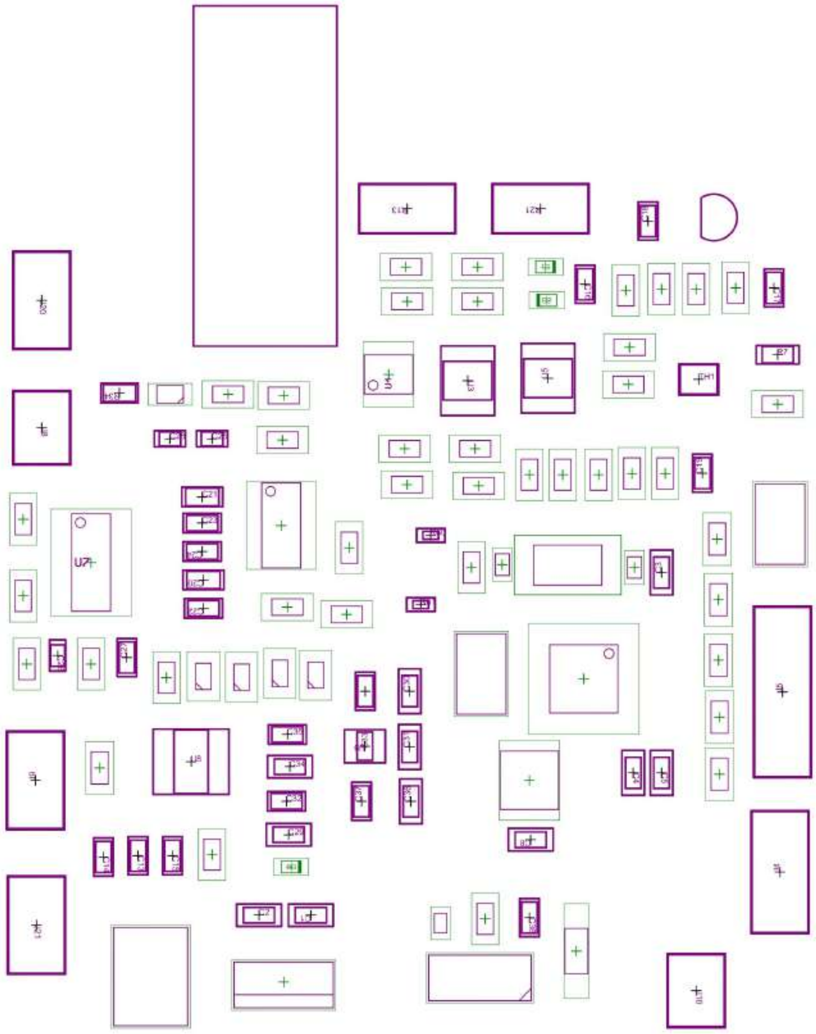


Figure 17: Top Designator

6. **Top Assembly**

The top assembly drawing shows the placement and orientation of all components on the top side of the PCB. It is used as a reference during the assembly process to ensure correct component placement.

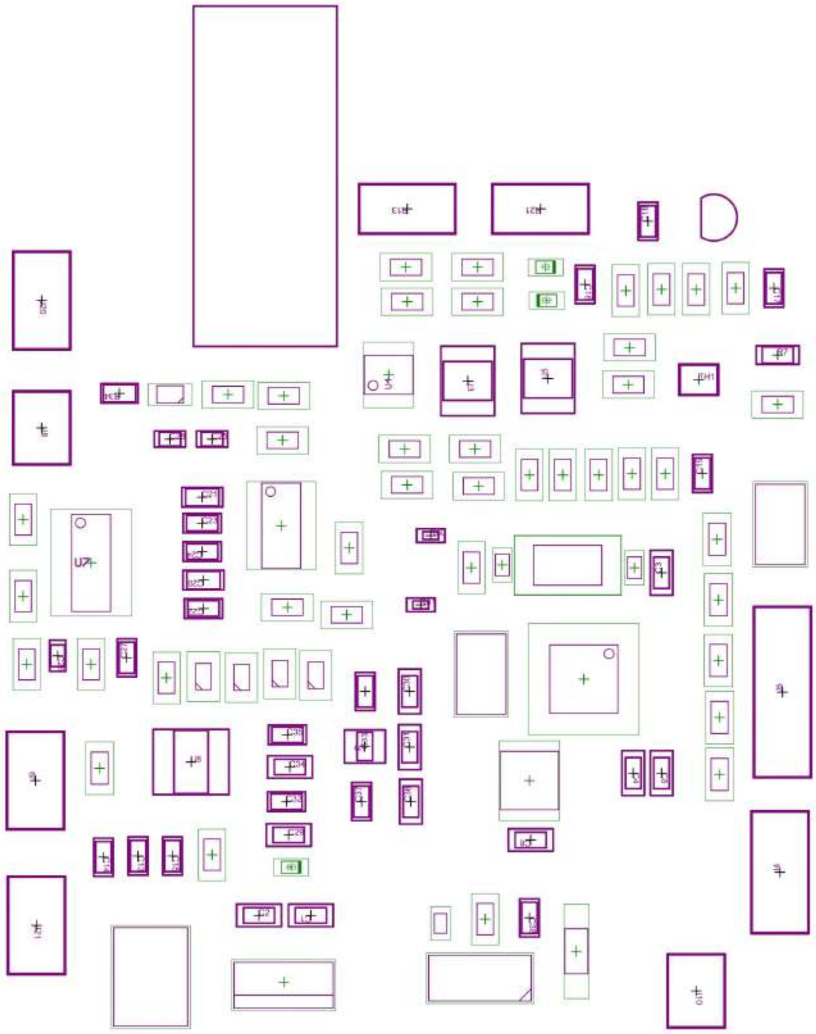


Figure 18: Top Assembly

7. **Top Component Outline**

The top component outline layer provides the exact outlines of all components placed on the top side of the PCB. It helps in verifying that all components fit properly on the board without interference.

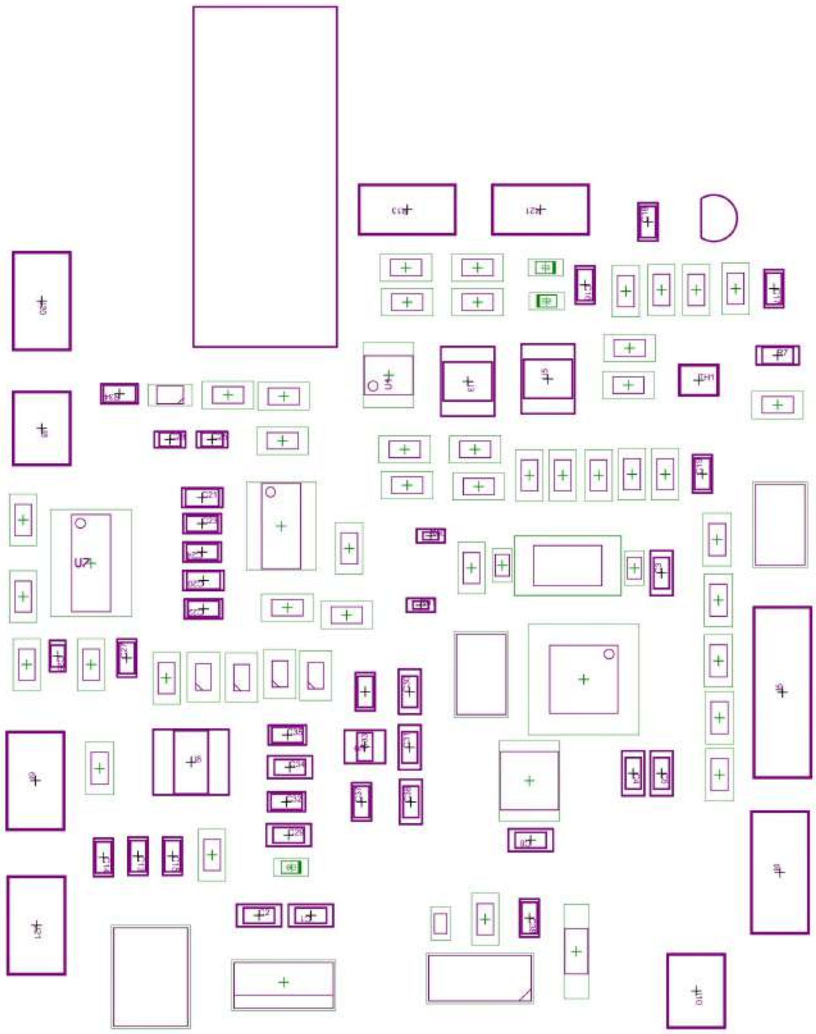


Figure 19: Top Component Outline

8. **Top 3D Body**

The top 3D body layer includes three-dimensional models of all components placed on the top side of the PCB. This layer aids in visualizing the assembled PCB in 3D, ensuring proper component fit and clearance.

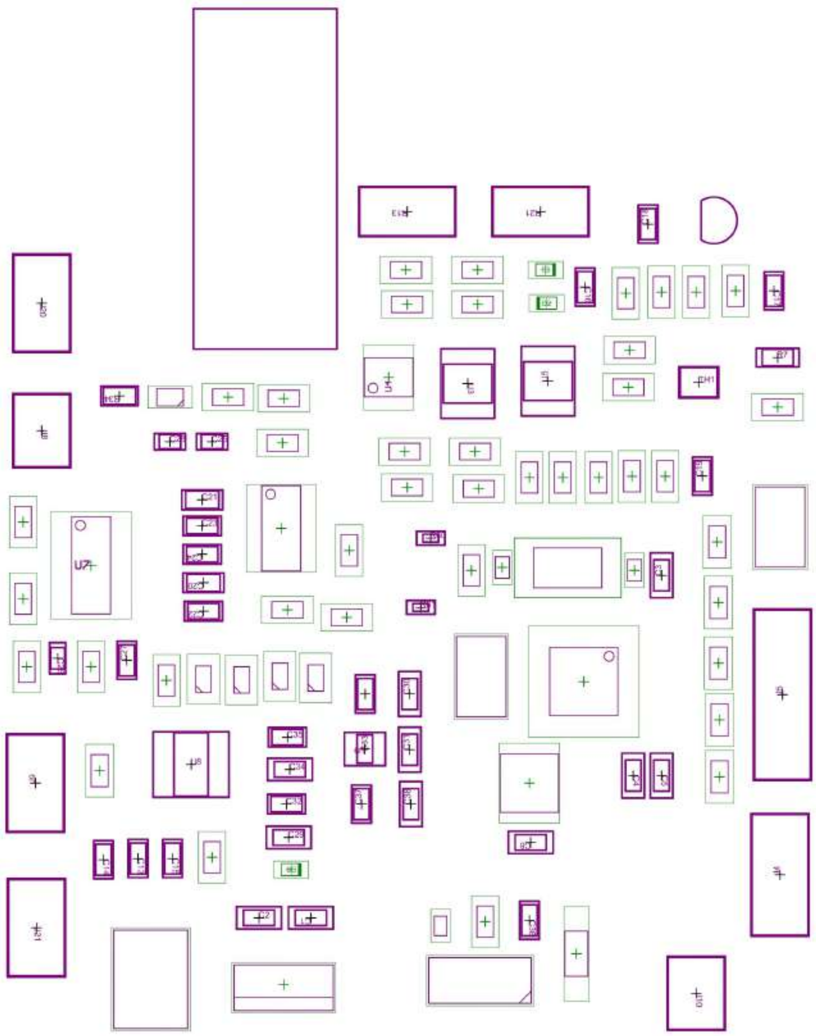


Figure 20: Top 3D Body

9. **Top Courtyard**

The top courtyard layer defines the minimum required space around each component on the top side of the PCB. This ensures there is enough clearance for manufacturing processes and to prevent components from interfering with each other.

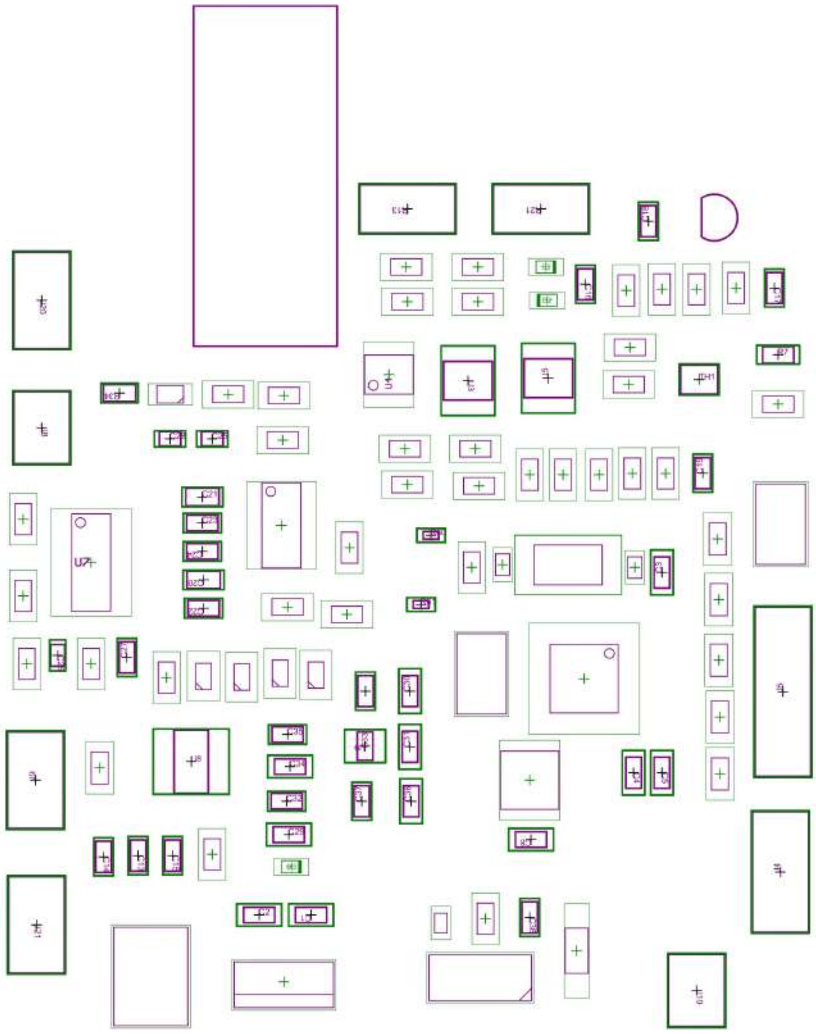


Figure 21: Top Courtyard

10. **Top Component Centre**

The top component center layer marks the exact center points of all components placed on the top side of the PCB. These points are used for automated placement machines during the assembly process.

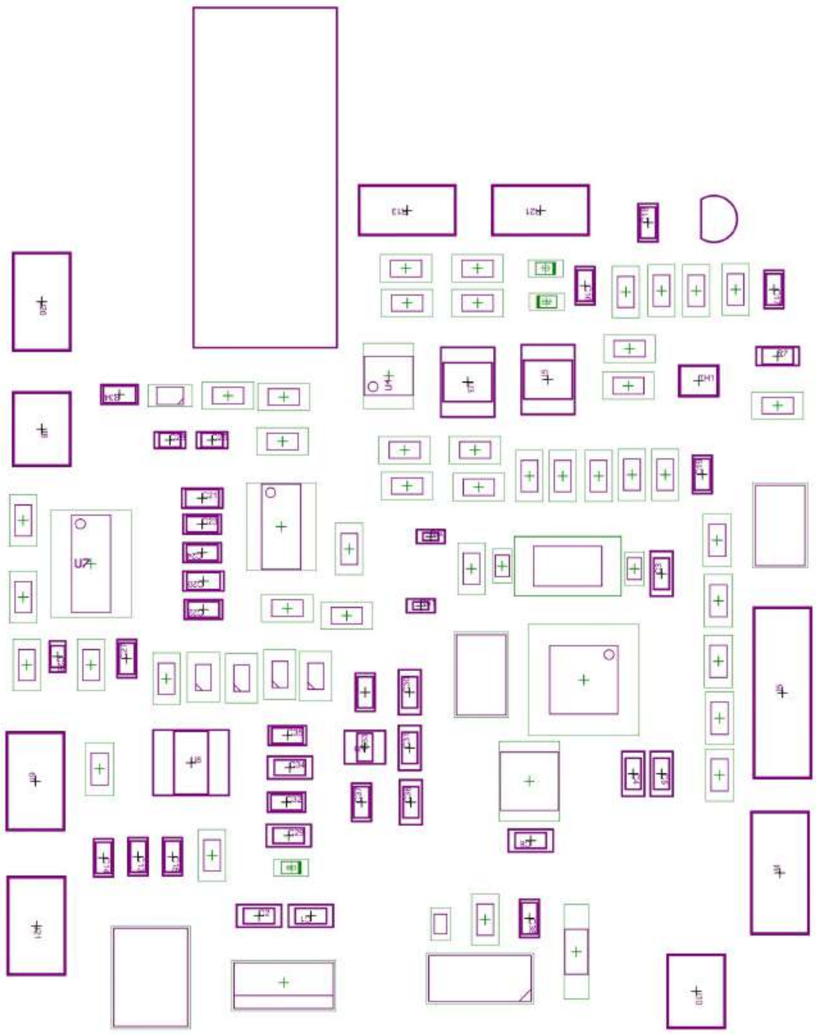


Figure 22: Top Component Centre

11. **Top Pad Master**

The top pad master layer shows all the pads (contact points) on the top side of the PCB where components will be soldered. It is used to create the solder paste stencil for the soldering process.

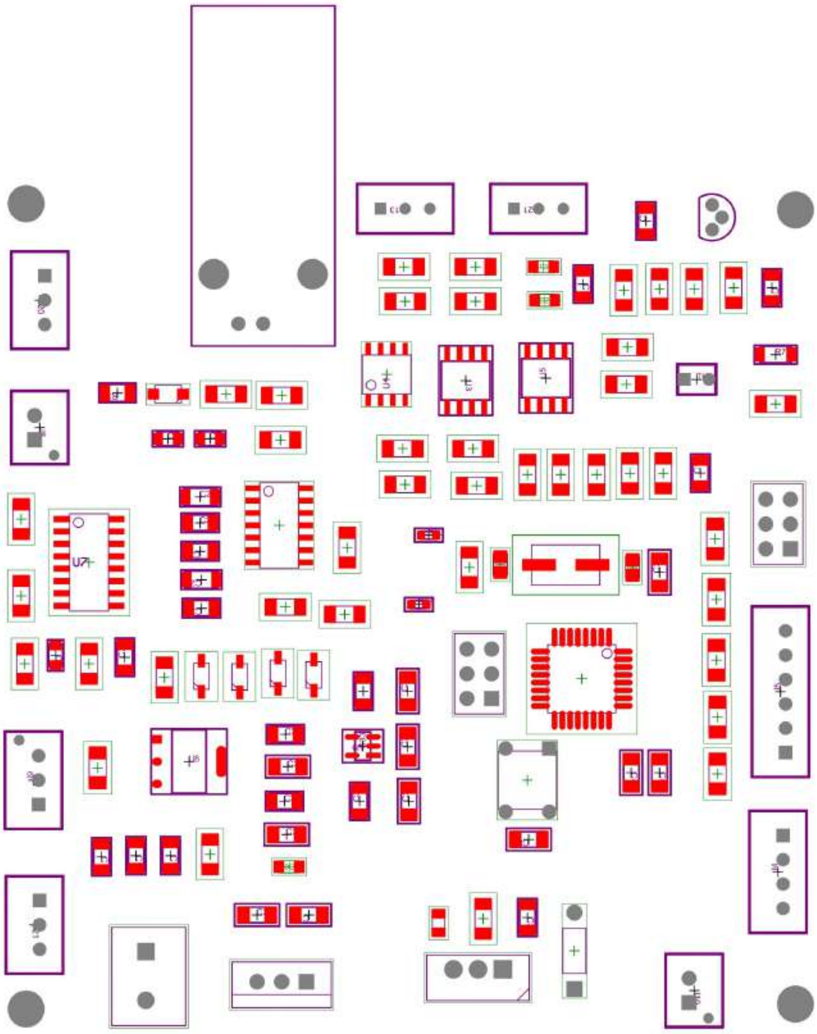


Figure 23: Top Pad Master

12. **Bottom Pad Master**

The bottom pad master layer shows all the pads (contact points) on the bottom side of the PCB where components will be soldered. It is used to create the solder paste stencil for the soldering process on the bottom side.

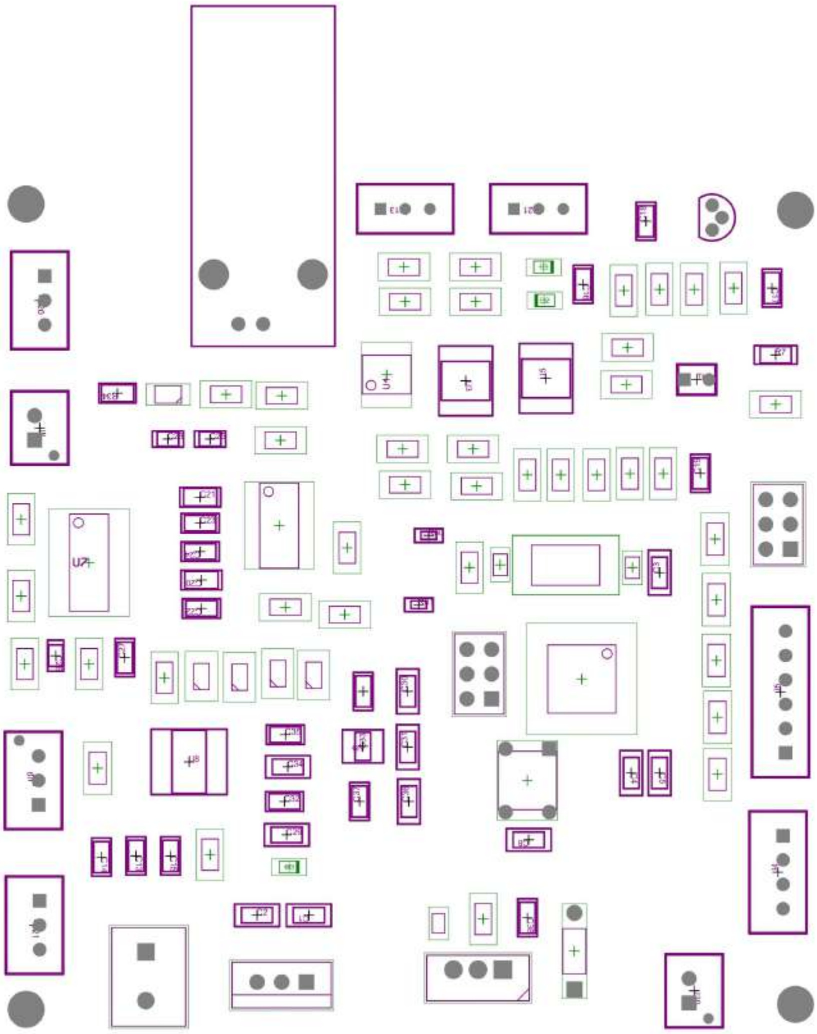


Figure 24: Bottom Pad Master

13. **Drill Drawing for Top-Bottom Layer**

The drill drawing for the top-bottom layer provides the locations and sizes of all holes to be drilled in the PCB, including vias, mounting holes, and component lead holes. It is used by the manufacturer to create the holes accurately.

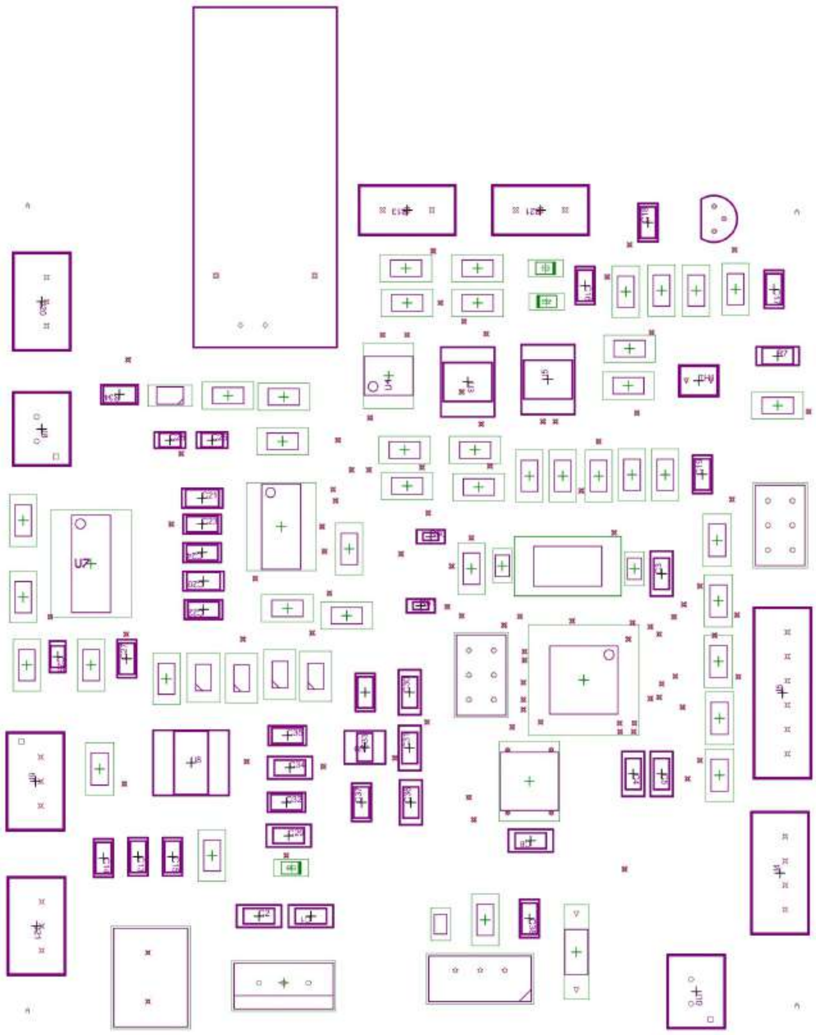


Figure 25: Drill Drawing for Top-Bottom Layer

14. **Drill Guide for Top-Bottom Layer**

The drill guide for the top-bottom layer provides detailed instructions and information on the drilling process, ensuring all holes are drilled correctly. It includes drill sizes and the sequence of drilling operations.

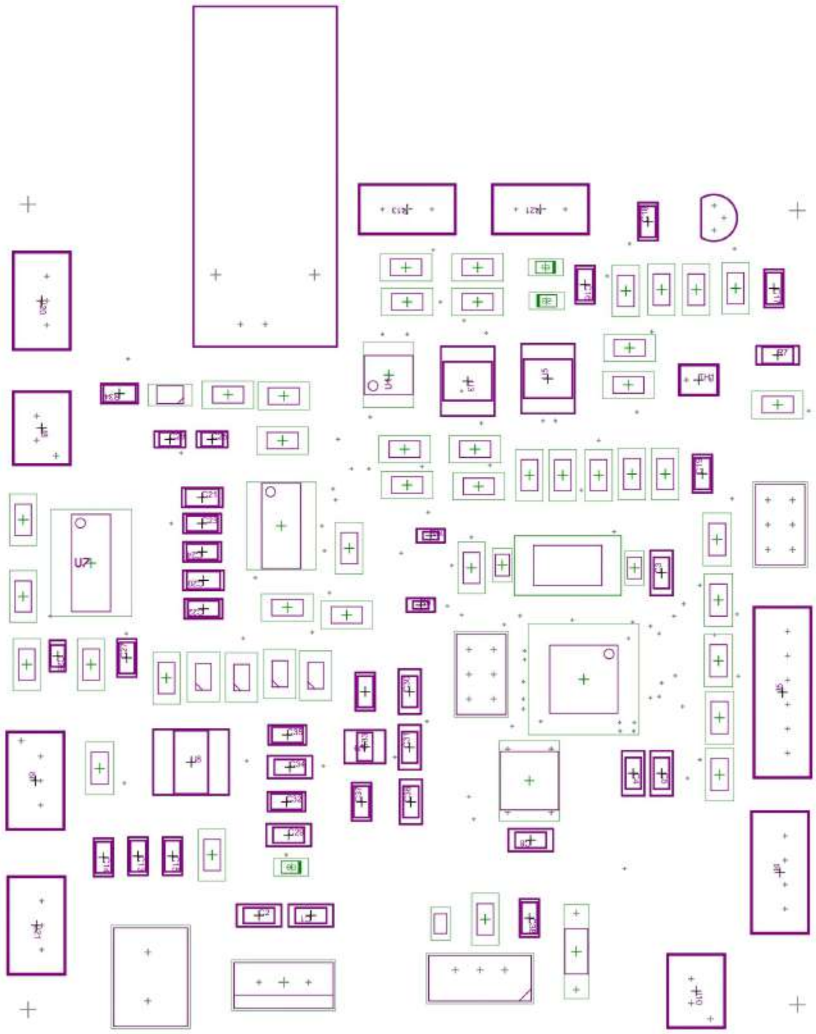


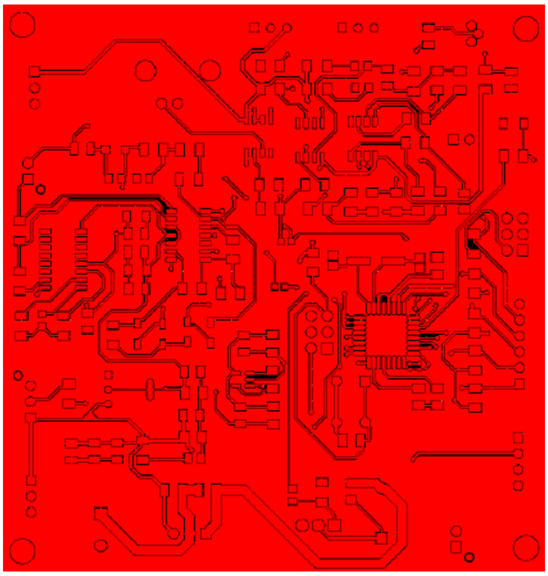
Figure 26: Drill Guide for Top-Bottom Layer

1.8.3 Gerber Files

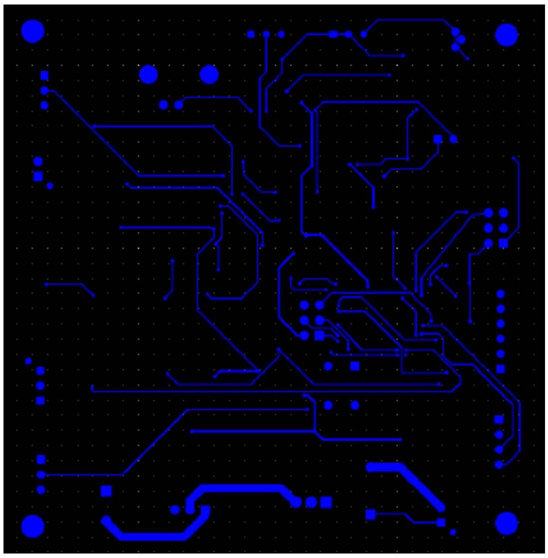
Gerber files are essential in the PCB manufacturing process as they provide a standardized format for describing PCB designs to fabrication facilities. They are used to convey detailed information about the PCB layout, including copper traces, drill holes, component placements, solder masks, and more. Each Gerber file corresponds to a specific layer or aspect of the PCB design and is crucial for accurately translating the designer's intent into physical PCBs. Without Gerber files, manufacturers would lack the precise instructions needed to produce PCBs to the exact specifications required by the designer.

1. **Copper Signal Bottom/Top**

   These Gerber files depict the copper traces and pads on the top and bottom layers of the PCB. They show the conductive pathways that connect components and provide electrical connectivity across the board operations.



(a) Copper Signal Top



(b) Copper Signal Bottom

Figure 27: Copper Signal Bottom/Top

2. **Legend Bottom/Top**

   The legend files indicate component outlines, reference designators, and other textual information needed for assembly and identification of components on the PCB. Since all the components are on the top, only legend top file is shown below.

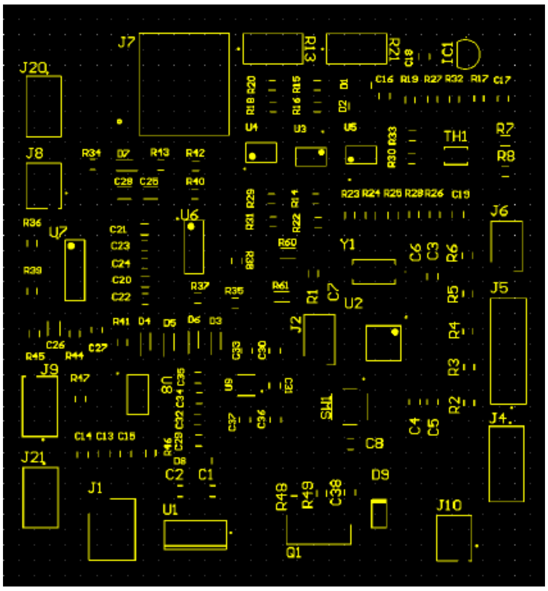


Figure 28: Legend Top

3. **NPTH Drill**

   Non-plated through-hole (NPTH) drill files specify locations and sizes of holes that do not require electrical connection between layers, such as mounting holes or test points. Since there are no non-plated through-holes the file is not shown.

4. **Paste Top/Bottom**

   These files define where solder paste should be applied during the PCB assembly process, aiding in the precise deposition of solder for surface-mount components. Since all the components are on the top, only top file is shown below.

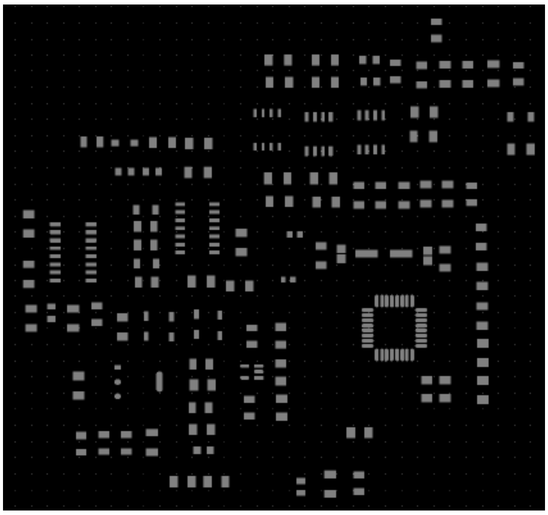


Figure 29: Paste Top

5. **Profile**

   The profile file outlines the outer dimensions and board outline, including cutouts and cut lines, ensuring the PCB is manufactured to the correct shape and size. Here, only the outer margin is cut-off.

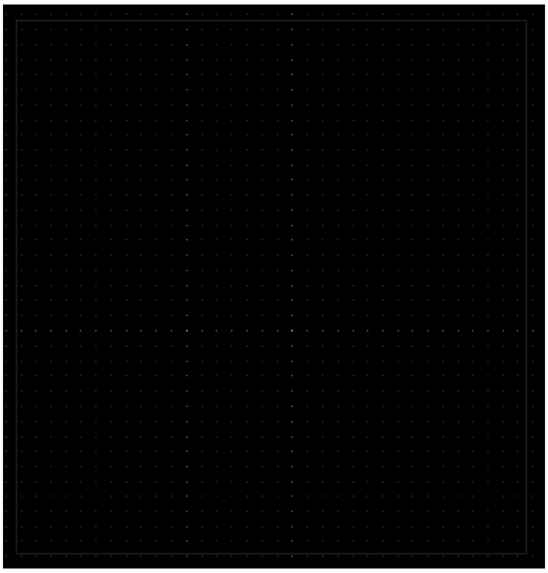


Figure 30: Profile

6. **PTH Drill**

   Plated through-hole (PTH) drill files specify locations and sizes of holes that require electrical connection between layers, typically for through-hole components.

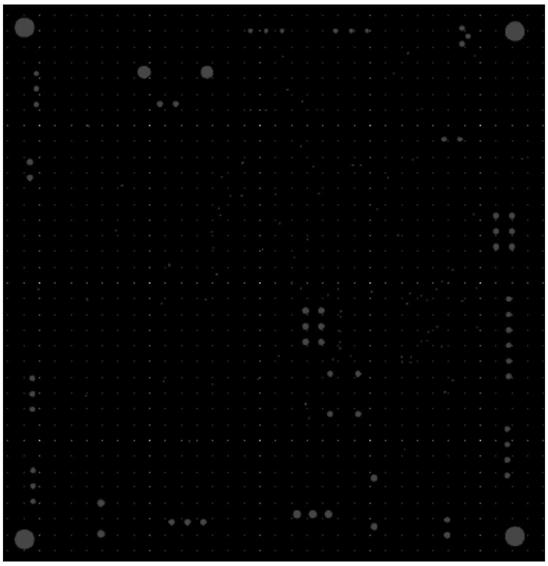
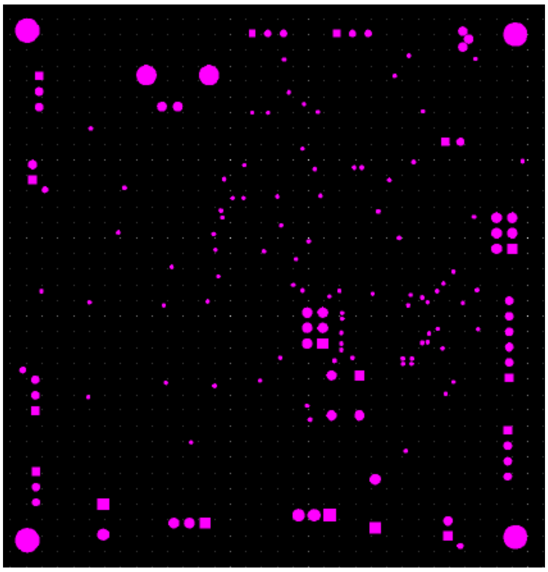


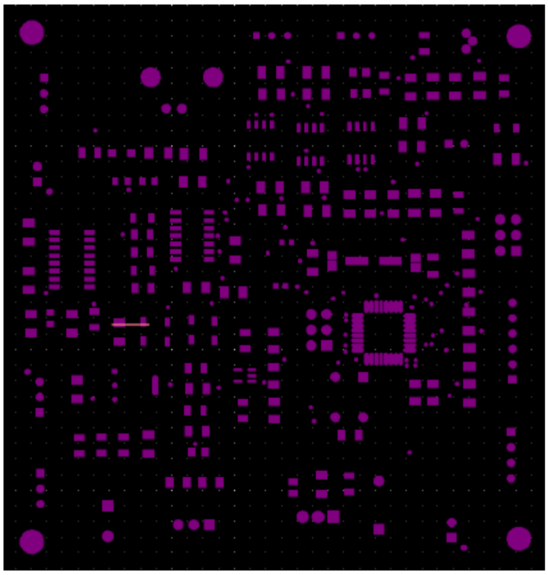
Figure 31: PTH Drill

7. **Soldermask Top/Bottom**

   These files define areas on the PCB where solder mask should be applied. Solder mask protects copper traces from oxidation and unintended solder bridges during assembly.



(a) Soldermask Bottom



(b) Soldermask Top

Figure 32: Soldermask Bottom/Top

1.8.4 NC drill File

The NC (Numerically Controlled) drill file is a critical component of PCB manufacturing, detailing the locations and sizes of all drill holes required in the PCB design. Unlike Gerber files which describe the copper traces and layers, the NC drill file specifically focuses on the drilling aspect of the PCB fabrication process. It provides precise coordinates and diameters for both plated through-holes (PTH) and non-plated through-holes (NPTH) based on the design requirements. This file guides the CNC (Computer Numerical Control) drilling machines in creating holes for mounting components, creating vias, and providing electrical connections between different layers of the PCB. Ensuring accuracy in the NC drill file is crucial as it directly impacts the alignment, functionality, and overall quality of the printed circuit board during assembly and operation.

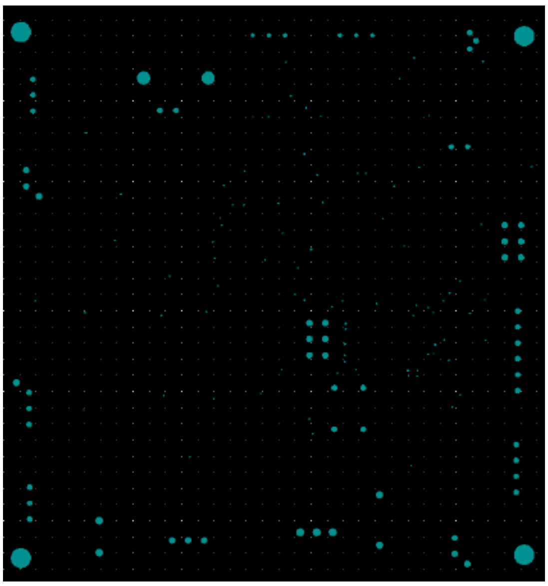


Figure 33: NC Drill Drawing

1.8.5 Bill of Materials

Comment	Description	Designator	Footprint	LibRef	Quantity
10uF	General Purpose Ceramic Capacitor, 1206, 10uF, 10%, X5R, 15%, 25V, General Purpose Ceramic Capacitor, 1206, 10uF, 10%, X5R, 15%, 16V	C1, C2, C3, C4, C5, C29, C30, C31, C34, C36	FP-1206-L_3_2_0_2W_1_6_0-IPC_B	CMP-2000-05863-2, CMP-04427-000082-1	10
22pF		C6, C7	CAPC2013X115X51NL20T26	CMP-2007-04866-1	2
0.1uF	General Purpose Ceramic Capacitor, 1206, 100nF, 10%, X7R, 15%, 10V	C8	FP-1206-L_3_2_0_2W_1_6_0-IPC_B	CMP-1037-04173-2	1
100nF	General Purpose Ceramic Capacitor, 1206, 100nF, 5%, X7R, 15%, 100V	C13, C14, C15, C16, C17, C18, C19, C22, C23, C24, C27, C32, C33, C35, C37, C38	FP-1206-L_3_2_0_2W_1_6_0-IPC_C	CMP-04427-013576-1	16
4.7uF	General Purpose Ceramic Capacitor, 1206, 4.7uF, 10%, X5R, 15%, 25V	C20, C21	FP-1206-L_3_2_0_2W_1_6_0-MFG	CMP-2000-05864-2	2
1uF	General Purpose Ceramic Capacitor, 0805, 1uF, 10%, X7R, 15%, 25V	C25, C26, C28	FP-0805-L_2_01_0_2W_1_25-MFG	CMP-2007-04692-2	3
150080RS75000		D1, D2, D8	WL-SMCW_0805_150080xx75000	CMP-1426-00011-2	3
1N4148W	Diode	D3, D4, D5, D6	SOD3716X125N	1N4148W	4
1N5819W	Schottky Diode	D7	SODFL3618X110N	1N5819W	1
1N581940	Schottky Barrier Rectifier, 1 A, 20 V, -65 to 125 degC, 2-Pin DO41, RoHS, Tape and Reel	D9	DIOD-DO-41-P-2	CMP-2000-05521-1	1
TL431ACZ-AP	Integrated Circuit	IC1	TO-92_BULK	TL431ACZ-AP	1
691137710002	Connector	J1	SHDR2W70P0X500_1X2_1000X750X1160P	691137710002	1
10-89-7062	Connector	J2, J6	HDRV6W87P254_2X3_775X508X864P	10-89-7062	2
B4B-XH-A(LF)(SN)	CONN HEADER VERT 4POS 2.5MM	J4	FP-B4B-XH-A_LF_SNMFG	CMP-17439-000250-1	1
B6B-XH-A(LF)(SN)	CONN HEADER VERT 6POS 2.5MM	J5	FP-B6B-XH-A_LF_SNMFG	CMP-17439-000076-1	1
5227661-1	Connector	J7	5227161-1	5227661-1	1
B2B-XH-AM(LF)(SN)	CONN HEADER VERT 2POS 2.5MM	J8, J10	FP-B2B-XH-AM_LF_SNMFG	CMP-17439-000037-1	2
B3B-XH-AM(LF)(SN)(P)	CONN HEADER VERT 3POS 2.5MM	J9	FP-B3B-XH-AM_LF_SN_P-MFG	CMP-17439-000213-1	1
B3B-XH-A(LF)(SN)	CONN HEADER VERT 3POS 2.5MM	J20, J21	FP-B3B-XH-A_LF_SNMFG	CMP-17439-000014-3	2
IRF540N	MOSFET (N-Channel)	Q1	TO254P469X1042X1967-3P	IRF540N	1
10k	Chip Resistor, 10 KOhm, +/- 1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel	R1, R18, R19, R20, R22, R23, R24, R25, R26, R27, R28, R29, R30, R32, R49	RESC3116X65X40ML10T20	CMP-1014-00623-2	15
4.7K	Chip Resistor, 4.7 KOhm, +/- 1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel	R2, R3, R4, R5, R6, R47	RESC3216X60X45ML15T20	CMP-2100-03618-1	6
2.2K	RES 2.20K OHM 1/4W .1% SMD 1206	R7	FP-RT1206-MFG	CMP-2003-04098-2	1
1K	Chip Resistor, 1 KOhm, +/- 1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel, Chip Resistor, 10 KOhm, +/1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel	R8, R14, R15, R16, R17	RESC3116X65X40ML10T20	CMP-2100-03678-1, CMP-1014-00623-2	5
3296W-1-504LF	TRIMMER 500K OHM 0.5W PC PIN TOP	R13, R21	FP-3296W-MFG	CMP-2000-05583-2	2
20K	Chip Resistor, 10 KOhm, +/- 1%, 0.25 W, -55 to 155 degC,	R31, R33	RESC3116X65X40ML10T20	CMP-1014-00623-2	2

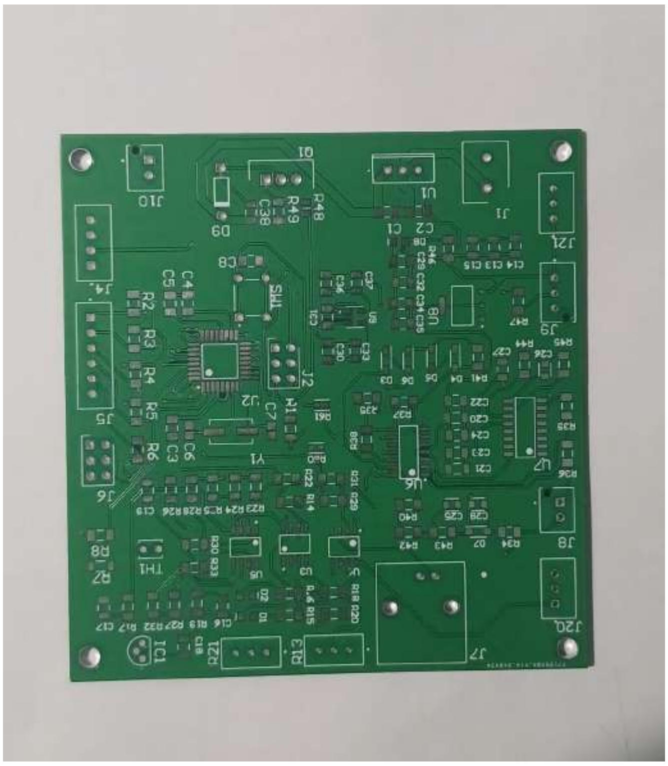
Figure 34: Bill of Materials - Page 1

	1206 (3216 Metric), RoHS, Tape and Reel				
1M-1%	RES Thick Film, 1MΩ, 1%, 0.25W, 100ppm/ °C, 1206	R34	FP-CRCW1206-e3-MFG	CMP-2003-01906-2	1
5.6K-1%	Chip Resistor, 10 KOhm, +/- 1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel	R35, R46	RESC3116X65X40ML10 T20	CMP-1014-00623-2	2
6.8K-1%	Chip Resistor, 10 KOhm, +/- 1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel	R36	RESC3116X65X40ML10 T20	CMP-1014-00623-2	1
10K-1%	Chip Resistor, 10 KOhm, +/- 1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel	R37, R38, R40, R41, R42, R43, R45	RESC3116X65X40ML10 T20	CMP-1014-00623-2	7
100K-1%	Chip Resistor, 100 KOhm, +/- 1%, 0.25 W, -55 to 155 degC, 1206 (3216 Metric), RoHS, Tape and Reel	R39, R44	RESC3116X65X40ML10 T20	CMP-2100-03674-1	2
100Ohm	Resistor	R48	RESC2013X65N	0805W8F1000T5E	1
0 ohm	Anti-Sulfurated Chip Resistors Automotive Grade 1.5kΩ 1% 100ppm/°C 0603	R60, R61	FP-AF0603-IPC_B	CMP-03412-043191-1	2
1825910-6	Tact Switch, SPST-NO, 0.05 A, -35 to 85 degC, 4-Pin THD, RoHS, Bulk	SW1	TECO-1825910-6_V	CMP-1684-00021-1	1
NTCLE100E3104JB0	NTC THERMISTOR 100K OHM 5% BEAD	TH1	FP-NTCLE100E3104JB0MFG	CMP-02404-000021-1	1
LM7805CT	Positive Voltage Regulator, 5 V, 1 A, 40 to 125 degC, 3-Pin TO-220, RoHS, Tube	U1	FAIR-TO-220-3	CMP-2000-04938-1	1
ATmega328P-AU	8-bit AVR Microcontroller, 32KB Flash, 1KB EEPROM, 2KB SRAM, 32-pin TQFP, Industrial Grade (-40°C to 85°C)	U2	32A_M	CMP-0095-00269-2	1
LM358DR	IC OPAMP GP 2 CIRCUIT 8SOIC	U3, U5	FP-D0008A-MFG	CMP-1685-00009-2	2
TLC4502IDR	Advanced LinEPIC SelfCalibrating Precision Dual Operational Amplifier, 4 to 6 V, -40 to 125 degC, 8-pin SOIC (D8), Green (RoHS & no Sb/Br)	U4	D0008A_L	CMP-0272-00725-3	1
LMV324IDR	Quad Low-Voltage Railto-Rail Output Operational Amplifier, 2.7 to 5.5 V, -40 to 125 degC, 14-Pin SOIC (D), Green (RoHS & no Sb/ Br), Tape and Reel	U6	D0014A_V	CMP-1685-00010-1	1
CD4060BM	CMOS 14-Stage Ripple Carry Binary Counter / Divider and Oscillator, 3 to 18 V, -55 to 125 degC, 16-Pin SOIC (D), Green (RoHS & no Sb/ Br), Tube	U7	D0016A_M	CMP-1086-00003-4	1
AMS1117-3.3	LDO Voltage Regulators 1A, 3.3V	U8	FP-AMS1117-IPC_C	CMP-209535-000001-1	1
TPS60400DBVR	IC REG CHARG PUMP INV 60MA SOT23	U9	FP-DBV0005A-IPC_B	CMP-0391-00019-3	1
ECS-160-20-3X-TR	Crystal or Oscillator	Y1	ECS160203XTR	ECS-160-20-3X-TR	1

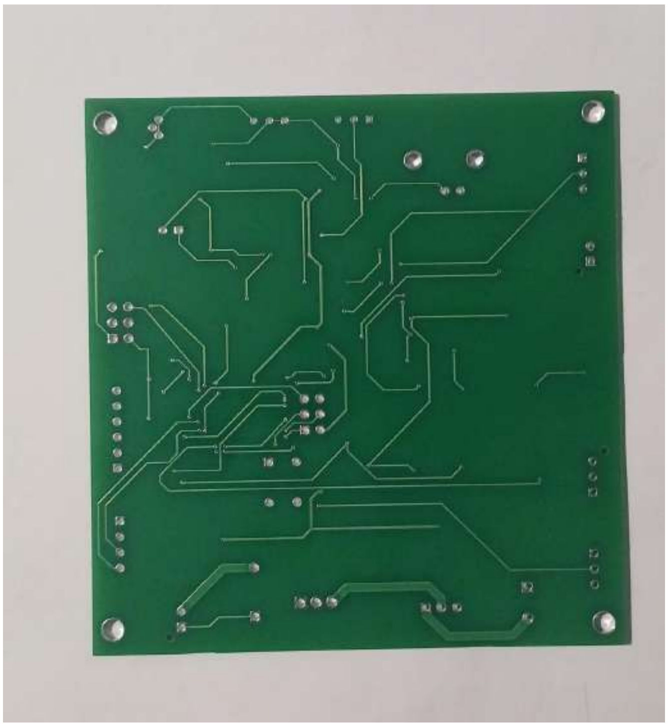
Figure 35: Bill of Materials - Page 2

1.9 Photographs of the PCB

1.9.1 Photographs of the Bare PCB



(a) Unsoldered PCB Top



(b) Unsoldered PCB Bottom

Figure 36: User Interface Enclosure - Back

1.9.2 Photographs of the Soldered PCB

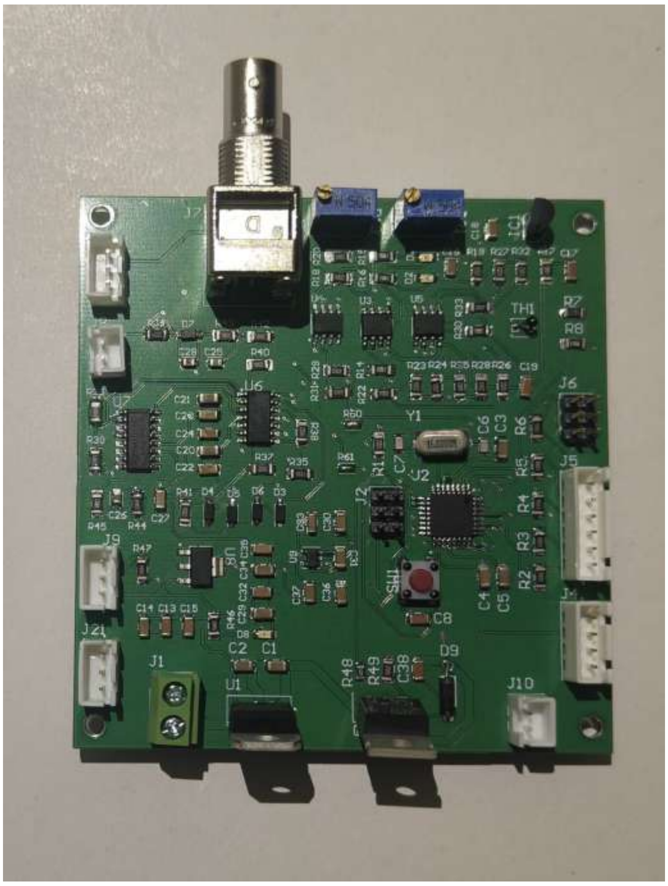


Figure 37: Soldered PCB

1.10 Solidwork Design

1.10.1 Sub-assemblies and their specifications

Components List

1. **LCD Display 16X4**
   - Specifications:
     - Voltage Requirements: 5V DC
     - Supply current : 1.5mA (at 5V supply voltage)
     - Type: Liquid Crystal Display (LCD) with 16 characters per line and 4 lines.
     - Resolution: Typically 5x8 dot matrix character resolution per character space.
     - Interface: Uses I2C (Inter-Integrated Circuit) or parallel interface for communication.
     - Backlight: Equipped with a backlight for visibility in various lighting conditions.
     - Dimensions: 87.0 x 60.0 mm
2. **I2C Module**
   - Specifications:
     - Operating Voltage: 5V DC
     - I2C Control: PCF8574
     - Maximum Modules: 8 on a single I2C bus
     - I2C Address: 0x20 0x27 (default address 0x20, adjustable via onboard jumper pins)
3. **Piezo buzzer**
   - Specifications:
     - Buzzer Type: Piezoelectric sound pressure level 95 dB
     - Operating Voltage: 3-24VDC
     - Operating current : 10 mA - 50 mA
     - Alarm Diameter: 22mm / 0.86”
     - Alarm Height: 10mm / 0.39”
     - Mounting Holes Distance: 30mm / 1.18”
4. **Wifi Module**
   - Specifications:
     - Model: ESP-01S
     - Operating Voltage: 3.3V DC (requires a proper 3.3V voltage regulator if using with 5V systems; IO pins are 5V tolerant)
     - WiFi Protocol: 802.11 b/g/n
     - Frequency: Operates at 2.4 GHz frequency band
     - Power Management: Integrated PLLs, regulators, DCXO, and power management units
     - Temperature Sensor: Integrated temperature sensor for monitoring onboard temperature
     - Antenna: Supports antenna diversity for better signal reception
5. **DC motor**
   - Specifications:
     - Model: GA25-370
     - Rated Power: 3.5W
     - Rated Voltage: 12VDC
     - Rated Current: 0.06A
     - Product Type: Brush DC motor
     - Rotations Per Minute (RPM): 50

  - Outer Diameter: 25 mm (Approx.)
  - Shaft Diameter: 3 mm (Approx.)
  - Shaft Length: 10 mm (Approx.)
  - Weight: 96g (Approx.)

6. **On/Off switches**
   - Specifications:
     - Black Mini 3-Pin On/Off Rectangular Rocker Switch SPST
     - Rating: 6A 250VAC / 10A 125VAC

7. **Push buttons**
   - Specifications:
     - Contact Type: Momentary Contact
     - Switch Size (Approx.): 16.5x8mm (WxH)
     - Number of pins: 4

8. **1 PCB**

Selection Criteria for the components

1. **LCD Display 16X4**
   - **Display Size and Resolution:** Provides clear and readable information to users with a 16x4 character configuration.
   - **Power Consumption:** Efficient use of energy resources to prolong device operation.

2. **Piezo Buzzer**
   - **Sound Pressure Level:** Produces a sound pressure level of 95 dB for effective audible alerts.
   - **Voltage Range:** Operates efficiently within a 3-24VDC range.

3. **WiFi Module**
   - **Model and Protocol:** ESP-01S model supporting WiFi 802.11 b/g/n protocol.
   - **Voltage Compatibility:** Operates at 3.3V DC with 5V tolerant IO pins.

4. **DC Motor**
   - **Specifications:** GA25-370 model with 12VDC, 3.5W power rating.
   - **Efficiency:** Runs at 50 RPM with low current draw (0.06A) for energy efficiency.
   - **Torque:** Stall torque of 10.5 kgf.cm and operating torque of 660 gf.cm suitable for rotating a screw conveyor to dispense fish feed.
   - **Stall current:** Stall current of 2A. The motor's requirements can be met without overheating or voltage drops.

5. **Push Buttons**
   - **Contact Type and Size:** Momentary contact, 16.5x8mm (WxH) suitable for user interaction.
   - **Number of Pins:** 4 pins per button, compatible with device control interface.

6. **On/Off Switches**
   - **Rating:** SPST switches rated for 6A 250VAC / 10A 125VAC.
   - **Size and Mounting:** Compact design suitable for integration into the user interface enclosure.

1.10.2 Solidworks Design

The enclosure design for our aquaculture monitoring and feeding system plays a crucial role in ensuring the reliability, durability, and functionality of the device in diverse environmental conditions. This section outlines the detailed design considerations and features for the two main enclosures:

- The Interface Enclosure (Control Unit Housing Enclosure)
- Feeder Enclosure

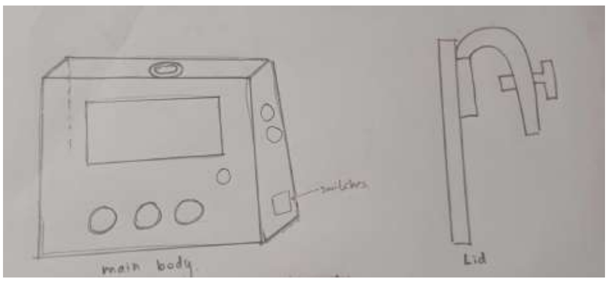
Each enclosure is meticulously designed to house specific components and functionalities essential to the operation and performance of our innovative aquaculture solution.

1) The User Interface Enclosure (Control Unit Housing Enclosure)

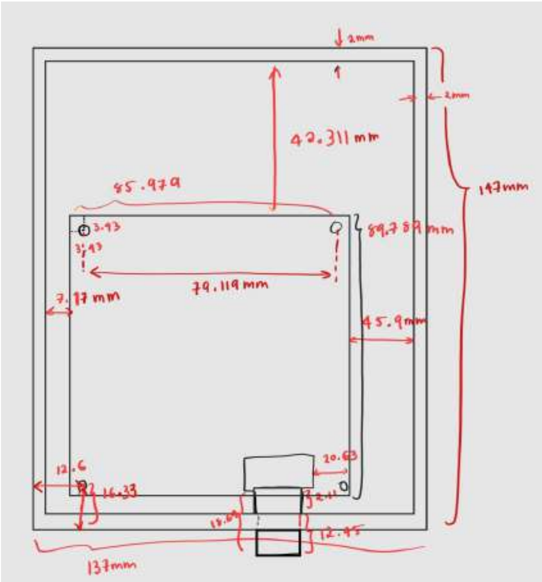
- **Purpose:** The Interface Enclosure serves as the protective housing for the core electronic components and user interface elements of our aquaculture monitoring system. It is designed to withstand environmental challenges while ensuring seamless operation and user interaction.

  The user interface provides a clear display where users can monitor pH, temperature, and TDS values of the tank in real-time. Additionally, it allows users to schedule feeding times and provides manual control options to switch the feeder ON or OFF for dispensing feed as needed. This intuitive interface ensures that users can easily manage and optimize their aquaculture environment for fish health and growth.

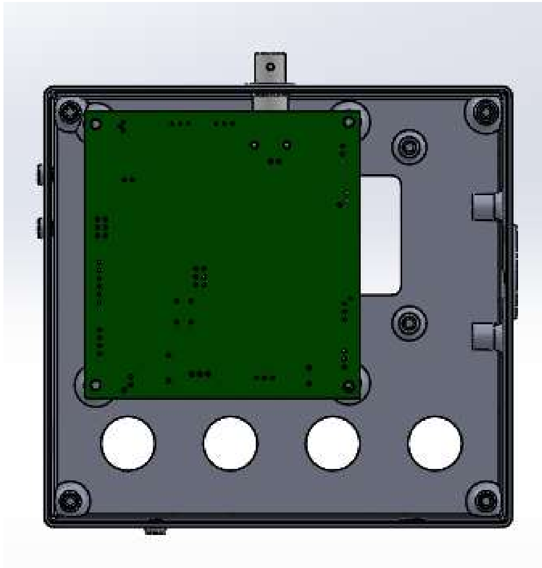
- **Draft Angles:** 1 degree applied to all enclosure walls.
- **Design Considerations:**
  - **Mounting Options:** Provision for secure mounting to ensure stability and accessibility for maintenance tasks. Ability to mount securely onto fish tanks with varying rim thicknesses.
  - **Aesthetic Appeal:** Smooth, uniform surface finish enhances visual appeal.
  - **Dimensional Requirements:** Dimensions are optimized to accommodate all internal components while allowing for efficient wiring and ventilation.
  - **User-friendly Design:** The interface enclosure is designed with intuitive controls and a clear display, ensuring ease of navigation and operation for users.
  - **Compact Design:** The compact enclosure design optimizes space utilization while accommodating all necessary components, minimizing footprint without compromising functionality.
- **Rough sketches**



- **PCB dimension and integration**

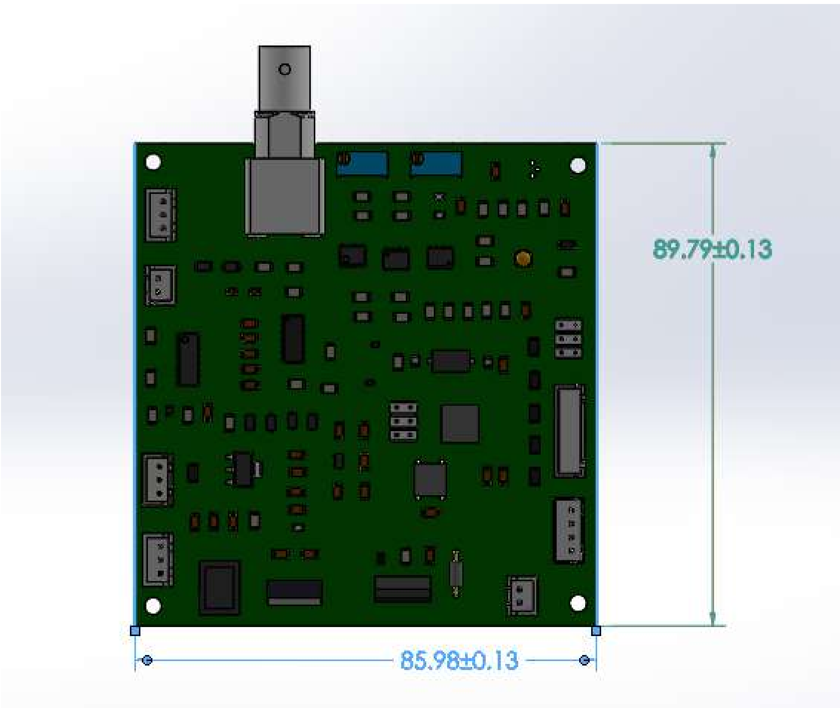


(a) Dimentions for mounting the PCB in enclosure

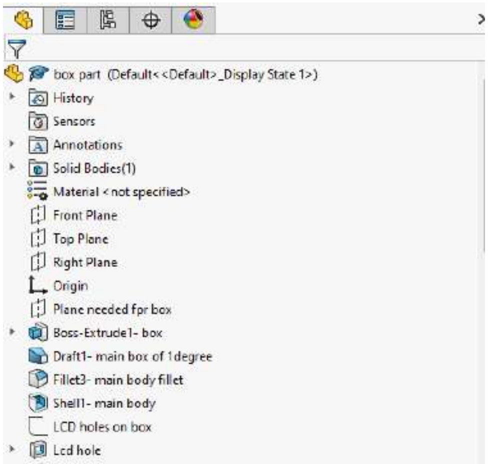


(b) PCB mounted in the enclosure

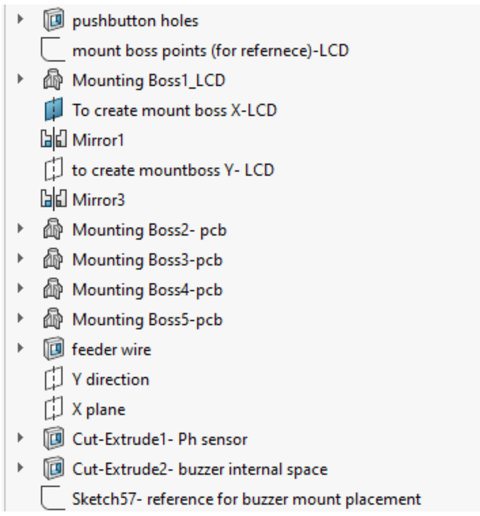
Figure 38: Two Pictures Side by Side



1. **Main body:**
   Model tree of main box



(a) 1



(b) 2

Figure 39: model tree

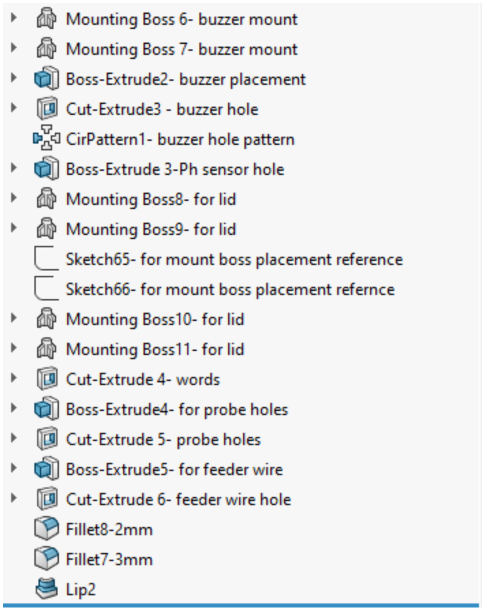


Figure 40: 3

(a) Front face of the main body:

Figure 41: Front Face of the Main Body

- Space for Push Buttons: The front face of the enclosure is designed to include spaces for four push buttons, each with a diameter of 16.5 mm. These buttons serve the following functions:
  - OK/Menu: For selecting menu options and confirming actions.
  - Cancel: For exiting menus or canceling actions.
  - Up/Down: For navigating through menu options and adjusting settings.
  - Manual Control: For manually switching the feeder on and off to dispense feed.
- Space for LCD Display: The front face also includes space for a 16x4 LCD display. This display is used to clearly show the sensed levels of pH, temperature, and TDS, as well as any warning messages. The resolution of the display was considered during selection to ensure it provides clear and readable information to the user.

(b) Sides of the Device:

- Buzzer Space: The sides of the device include a designated space for the buzzer and its mounts. A pattern of holes is provided to allow the alarm noise to be heard clearly outside the enclosure, despite the buzzer being located inside.

Figure 42: Buzzer placement inside the enclosure

- Sensor Probes Wires and Power Supply Inlet :
  - Sensor Probes Wires and wire connecting to the Feeder: There are specific spaces for the wires of the sensor probes and to the feeder to pass through, ensuring a neat and organized arrangement.
  - Power Supply Inlet: The enclosure includes a 12V DC female jack for the power supply inlet, allowing for easy and secure connection to the power source.

Figure 43: Openings for sensor probe wires

Figure 44: 12V DC female ja

- Power ON/ Off switch: Space for a Rotary ON/OFF switch is implemented at the side.

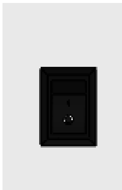


Figure 45: Rotary switch

(c) pH Sensor Consideration: Special consideration has been given to the pH sensor's female jack, which is mounted on the PCB and protrudes outside the enclosure. Ample space has been created to accommodate this component, ensuring it is securely positioned and easily accessible.

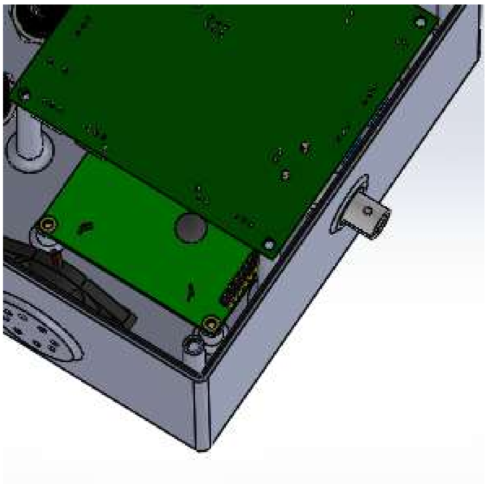


Figure 46: Dimensions for the PH sensors Jack were carefully calculated

(d) Inside the device:

Figure 47: Interior of the enclosure

- PCB Mounts: Secure mounts with precise dimensions to hold the PCB in place.
- Buzzer and LCD Mounts: Dedicated mounts to firmly position the buzzer and LCD display.
- Lid Mounts: Secure mounts to fix the lid tightly to the enclosure.
- Component Placement: Carefully selected dimensions to prevent interference between components.

(e) Assembly:

Figure 48: Assembled front face of the enclosure

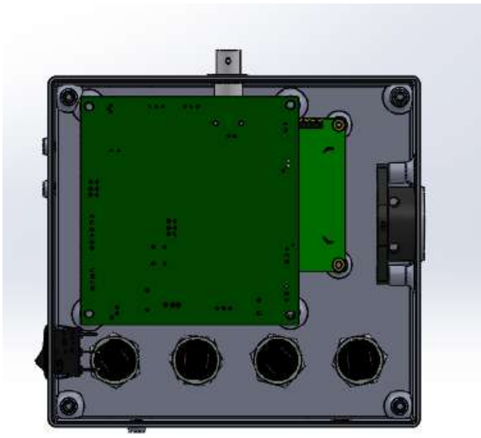
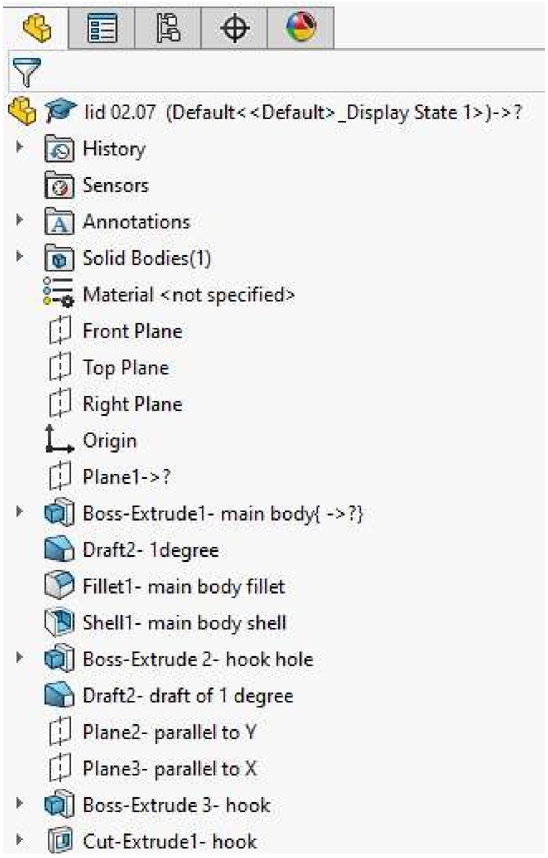
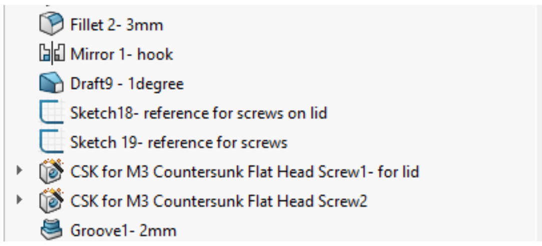


Figure 49: Assembled Interior of Enclosure

(f) **Lid:**
Model tree of lid

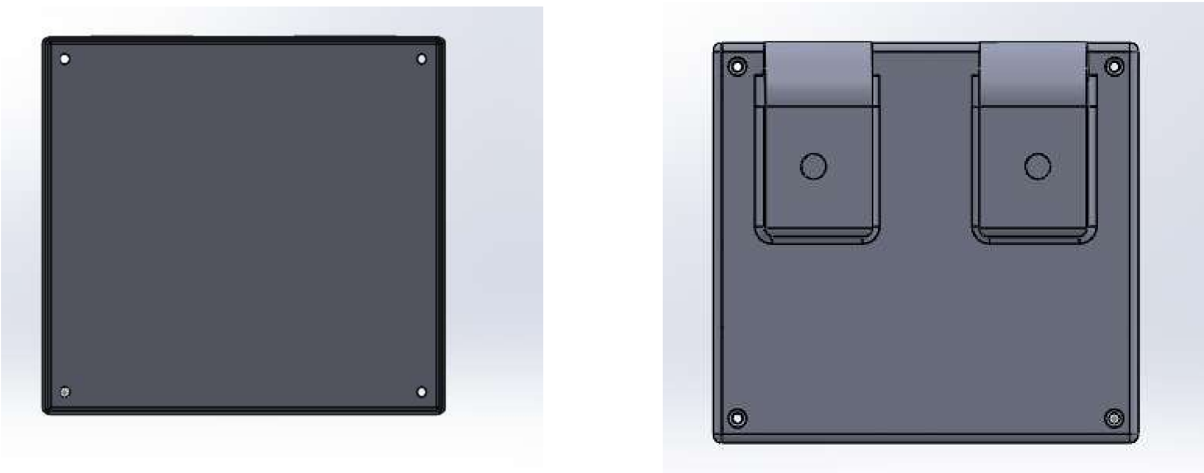


(a) 1



(b) 2

Figure 50: Lid of the Interface enclosure



(a) Inside of Lid

(b) Outside of lid

Figure 51: Lid of the Interface enclosure

(g) Assembly of the device:

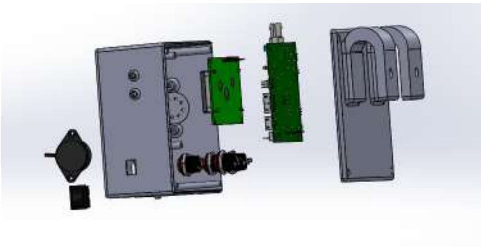
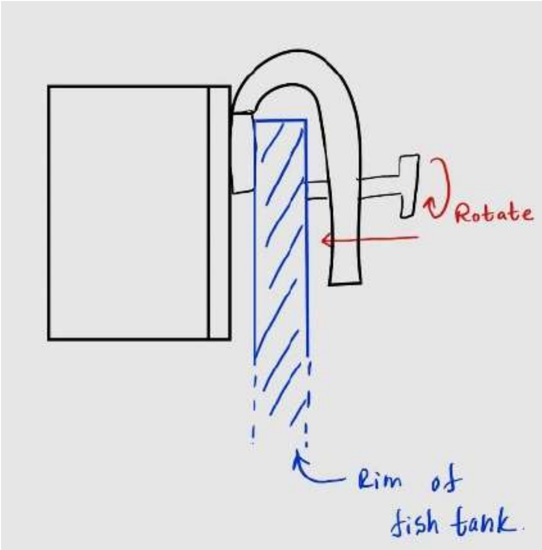
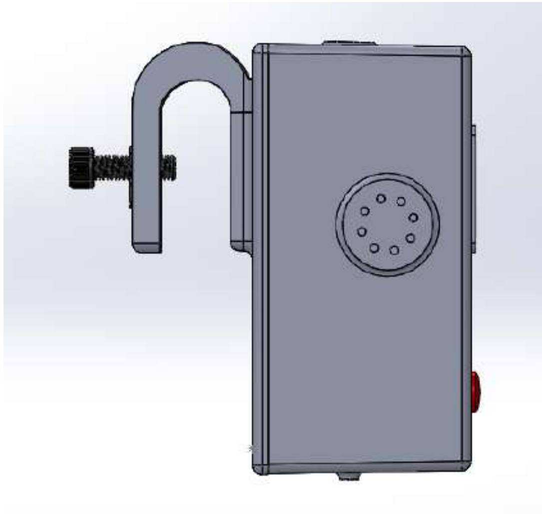


Figure 52: Assemble of the Interface enclosure

We will secure the interface enclosure to the rim of the fish tank using hooks on the lid. Adjustable screws will be used to firmly hold the device in place, ensuring stability and easy access for maintenance when needed.



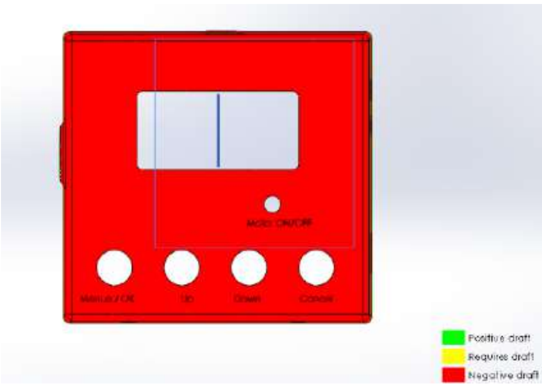
(a) How the device will be fixed to the tank rim.



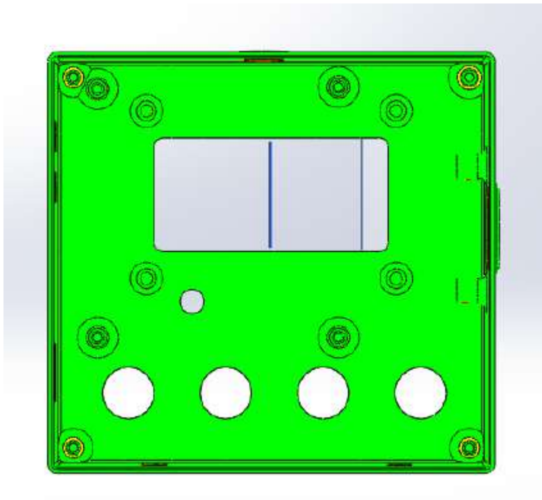
(b) Adjustable screws on the lid hooks used for fixing the device on tto the rim of the tank

Figure 53: Lid of the Interface enclosure

(h) Moldability:
1 degree applied to all enclosure walls.

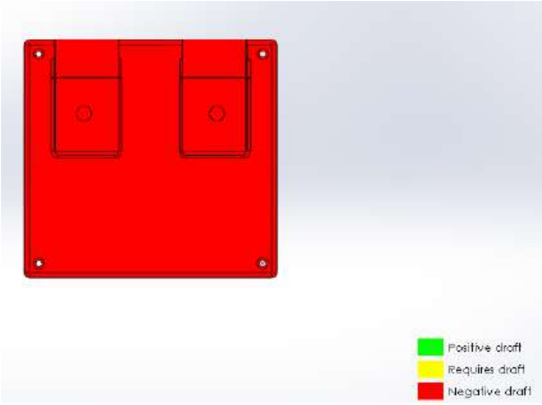


(a) Draft analysis of main body (1)

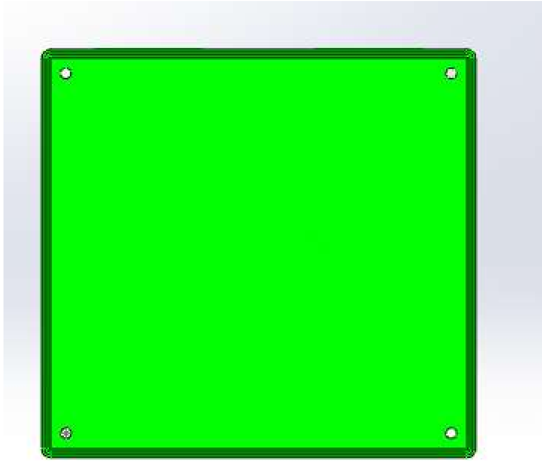


(b) Draft analysis of main body (2)

Figure 54: Draft analysis



(a) Draft analysis of lid (1)



(b) Draft analysis of lid (2)

Figure 55: Draft analysis

2) Feeder

- **Purpose:**
  - The Feeder Enclosure is designed to house the mechanism responsible for storing and dispensing fish feed at scheduled intervals. It ensures efficient feed management and distribution, crucial for the health and growth of aquatic organisms in our aquaculture system.

  q1

- **Design Considerations:**
  - **Mounting Options:**
    - Securely mount the DC motor and connected screw conveyor to ensure stable operation, especially during use.
    - Ensure the feed reservoir is mounted securely, capable of withstanding its weight without compromising device integrity.
    - Design the feeder to securely mount onto fish tanks with varying rim thicknesses, accommodating different tank configurations for optimal stability and functionality.
  - **Aesthetic Appeal:** Smooth, uniform surface finish enhances visual appeal.
  - **Dimensional Requirements:** Dimensions are optimized to accommodate all internal components while allowing for efficient wiring and ventilation.
  - **User-friendly Design:** The interface enclosure is designed with intuitive controls and a clear display, ensuring ease of navigation and operation for users.
  - **Compact Design:** The compact enclosure design optimizes space utilization while accommodating all necessary components, minimizing footprint without compromising functionality.

- **Rough sketches**

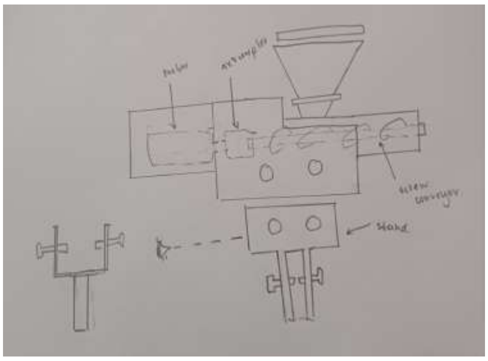


Figure 56: Sketch of the feeder

Parts of the Feeder:

- Screw Conveyor
- Main Body (housing the screw conveyor and the DC motor)
- Stand
- Feed Reservoir

a) Screw conveyor
How a screw conveyor work: The screw conveyor dispenses food through a rotating helical screw inside a cylindrical housing. When the DC motor is activated, it drives the screw, which moves the feed from the feed reservoir towards the dispensing end. The rotation of the screw ensures a consistent and controlled flow of feed, effectively distributing it to the fish tank.

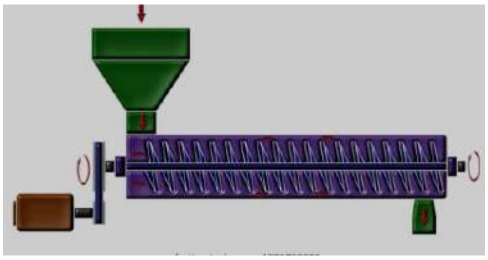


Figure 57: Mechanism of screw conveyor

The screw conveyor is a critical component responsible for dispensing feed when the DC motor is activated. It rotates via a coupler connected to the DC motor, facilitating efficient feed distribution within the main body. Careful consideration of dimensions ensures optimal performance, allowing for effective dispensation of feed throughout the aquaculture system.

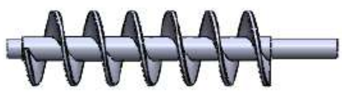


Figure 58: Screw conveyor

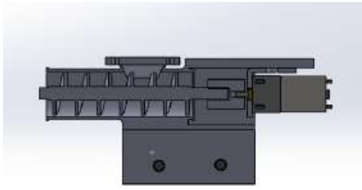


Figure 59: Screw conveyor implementation

Model tree of screw conveyor

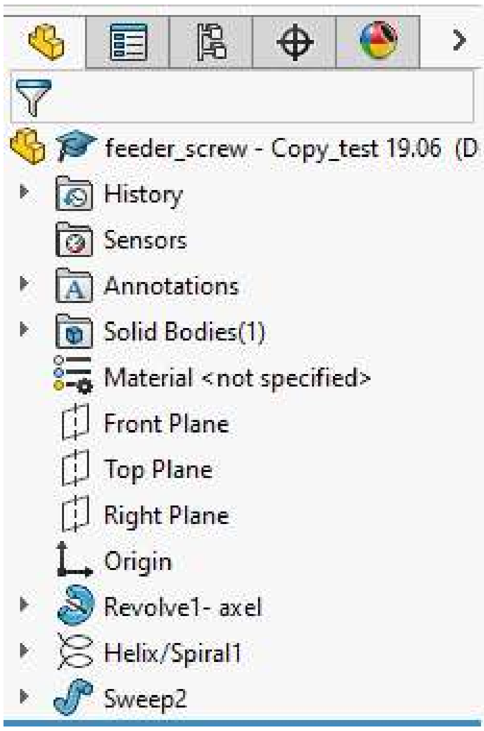
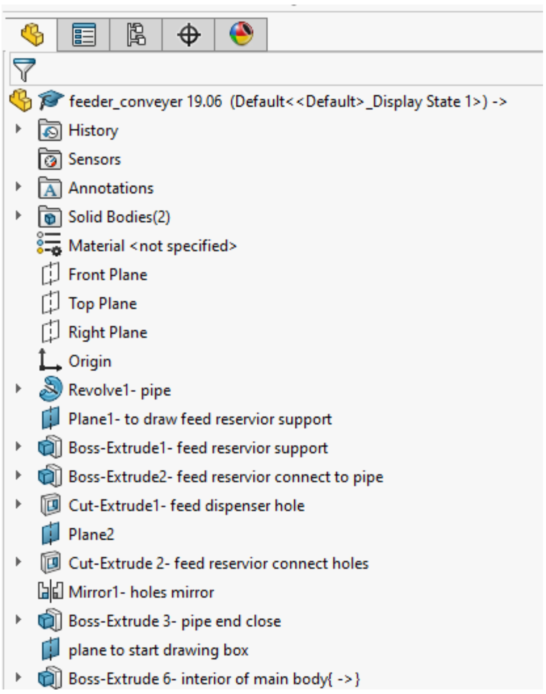
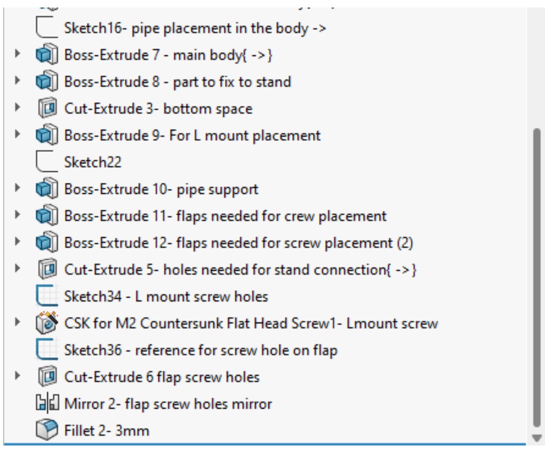


Figure 60: 1

b) (i)Main body that houses the screw conveyor and the DC motor Model tree of body



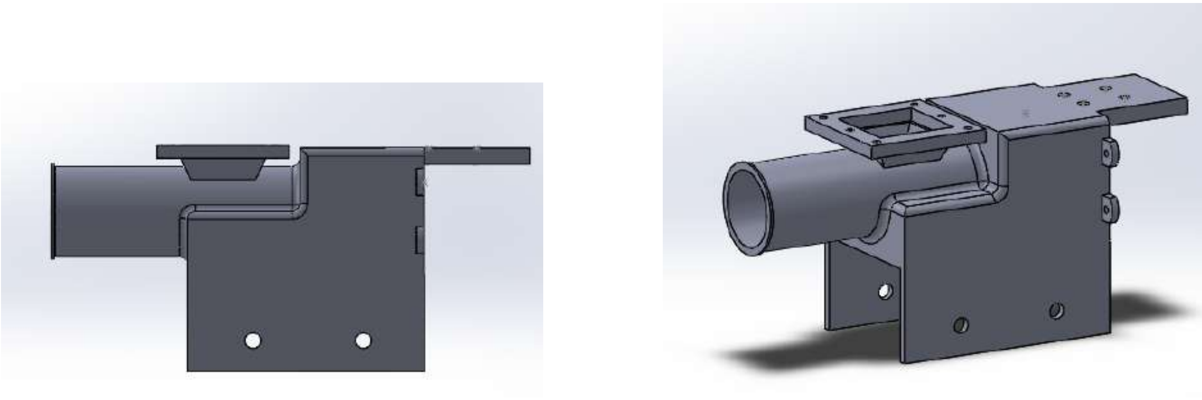
(a) 1



(b) 2

Figure 61: modle tree

The main body serves as the central housing unit for the feeder system, accommodating the screw conveyor, DC motor, and feed reservoir. It includes a designated hole for wiring connections from the interface enclosure, facilitating control of the DC motor. This component integrates all essential parts of the feeder system, ensuring efficient operation and coordination between components for precise feed dispensation in aquaculture settings.



(a) Main body (1) (b) Main body (2)

Figure 62: Main body of the feeder

b) (ii) Lid of the main body
Model tree of lid

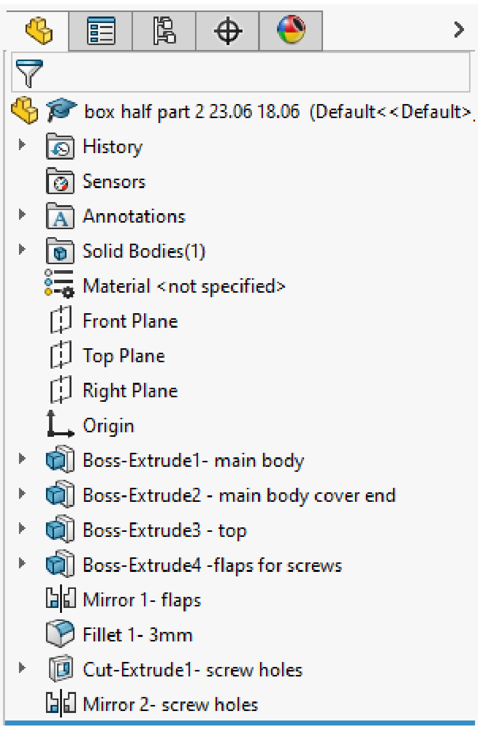
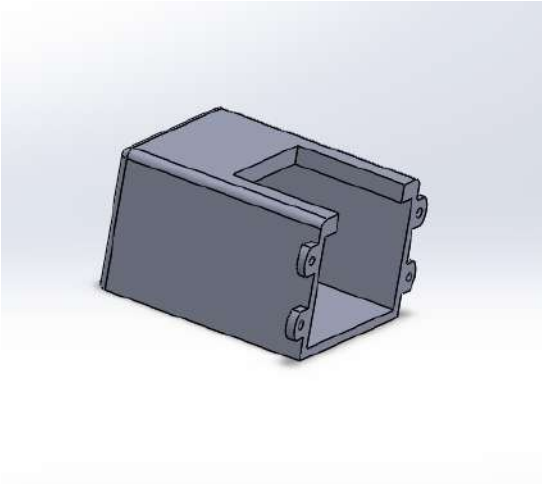
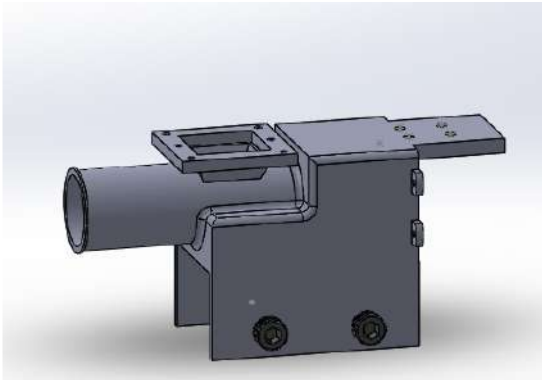


Figure 63: Model tree



(a) Lid of the main body of the feeder (1)



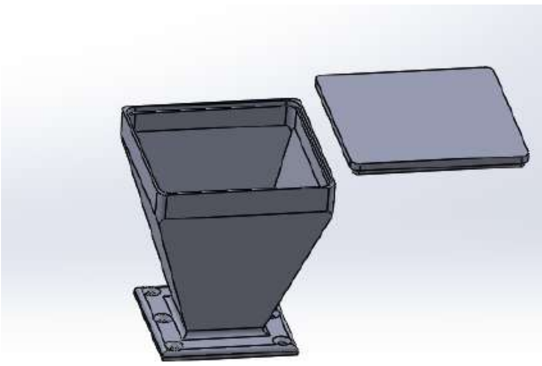
(b) Lid connected to the main body of the feeder

Figure 64: Main body of the feeder

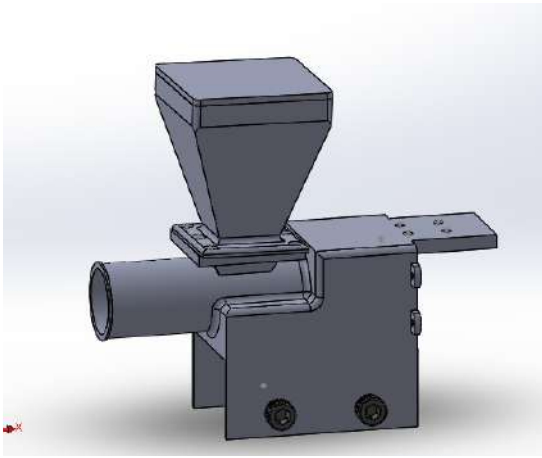
c) Feed reservoir

Model tree of food reservoir and its lid

The feed reservoir serves as a container to store an adequate amount of feed ready for dispensing. It is designed with sufficient capacity to meet the feeding needs of the aquaculture system. The reservoir is securely held by the device, ensuring stability and preventing any weight-related issues. Additionally, it features a lid to protect the feed from exposure to the environment, maintaining its freshness and quality until dispensation.

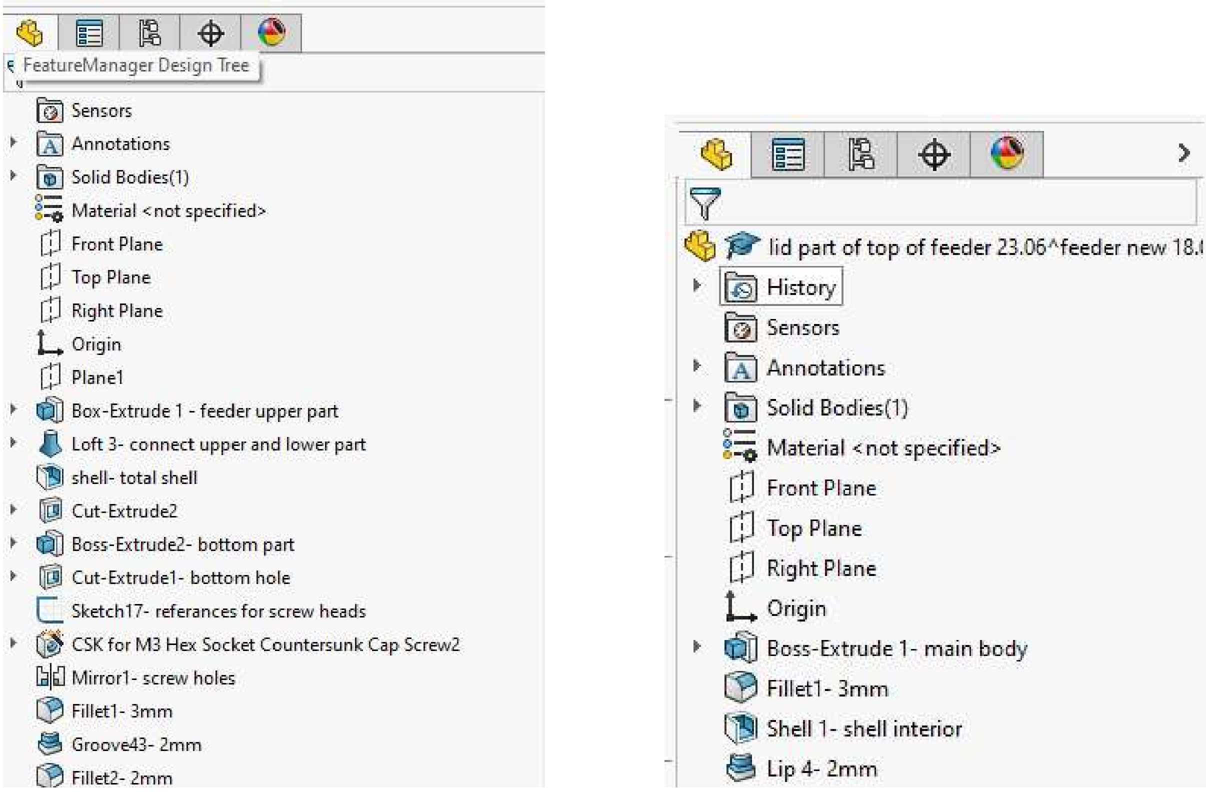


(a) Feed reservoir and lid



(b) Feed reservoir assembled on the feeder

Figure 66: Main body of the feeder



(a) Model tree of feed reservoir

(b) Model tree of lid

Figure 65: Model tree

d) Stand

Model tree of stand

The stand is constructed from iron for robustness and durability, providing sturdy support to securely hold the feeder device in place. Its primary function is to fix the device to the rim of the fish tank using four adjustable screws, ensuring stability and preventing movement during operation. The stand is designed with sufficient thickness to bear the weight of the device effectively. It is connected to the feeder device in a manner that distributes the weight evenly, minimizing the risk of tipping or instability. This ensures the feeder remains securely positioned for reliable operation in aquaculture environments.

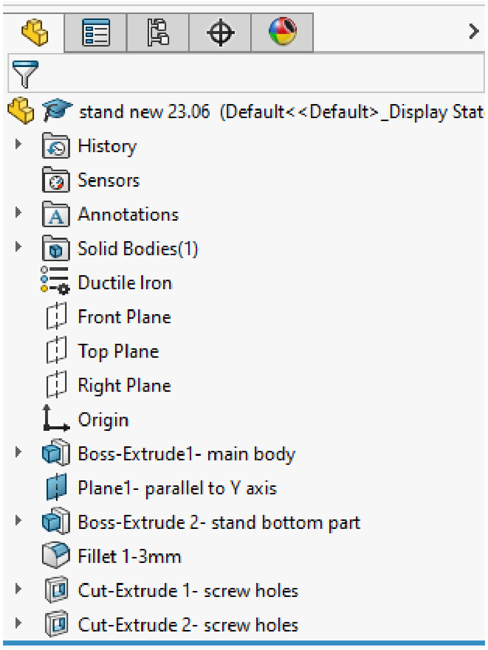
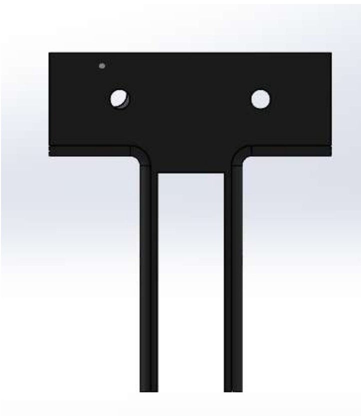
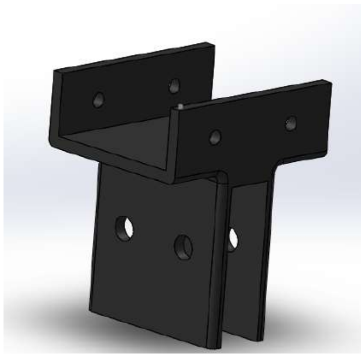


Figure 67: Model tree of feeder stand



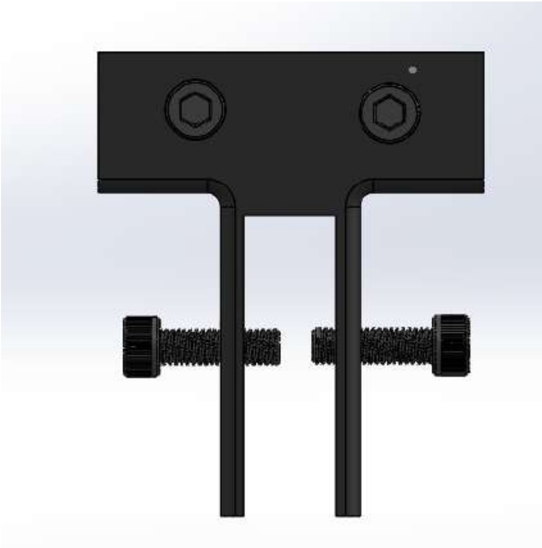
(a) The iorn stand (1)



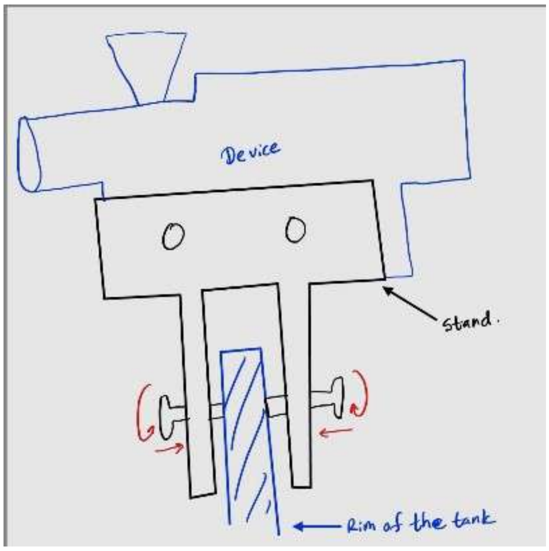
(b) The iorn stand (2)

Figure 68: The stand that holds the device and fix it to the rim of the tank

How the stand holds the device in place:



(a) The iorn stand (1)



(b) sketch of hoe the device is fixed to the rim of the tank

Figure 69: The stand that holds the device and fix it to the rim of the tank

- Final Assembly:

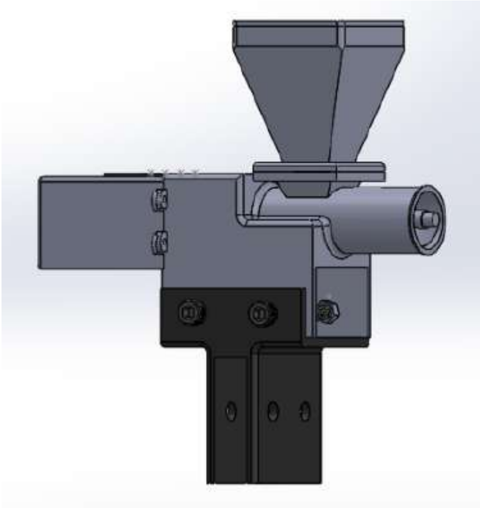
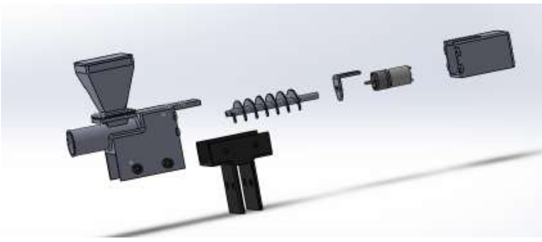
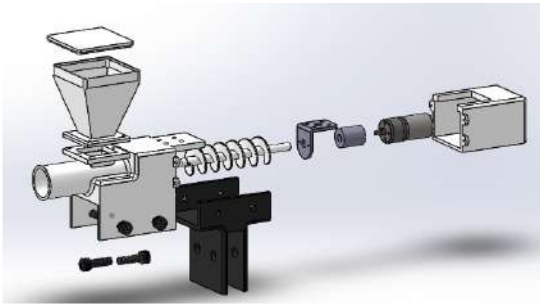


Figure 70: Final feeder Assembly



(a) Assembly of the feeder



(b) Feeder assembly

Figure 71: Feeder assembly

1.10.3 Standardization and Industrial Design

- **Adherence to Standards:** By adhering to industry standards and principles of industrial design, the monitoring / management system and feeder exhibits standardized features and ergonomic considerations, ensuring compatibility with user expectations and industry norms.
- **User-Friendly Design:** The device incorporates a user-friendly design with standardized components and thoughtful placement, enhancing usability and accessibility. This alignment with industrial design standards promotes ease of maintenance, operation, and overall user satisfaction.

Standards used:

- **Electrical Safety Standards:**
  - Compliance with IEC 60950-1 or equivalent for electrical safety of information technology equipment.
  - Ensure components and wiring meet applicable safety standards for electrical appliances.
- **Environmental Protection (Waterproofing):**
  - Adherence to mechanical design standards (e.g., ASME Y14.5) ensures proper fit, tolerance, and structural integrity of components. This enhances durability and operational efficiency under varying environmental conditions typical in aquaculture settings.
- **Component and Connector Standards:**
  - Using connectors compliant with IEC 60603 or equivalent for connectors used in electronic equipment.
  - Ensuring connectors and components meet relevant environmental and electrical standards for reliability and durability.
- **Assembly and Mounting Considerations:**
  - Providing secure mounting options with adjustable screws and hooks suitable for various tank rim thicknesses.
  - Ensuring ease of assembly and maintenance with clear access points for wiring and component servicing.
- **Ergonomic and User Interface Standards:**
  - Following ergonomic design principles (e.g., ISO 9241) ensures user-friendly interfaces and intuitive controls. Our system integrates clear displays and ergonomic button placements to facilitate ease of use and efficient monitoring of aquaculture parameters.
- **Industrial Design Principles:**
  - Integrating industrial design principles for standardized features and aesthetic appeal.
  - Ensuring the system meets expectations for usability, maintenance, and operational efficiency.

1.11 Drawings

1.11.1 User Interface enclosure

a)Main box (dimensions in millimeters)

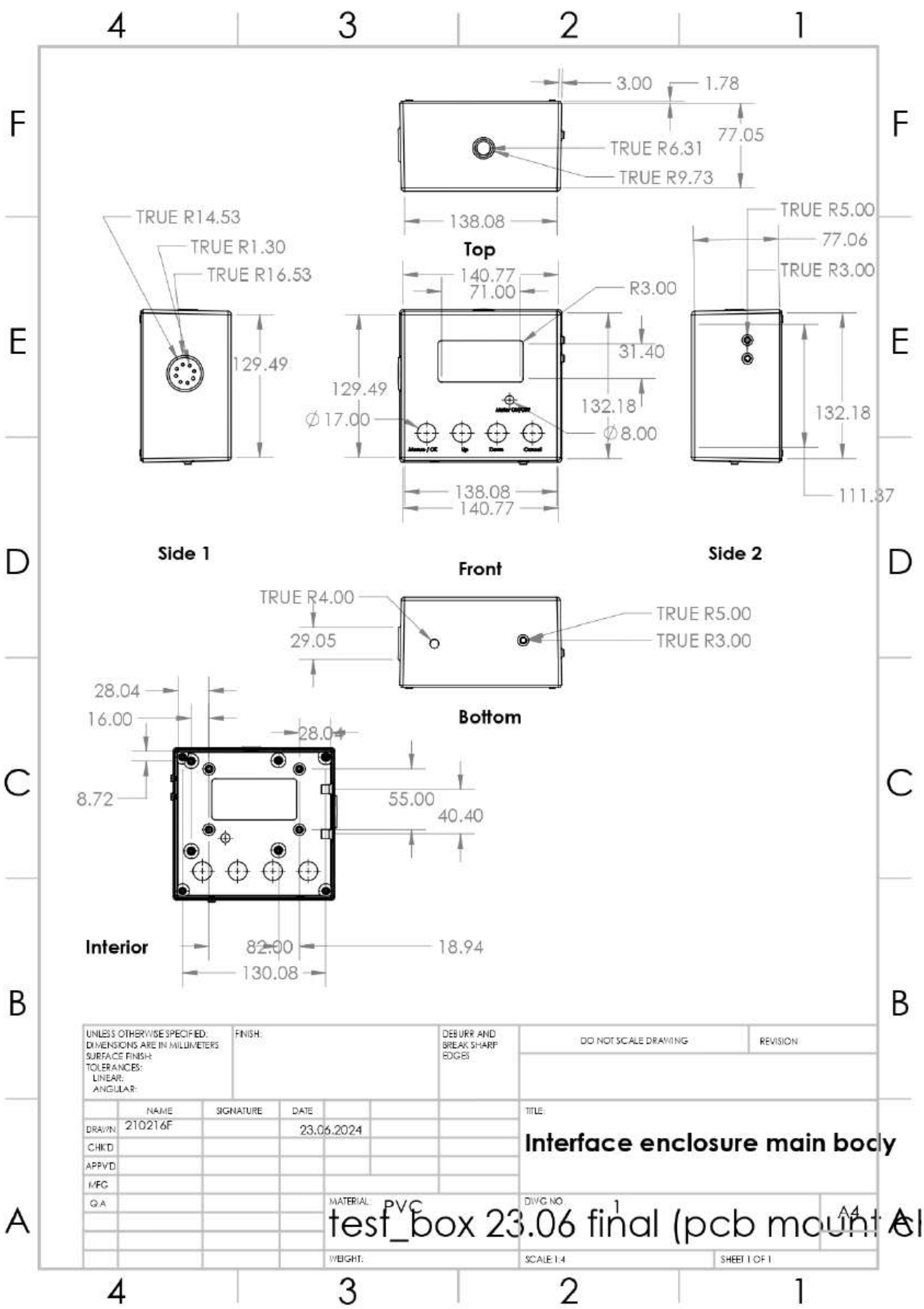


Figure 72: Main box

a)Lid of the user interface enclosure (dimensions in millimeters)

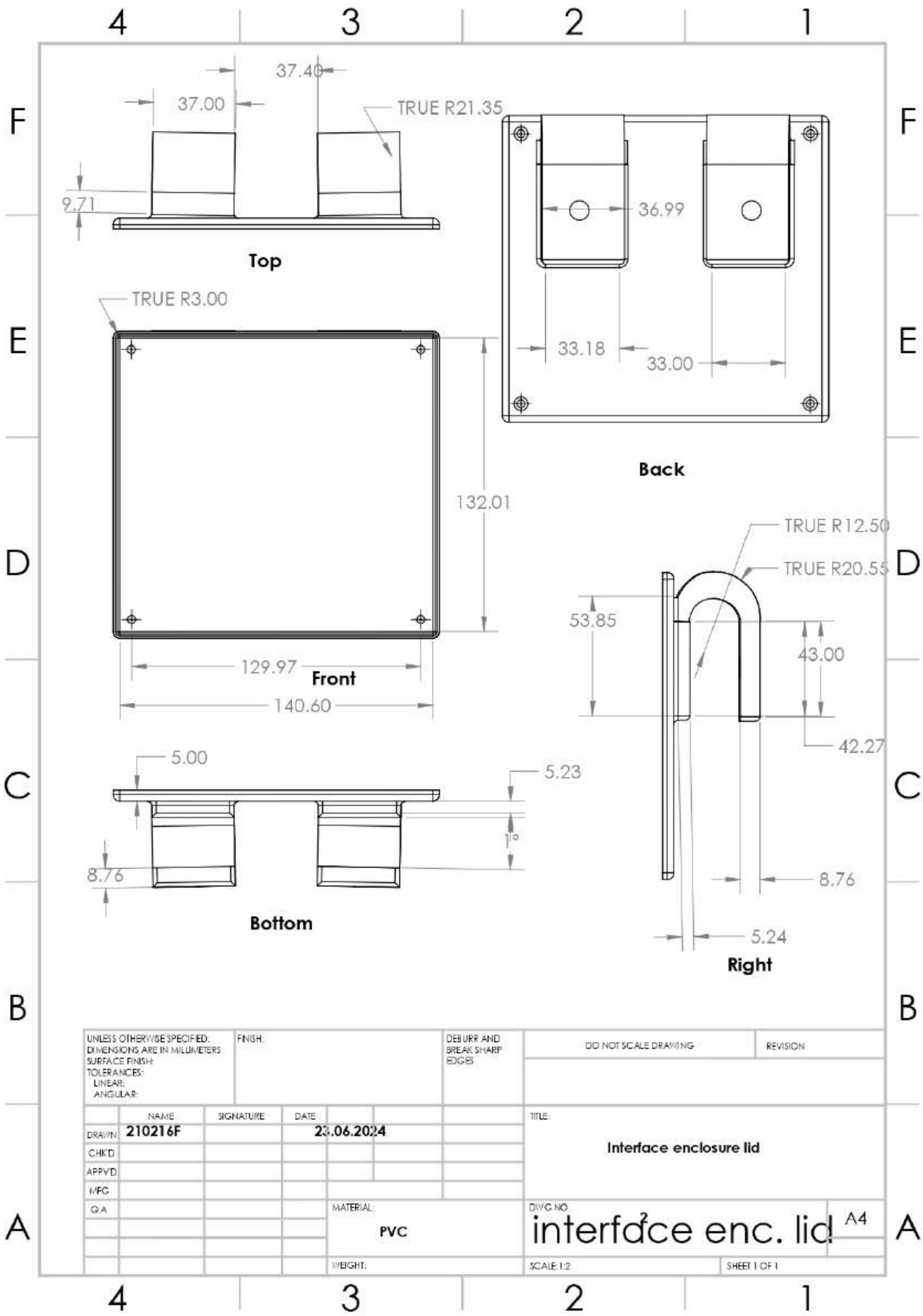


Figure 73: Lid

1.11.2 Feeder enclosure

a)Feeder main body (dimensions in millimeters)

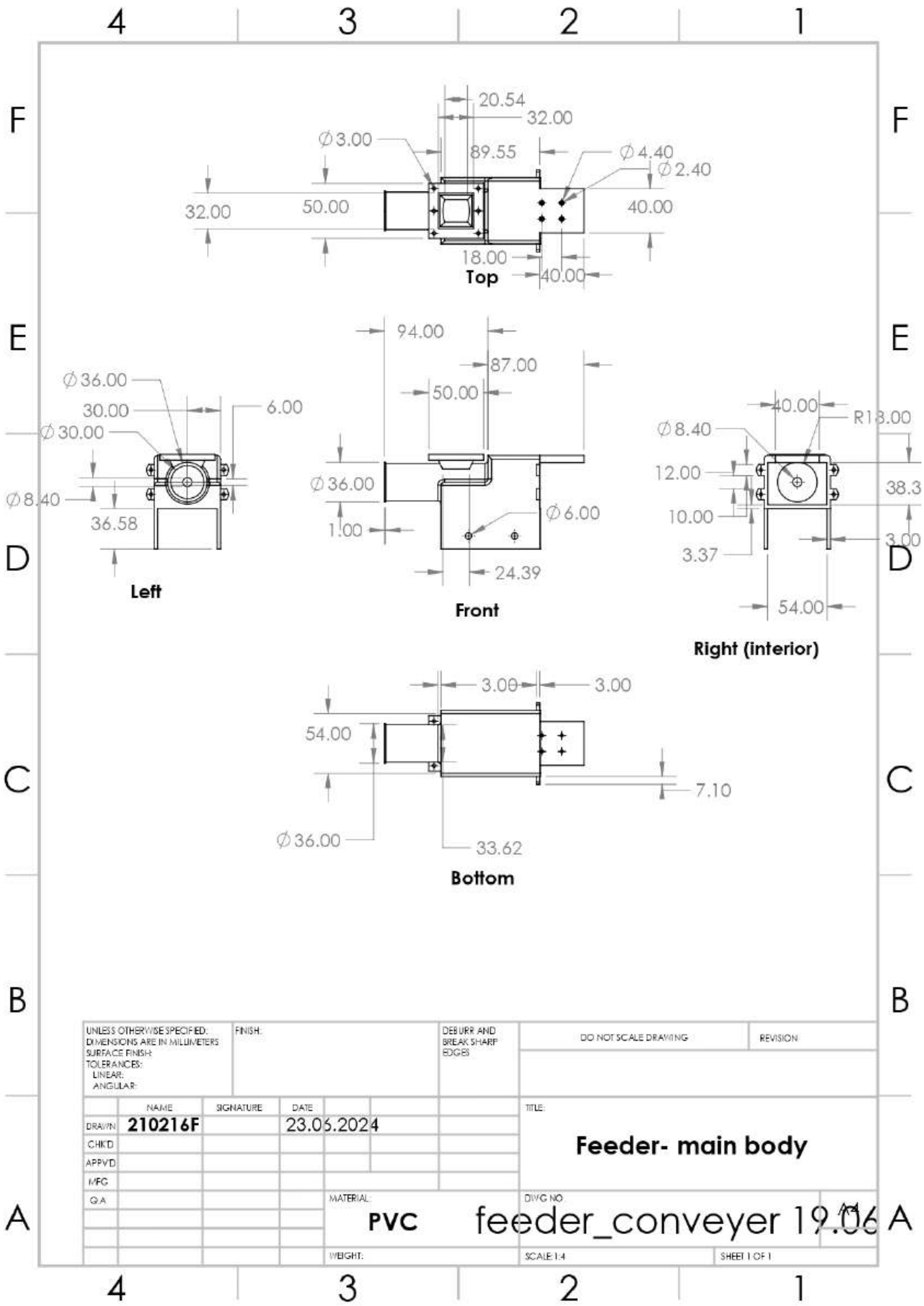


Figure 74: Feeder main body

b) Lid of Feeder main body (dimensions in millimeters)

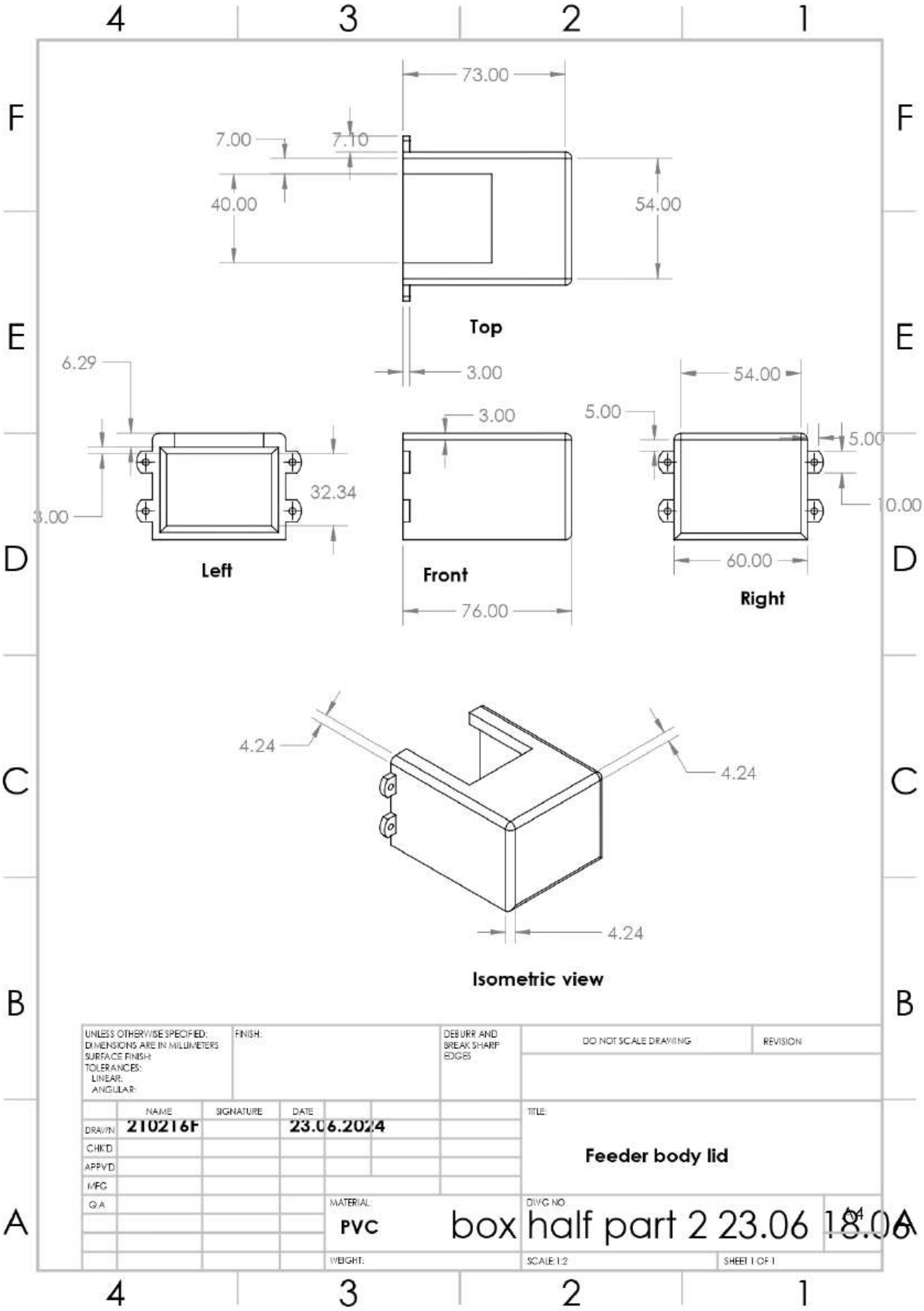


Figure 75: Feeder main body lid

c) Screw conveyor (dimensions in millimeters)

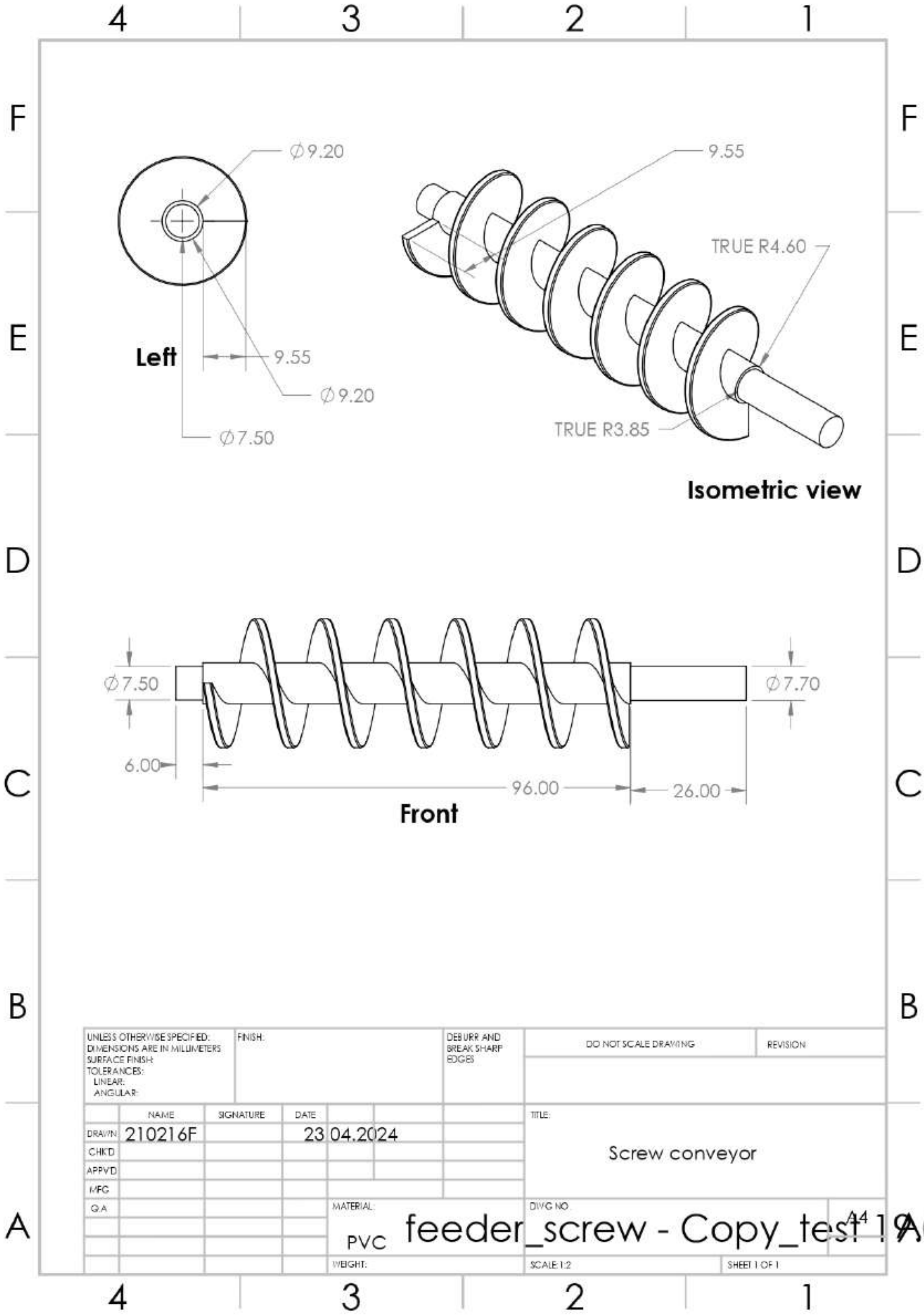


Figure 76: Screw conveyor

d) Feed reservoir (dimensions in millimeters)

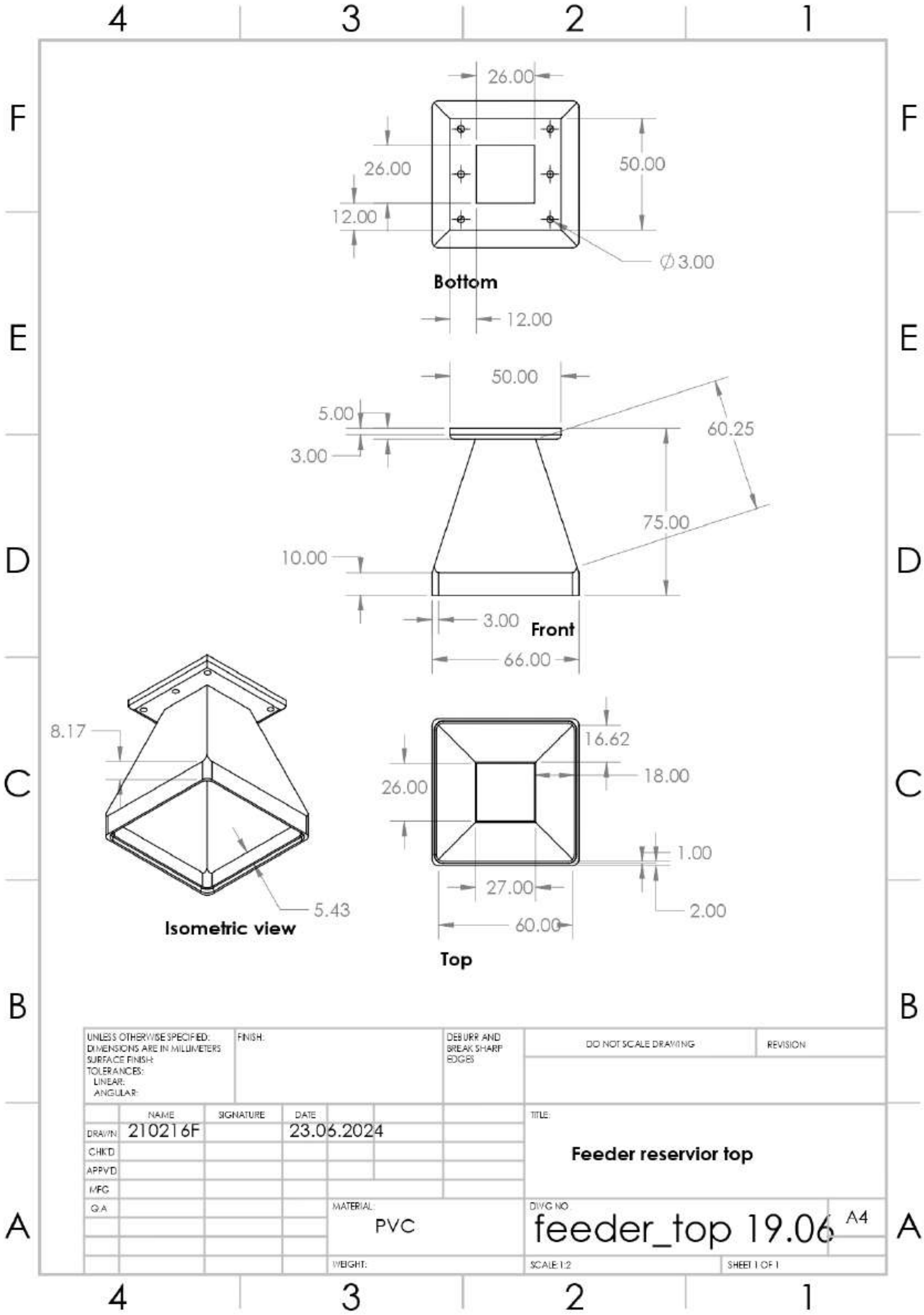


Figure 77: Feed reservoir

e) Feed reservoir lid (dimensions in millimeters)

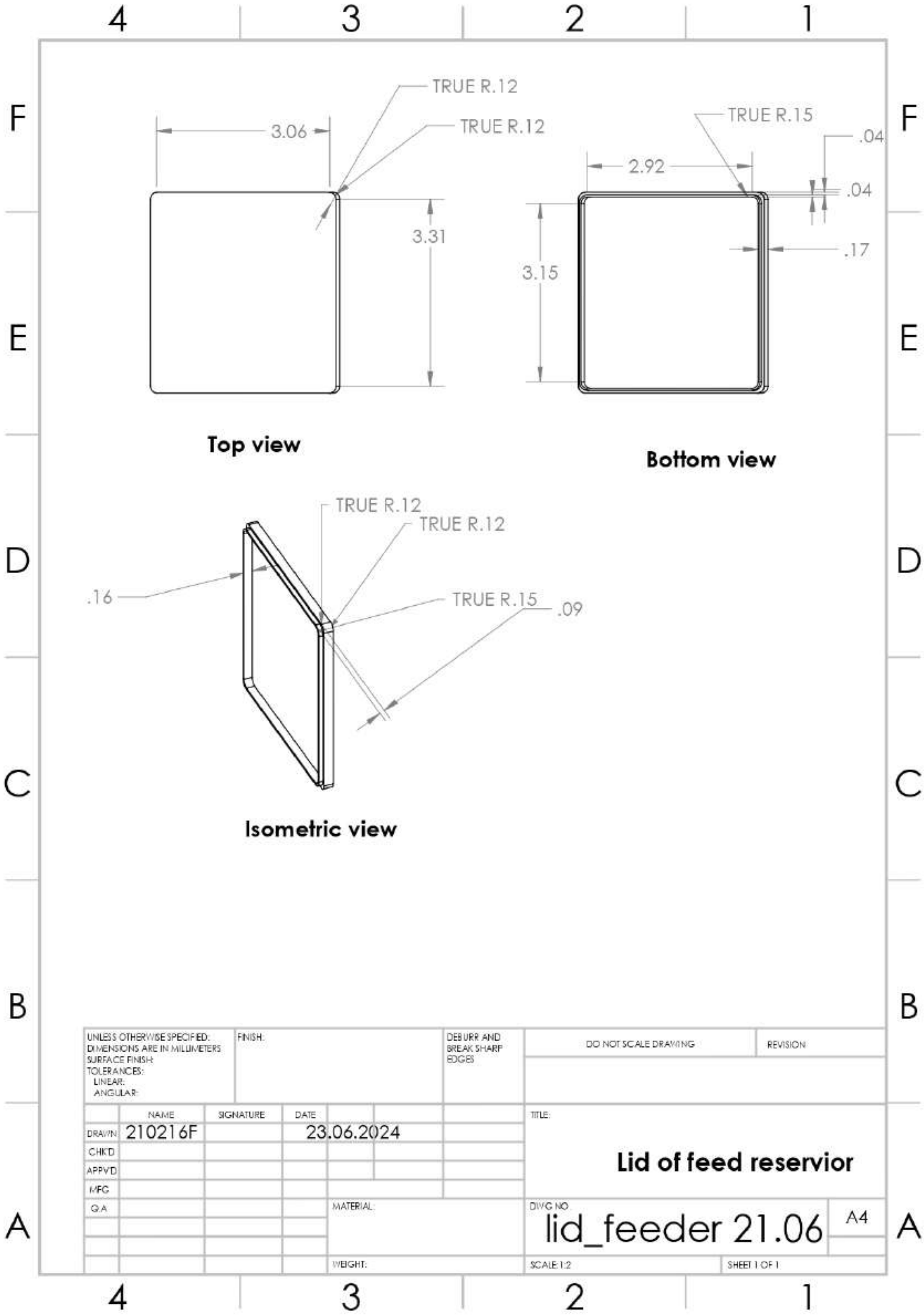


Figure 78: Feed reservoir lid

f) Stand (dimensions in millimeters)

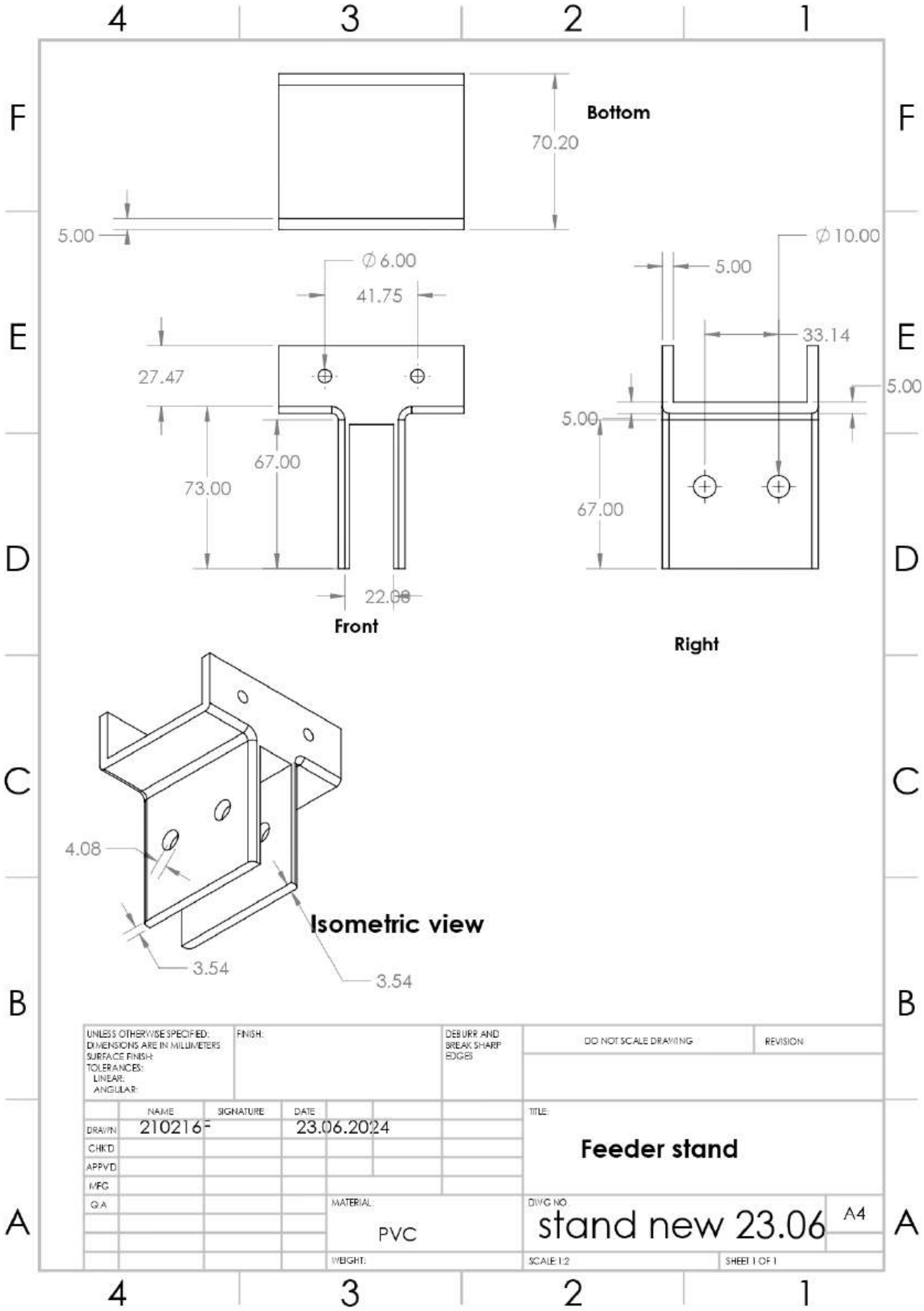


Figure 79: Feeder stand

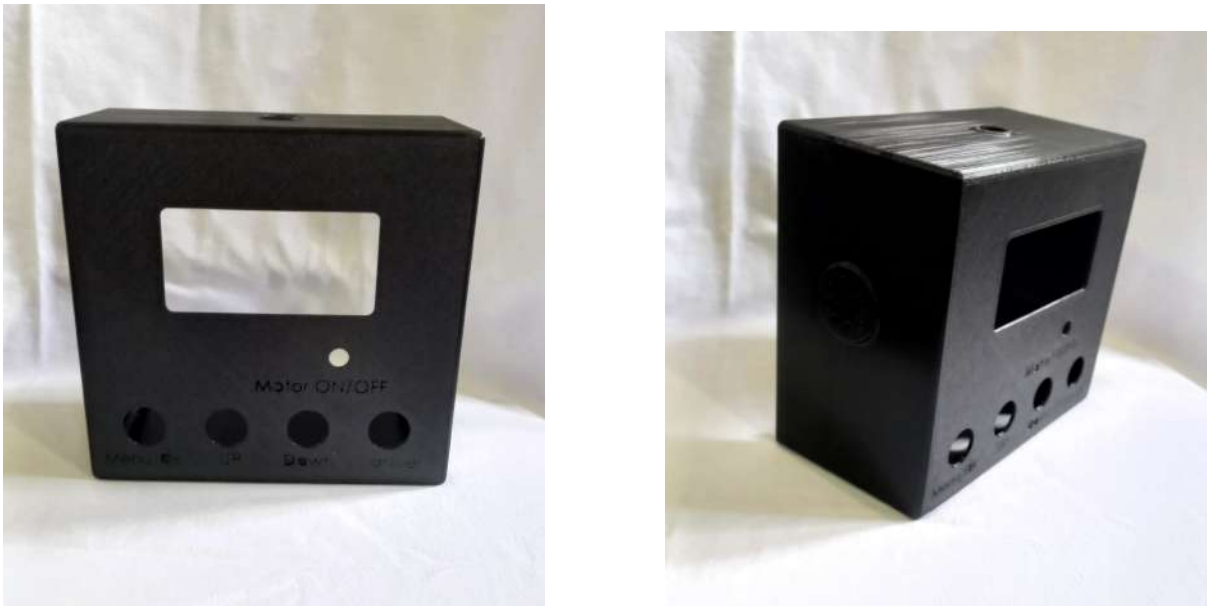
1.12 Photographs of physically built enclosures

1.12.1 User Interface enclosure

(a) Interface Enclosure - image 1

(b) Interface Enclosure - image 2

Figure 80: User Interface Enclosure - Back



(a) Interface Enclosure Lid - image 1

(b) Interface Enclosure Lid - image 2

Figure 81: User Interface Enclosure - Lid

1.12.2 Feeder

Main body of the feeder

(a) Main Body of the Feeder - image 1

(b) Main Body of the Feeder - image 2

Figure 82: Main Body of the Feeder

Back Lid of the Feeder Main Body

(a) Back Lid - image 1

(b) Back Lid - image 2

Figure 83: Back Lid of the Feeder Main Body

Screw Conveyor Spiral

Figure 84: Screw conveyor Spiral

Food Container

(a) Food Container - image 1

(b) Food Container - image 1

Figure 85: Stand

Stand

(a) Feeder Stand - image 1

(b) Feeder Stand - image 2

Figure 86: Feeder Stand

1.13 Firmware Development/Algorithm Implementation

AVR Code:

In our project, we opted for register-level programming and direct port manipulation instead of relying on standard Arduino libraries. This approach enabled us to optimize the code for speed and efficiency.

To achieve precise control over delays, we developed custom delay functions using timer registers. By configuring the timer to generate interrupts at specific intervals, we created accurate microsecond and millisecond delays. This method ensured the precision necessary for our application.

For efficient communication between the microcontroller and other devices, we developed a custom UART library. This approach allowed us to tailor the code to the specific requirements of our application, ensuring reliable and fast data transmission crucial for real-time operations.

Additionally, we required several other libraries, including DS18B20.h, SoftwareSerial.h, LiquidCrystal I2C.h, and WifiEspAT.h. These libraries, originally developed as Arduino libraries by various contributors, were converted into C++ libraries with register-level coding. We modified the functions within these libraries to meet the specific needs of our project, ensuring seamless integration and optimal performance.

1.13.1 Main code

```
#include <avr/io.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>

#include "DS18B20.h"
#include "SoftwareSerial.h"
#include "WifiEspAT.h"
#include "LiquidCrystal_I2C.h"
#include "uart.h"
#include "delay.h"

// Define macros for register-level access for OneWire
#define PIN_TO_BITMASK(pin) (1 << (pin))
#define DIRECT_MODE_INPUT(base, mask) (*(base - 1)) &= ~(mask)

// Define LCD pins
#define LCD_SDA PC4
#define LCD_SCL PC5

// Pushbuttons
#define PB_cancel PD2
#define PB_OK PD3
#define PB_UP PD4
#define PB_DOWN PD5

#define MOTOR_BUTTON PD6

// Buzzer pin
#define BUZZER PB3

// Feeder Motor pin
#define MOTOR_PIN PD7

// TDS pin
#define TdsSensorPin PC2

// PH pin
#define PhSensorPin PC3

// Temp pin
#define ONE_WIRE_BUS PB0

// WiFi pins
#define ESP_TX PD0
#define ESP_RX PD1

```

```
// Ranges
int h_temp = 25;
int l_temp = 2;

float phMinValue = 6.5;
float phMaxValue = 8.0;

float tdsMinValue = 300;
float tdsMaxValue = 1000;

// WiFi and NTP settings
const char* ssid = "MyHomeNetwork";
const char* passphrase = "123home";
const char* ntpServerName = "pool.ntp.org";
#define NTP_PORT 123



// Motor control variables
int motor_hour = 0;            // Hour for motor activation
int motor_minute = 0;          // Minute for motor activation
bool motor_triggered = false;  // Flag to track motor activation
int motor_duration = 10;       // Default motor rotation duration in seconds

// Menu
int current_mode = 0, max_modes = 4;  // Increased max_modes for additional sensor
String modes[] = { "1- Temp range", "2- pH range", "3- TDS range", "4- Set Alarm 2", "5-
     Set Motor Time", "6- Set Motor Duration" };

// For temperature sensor

// For pH sensor
float phCalibrationValue = 21.34;
int phBufferArr[10], phTemp;

// For TDS sensor
#define VREF 5.0
#define SCOUNT 30
int tdsAnalogBuffer[SCOUNT];
int tdsAnalogBufferTemp[SCOUNT];
int tdsAnalogBufferIndex = 0;
int tdsCopyIndex = 0;
float tdsAverageVoltage = 0;
float tdsValue = 0;
float temperature = 16;  // Initial temperature for compensation


//for alarm
bool should_ring_alarm = false;  // Flag to control alarm ringing

// Notes for the buzzer melody
int notes[] = { 262, 294, 330, 349, 392, 440, 494 };

// Number of notes in the melody
int n_notes = 7;



// Function Prototypes
void WiFi_init(const char* ssid, const char* passphrase);
void NTPClient_sendRequest();
bool NTPClient_receiveResponse(uint32_t* time);
void get_current_time(uint32_t epochTime, int* hours, int* minutes, int* seconds);
void OneWire_begin(uint8_t pin);
void connectToWiFi();
void setup();
void loop();
int main();
void update_time_with_check_motor();
int wait_for_button_press();
void go_to_menu();
void set_motor_activation_time();
void set_motor_duration();
```

```
void display_motor_status();
void set_temp_range();
void set_ph_range();
void set_tds_range();
void run_mode(int mode);
void adc_init();
uint16_t adc_read(uint8_t channel);
void check_sensors();
void ring_alarm();

// Initialize UART communication and send AT commands to connect to WiFi
void WiFi_initAndConnect(const char* ssid, const char* passphrase) {
    // Initialize UART communication
    UART_INIT();

    // Send AT commands to connect to WiFi
    UART_SEND("AT+CWJAP=\"");
    UART_SEND(ssid);
    UART_SEND("\",\"");
    UART_SEND(passphrase);
    UART_SEND("\"\r\n");
}

 // Send NTP request packet using UART commands to establish a UDP connection and send
     NTP request packet
void NTPClient_sendRequest() {

    UART_SEND("AT+CIPSTART=\"UDP\",\"");
    UART_SEND(NTP_SERVER);
    UART_SEND("\",");
    UART_SEND(NTP_PORT);
    UART_SEND("\r\n");

    // Construct and send NTP request packet
    uint8_t packetBuffer[48] = {0};
    packetBuffer[0] = 0b11100011;  // NTP request header
    packetBuffer[1] = 0;           // Mode
    packetBuffer[2] = 6;           // Stratum
    packetBuffer[3] = 0xEC;        // Poll Interval
    packetBuffer[12] = 49;         // NTP Magic String
    packetBuffer[13] = 0x4E;
    packetBuffer[14] = 49;
    packetBuffer[15] = 52;

    UART_SEND(packetBuffer, sizeof(packetBuffer)); // Send the packet
}

// Receive and parse the NTP response via UART
bool NTPClient_receiveResponse(uint32_t* time) {
    // Receive NTP response via UART
    uint8_t packetBuffer[48] = {0};
    int index = 0;

    while (index < sizeof(packetBuffer)) {
        uint8_t byte = UART_RECEIVE();
        packetBuffer[index++] = byte;
    }

    // Parse the response
    uint32_t highWord = (packetBuffer[40] << 8) | packetBuffer[41];
    uint32_t lowWord = (packetBuffer[42] << 8) | packetBuffer[43];
    uint32_t secsSince1900 = (highWord << 16) | lowWord;
    *time = secsSince1900 - 2208988800UL;
    return true;
}
// Convert epoch time to hours, minutes, and seconds
void get_current_time(uint32_t epochTime, int* hours, int* minutes, int* seconds) {
    // Convert epoch time to struct tm
    time_t rawTime = (time_t)epochTime;
    struct tm *timeInfo = gmtime(&rawTime); // gmtime converts epoch time to GMT

    if (timeInfo != NULL) {
        *hours = timeInfo->tm_hour;
```

```
        *minutes = timeInfo->tm_min;
        *seconds = timeInfo->tm_sec;
    } else {
        *hours = *minutes = *seconds = -1; // Error value
    }
}

// Set the specified pin as an input for the OneWire communication
void OneWire_begin(uint8_t pin) {
    // Set pin direction to INPUT
    DIRECT_MODE_INPUT(&DDRB, PIN_TO_BITMASK(pin));
}

void setup() {
  //initiate serial communication with wifi
  SoftwareSerial_Init(Rx-pin, Tx_pin,9600);
  //start communication with tempreature pin
  OneWire_begin(TempSensorPin);

  // Initialize UART (Serial) at 9600 baud
  UBRR0H = 0; // Set baud rate to 9600
  UBRR0L = 16; // Set baud rate to 9600
  UCSR0B |= (1 << RXEN0) | (1 << TXEN0); // Enable RX and TX
  UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00); // Set data bits to 8

  // Set input pins
  DDRD &= (~(1<<DDD2)); // Set PD2 (PD2) as input
  DDRD &= (~(1<<DDD3)); // Set PD3 (PD3) as input
  DDRD &= (~(1<<DDD4)); // Set PD3 (PD4) as input
  DDRD &= (~(1<<DDD5)); // Set PD5 (PD5) as input
  DDRD &= (~(1<<DDD6)); // Set PD5 (PD5) as input

  DDRC &= (~(1<<DDC2)); // Set TDS_PIN (PC2) as input
  DDRC &= (~(1<<DDC3)); // Set PH_PIN (PC3) as input

  DDRB &= (~(1<<DDB0)); // Set TempSensorPin (PB0) as input

  DDRD &= (~(1<<DDD0)); // Set ESP_RX (PD0) as input
  DDRD &= (~(1<<DDD1)); // Set ESP_TX (PD1) as input

  // Set output pins
  DDRB |= (1<<DDB3); // Set BUZZER (PB3) as output
  DDRD |= (1<<DDD7); // Set MOTOR_PIN (PD7) as output

  // Connect to WiFi
  connectToWiFi();

  // Initialize time client
  NTPClient_init(ssid,passphrase);

  //Initiate tempreature sensor
  DS18B20_Init(TempSensorPin);

  // Initialize LCD
   // Adjust the address and dimensions according to your LCD module
  lcd_init(0x27, 16, 2);
  lcd_backlight();  // Turn on the backlight
  lcd_clear();
  lcd_setCursor(0, 0);
  lcd_write("welcome!");
  delay_ms(2000);
  lcd_clear();
}

void loop() {
  update_time_with_check_motor();
  check_sensors();
  if (!(PIND & (1 << PB5))) {
    delay_ms(200);
    go_to_menu();
  }
}
```

```
int main(){
    setup();
    while (1)
    {
    loop();
    }
}


void connectToWiFi() {
  // Initialize the WiFi library
  WiFi_init();
  // Connect to WiFi
  wifi_begin(ssid, password);
  while (wifi_status() != WL_CONNECTED) {
    delay_ms(500);
  }
}


// Function to update time and check for motor activation
void update_time_with_check_motor() {

  NTPClient_sendRequest();
  if (NTPClient_receiveResponse()) {
    int hours, minutes, seconds;
    get_current_time(ntpTime, &hours, &minutes, &seconds);

  // Print time on LCD
  lcd_clear();
  lcd_setCursor(0, 0);
  lcd_write(hours);
  lcd_write(":");
  lcd_write(minutes);

  //Check if the manual control is pressed for the motor
  int pressed = wait_for_button_press();
  if (pressed = MOTOR_PIN){
    // If the Motor is working when MOTOR_BUTTON is pressed, turn it off
    if (PIND & (1 << PD7)) {
      // Stop the motor
      PORTD &= ~(1 << PD7);;
      motor_triggered = false;
      lcd_clear();
      lcd_setCursor(6, 0);
      lcd_write("Motor Off");

    // If the Motor is off when MOTOR_BUTTON is pressed, turn it on
    } else if (!(PIND & (1 << PD7))) {
      // Start the motor
      PORTD |= (1 << PD7);
      motor_triggered = true;
      lcd_clear();
      lcd_setCursor(6, 0);
      lcd_write("Motor On");

    }
  }


  // Check if it's time to activate the motor
  if (!motor_triggered && hours == motor_hour && minutes == motor_minute) {
    // Start the motor
    PORTD |= (1 << PD7);
    motor_triggered = true;
    lcd_clear();
    lcd_setCursor(0., 0.);

    lcd_setCursor(6, 0);
    lcd_write("Motor On");

    delay_ms(motor_duration * 1000);  // Run motor for motor_duration seconds
    PORTD &= ~(1 << PD7);;  // Stop the motor
```

```
    motor_triggered = false;
    lcd_clear();
    lcd_setCursor(6, 0);
    lcd_write("Motor Off");
  }

  // Check if motor should be stopped
  if (motor_triggered && (hours != motor_hour || minutes != motor_minute)) {
    // Stop the motor
    PORTD &= ~(1 << PD7);;
    motor_triggered = false;
    lcd_clear();
    lcd_setCursor(0., 0.);
    lcd_setCursor(6, 0);
    lcd_write("Motor Off");
  }
}
// Continuously check for button presses
int wait_for_button_press() {
  while (true) {
    if (!(PIND & (1<<PD2))) {
      return PB_cancel;
    } else if (!(PIND & (1<<PD3))) {
      return PB_OK;
    } else if (!(PIND & (1<<PD4))) {
      return PB_UP;
    } else if (!(PIND & (1<<PD5))) {
      return PB_DOWN;
    }else if (!(PIND & (1 << PD6))) {
      return MOTOR_BUTTON;
    }
  }
}
/**
 * Function to manage the menu interface on an LCD screen.
 * This function allows the user to navigate through different modes and select a
     desired mode.
 * It handles button presses for navigation (UP, DOWN) and actions (OK, CANCEL) while
     updating the display accordingly.
 */
void go_to_menu() {
  while (PIND & (1 << PD2)) {
    lcd_clear();
    lcd_setCursor(0,0);
    lcd_write(modes[current_mode]);

    int pressed = wait_for_button_press();
    delay_ms(1000);

    if (pressed == PB_UP) {
      delay_ms(200);
      current_mode -= 1;
      if (current_mode < 0) {
        current_mode = max_modes - 1;
      }
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      current_mode += 1;
      current_mode = current_mode % max_modes;
    } else if (pressed == PB_OK) {
      delay_ms(200);
      run_mode(current_mode);
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
  }
  lcd_clear();
}

// Function to set motor activation time
void set_motor_activation_time() {
  int entered_hour = 0;
```

```
  int entered_minute = 0;

  // Loop to handle user input for setting motor time
  while (true) {
    lcd_clear();
    lcd_setCursor(0, 0);

    // Display prompt and current entered time
    lcd_setCursor(6, 0);
    lcd_write("Set Motor Time:");
    lcd_setCursor(0, 1);
    lcd_write(String(entered_hour, DEC) + ":" + String(entered_minute, DEC));

    int pressed = wait_for_button_press();

    // Handle button presses to adjust minutes and set the time
    if (pressed == PB_UP) {
      delay_ms(200);
      entered_minute = (entered_minute + 1) % 60;
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_minute = (entered_minute - 1 + 60) % 60;
    } else if (pressed == PB_OK) {
      delay_ms(200);
      motor_hour = entered_hour;
      motor_minute = entered_minute;

      // Display confirmation message
      lcd_clear();
      lcd_setCursor(0, 0);
      lcd_setCursor(6, 0);
      lcd_write("Motor Time Set");
      lcd_setCursor(0, 1);
      lcd_write(String(motor_hour, DEC) + ":" + String(motor_minute, DEC));
      delay_ms(2000);
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
    delay_ms(100);
  }
}


// Function to set motor duration
void set_motor_duration() {
  int entered_duration = motor_duration;

  while (true) {
    lcd_clear();
    lcd_setCursor(0., 0.);
    lcd_setCursor(6, 0);
    lcd_write("Set Motor Duration:");
    lcd_setCursor(0, 1);
    lcd_write(String(entered_duration, DEC) + " sec");


    int pressed = wait_for_button_press();
    if (pressed == PB_UP) {
      delay_ms(200);
      entered_duration += 1;
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_duration = max(entered_duration - 1, 1);  // Ensure duration is at least 1
           second
    } else if (pressed == PB_OK) {
      delay_ms(200);
      motor_duration = entered_duration;
      lcd_clear();
      lcd_setCursor(0., 0.);

      lcd_setCursor(6, 0);
```

```
      lcd_write("Motor Duration Set");
      lcd_setCursor(0, 1);
      lcd_write(String(motor_duration, DEC) + " sec");

      delay_ms(2000);
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
    delay_ms(100);
  }
}

// Function to display motor status
void display_motor_status() {
  lcd_clear();
  lcd_setCursor(0., 0.);
  lcd_write("Motor ");
  lcd_setCursor(6, 0);
  lcd_write(motor_triggered ? "On" : "Off");
  delay_ms(2000);
}
//set desired range for tempreature
void set_temp_range() {
  int entered_h_temp = 27;
  while (true) {
    lcd_clear();
    lcd_setCursor(0,0);
    lcd_write("high temp: " + String(entered_h_temp));

    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay_ms(200);
      entered_h_temp += 1;
      entered_h_temp = entered_h_temp % 31;
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_h_temp -= 1;
      if (entered_h_temp < 25) {
        entered_h_temp = 30;
      }
    } else if (pressed == PB_OK) {
      delay_ms(200);
      h_temp = entered_h_temp;
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
  }

  int entered_l_temp = 27;
  while (true) {
    lcd_clear();
    lcd_setCursor(0,0);
    lcd_write("lowest temp: " + String(entered_l_temp));

    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay_ms(200);
      entered_l_temp += 1;
      entered_l_temp = entered_l_temp % 31;
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_l_temp -= 1;
      if (entered_l_temp < 25) {
        entered_l_temp = 30;
      }
    } else if (pressed == PB_OK) {
      delay_ms(200);
```

```
      l_temp = entered_l_temp;
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
  }
}
//set desired range for pH values
void set_ph_range() {
  float entered_ph_min = 6.5;
  while (true) {
    lcd_clear();
    lcd_setCursor(0,0);
    lcd_write("pH Min: " + String(entered_ph_min));

    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay_ms(200);
      entered_ph_min += 0.1;
      if (entered_ph_min > 14.0) {
        entered_ph_min = 6.5;
      }
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_ph_min -= 0.1;
      if (entered_ph_min < 0.0) {
        entered_ph_min = 0.0;
      }
    } else if (pressed == PB_OK) {
      delay_ms(200);
      phMinValue = entered_ph_min;
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
  }

  float entered_ph_max = 8.0;
  while (true) {
    lcd_clear();
    lcd_setCursor(0,0);
    lcd_write("pH Max: " + String(entered_ph_max));


    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay_ms(200);
      entered_ph_max += 0.1;
      if (entered_ph_max > 14.0) {
        entered_ph_max = 8.0;
      }
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_ph_max -= 0.1;
      if (entered_ph_max < 0.0) {
        entered_ph_max = 0.0;
      }
    } else if (pressed == PB_OK) {
      delay_ms(200);
      phMaxValue = entered_ph_max;
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
  }
}

//set desired range for tds values
```

```
void set_tds_range() {
  int entered_tds_min = 300;
  while (true) {
    lcd_clear();
    lcd_setCursor(0,0);
    lcd_write();
    "TDS Min: " + String(entered_tds_min)
    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay_ms(200);
      entered_tds_min += 10;
      if (entered_tds_min > 2000) {
        entered_tds_min = 300;
      }
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_tds_min -= 10;
      if (entered_tds_min < 0) {
        entered_tds_min = 0;
      }
    } else if (pressed == PB_OK) {
      delay_ms(200);
      tdsMinValue = entered_tds_min;
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
  }

  int entered_tds_max = 1000;
  while (true) {
    lcd_clear();
    lcd_setCursor(0,0);
    lcd_write("TDS Max: " + String(entered_tds_max));

    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay_ms(200);
      entered_tds_max += 10;
      if (entered_tds_max > 2000) {
        entered_tds_max = 1000;
      }
    } else if (pressed == PB_DOWN) {
      delay_ms(200);
      entered_tds_max -= 10;
      if (entered_tds_max < 0) {
        entered_tds_max = 0;
      }
    } else if (pressed == PB_OK) {
      delay_ms(200);
      tdsMaxValue = entered_tds_max;
      break;
    } else if (pressed == PB_cancel) {
      delay_ms(200);
      break;
    }
  }
}

// Execute the selected mode
void run_mode(int mode) {
  if (mode == 0) {
    set_temp_range();  // Set temperature range
  } else if (mode == 1) {
    set_ph_range();  // Set pH range
  } else if (mode == 2) {
    set_tds_range();  // Set TDS range
  } else if (mode == 3) {
    set_motor_activation_time();  // Set motor activation time
  } else if (mode == 4) {
```

```
    set_motor_duration();  // Set motor duration
  } else if (mode == 5) {
    int x = 1;  // Placeholder for future functionalities
  }
}



// Function to initialize ADC
void adc_init() {
    ADMUX = (1 << REFS0); // Reference voltage set to AVcc
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1); // Enable ADC and set prescaler
        to 64
}

// Function to read ADC value
uint16_t adc_read(uint8_t channel) {
    ADMUX = (ADMUX & 0xF0) | (channel & 0x0F); // Select the corresponding channel 0~7
    ADCSRA |= (1 << ADSC); // Start conversion
    while (ADCSRA & (1 << ADSC)); // Wait for conversion to complete
    return ADC;
}

// Function to check sensor readings and trigger alarms if out of range
void check_sensors() {
    // Read temperature from DS18B20 sensor
    float temperatureC = DS18B20_ReadTemperature(TempSensorPin);

    // Check if temperature is above the high threshold
    if (temperatureC > h_temp) {
        lcd_setCursor(0, 40);
        lcd_write("TEMP HIGH");
        ring_alarm();
        delay_ms(200);
    }
    // Check if temperature is below the low threshold
    else if (temperatureC < l_temp) {
        lcd_setCursor(0, 40);
        lcd_write("TEMP LOW");
        ring_alarm();
        delay_ms(200);
    }

    // Read pH value from ADC and convert to pH units
    uint16_t rawPhValue = adc_read(PhSensorPin);
    float phValue = (float)rawPhValue * 5.0 / 1024.0;
    phValue = 3.5 * phValue + phCalibrationValue;

    // Check if pH is out of the defined range
    if (phValue > phMaxValue || phValue < phMinValue) {
        lcd_setCursor(0, 50);
        lcd_write("pH OUT OF RANGE");
        ring_alarm();
        delay_ms(200);
    }

    // Read TDS value from ADC and process the buffer
    uint16_t rawTdsValue = adc_read(TdsSensorPin);
    tdsAnalogBuffer[tdsAnalogBufferIndex] = rawTdsValue;
    tdsAnalogBufferIndex++;
    if (tdsAnalogBufferIndex == SCOUNT) {
        tdsAnalogBufferIndex = 0;
        tdsAverageVoltage = 0;
        for (int i = 0; i < SCOUNT; i++) {
            tdsAverageVoltage += tdsAnalogBuffer[i];
        }
        tdsAverageVoltage /= SCOUNT;
        float tdsVoltage = tdsAverageVoltage * 5.0 / 1024.0;
        float tdsValue = (133.42 * pow(tdsVoltage / VREF, 3) - 255.86 * pow(tdsVoltage /
             VREF, 2) + 857.39 * (tdsVoltage / VREF)) * 0.5;

        // Check if T

```

```

//function to ring the buzzer
void ring_alarm() {
  bool break_happened = false;
  while (should_ring_alarm && (PINB & (1<<PB3))) {
    for (int i = 0; i < n_notes; i++) {
      if (!(PINB & (1<<PB3))) {
        delay_ms(200);
        should_ring_alarm = false;  // Set flag to false
        break_happened = true;
        // Turn off the buzzer
        TCCR0A &= ~(1 << COM0A0);  // Disable output
        break;
      }
      // Generate tone
      int period = 16000000 / notes[i];  // Calculate the period
      OCR0A = period / 2;  // Set the output compare register
      TCCR0A = (1 << WGM01) | (1 << COM0A0);  // Set the timer mode and output mode
      TCCR0B = (1 << CS01) | (1 << CS00);  // Set the prescaler
      DDRB |= (1 << PB3);  // Set PB3 as output

      delay_ms(500);

      // Turn off the buzzer
      TCCR0A &= ~(1 << COM0A0);  // Disable output
      delay_ms(2);
    }
  }

  // Add a delay_ms after the loop to prevent immediate restart
  delay_ms(1000);
}
```

Listing 1: The Main Code

1.13.2 uart library code

For developing this code we obsereved the data sheet of ATmega328PU Microcontroller.
In the library code we manipulated the

1. **UCSR0C Register**: The UCSR0C register (USART Control and Status Register C) is part of the USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) control registers in the ATmega328P microcontroller. This register is used to configure the characteristics of the USART communication, such as the mode of operation, parity bit setting, stop bit setting, and data frame size.
   - **UMSEL01, UMSEL00 (USART Mode Select)**:
     - 00: Asynchronous USART
     - 01: Synchronous USART
   - **UPM01, UPM00 (Parity Mode)**:
     - 00: Disabled (No Parity)
   - **UCSZ02, UCSZ01, UCSZ00 (Character Size)**:
     - 011: 8-bit data
2. **UCSR0B Register**: This register is where the RXEN0 and TXEN0 pins are used to enable and disable the transmitter (TX) and receiver (RX).
3. **UCSR0A Register**: This register includes the U2X0 bit, which is used to select double speed or normal speed mode for USART communication.
   - 0: Normal speed
   - 1: Double speed
4. **UDR0 Register**: This register holds the data that is received and the data that is ready to be transmitted.

5. **UBRR0 Register**: This register is used to set the baud rate for the USART communication.

```
//UART Transmission


//receive one byte data
// It waits until the receive buffer is full by checking the RXC0 bit in the UCSRA
    register. Once data is available, it returns the data from the UART Data Register (
    UDR0).
unsigned char USART_Receive()
{
  while(!(UCSR0A & (1<<RXC0)));  //checks recieve buffer
  return UDR0; // returns the data from the UART Data Register
}


//transmit one byte
//t waits until the transmit buffer is empty by checking the UDRE0 bit in the UCSRA
    register. Then it loads the character ch into the UART Data Register (UDR0) for
    transmission.
void UART_Txchar(char ch)
{
  while(!(UCSR0A & (1<<UDRE0))); //WAIT UNTIL EMPTY TRANSMIT BUFFER
  UDR0 = ch; // after transmit buffer empty load data to the uart data register

}


//transmit string through UART
//It calculates the length of the string and transmits each character in the string one
    by one using the UART_Txchar function.
void trans_string(char str[]){
  int lenght = strlen(str);
  char trans;
  for(int i=0;i<lenght;i++){
    trans = str[i];
    UART_Txchar(trans);
  }
}

//transmit number
// It converts the number to a string, calculates the length of the string, and
    transmits each character in the string one by one using the UART_Txchar function.
void trans_num(uint16_t num){
  String number = String(num);
  int lenght = number.length();
  char trans;
  for(int i=0;i<lenght;i++){
    trans = number[i];
    UART_Txchar(trans);
  }
  // UART_Txchar('\n');
}

void init_uart(){
  //Enable transmitter and recirever bits - Use USART0

  UCSR0B |= (1<<RXEN0) | (1<<TXEN0); // ENABLE TX AND RX

  //SET THE DATA SIZE FOR COMMUNIATION
  UCSR0C &= (~(1<<UMSEL00)) & (~(1<<UMSEL01));   //ENABLE ASYNCHRONOUS USART
      COMMUNICATION BY CLEARING UMSEL00 and UMSEL01 bits in the UCSR0C register
  UCSR0C &= (~(1<<UPM00)) & (~(1<<UPM01));   // DISABLE PARITY BIT BY CLEARING UPM00 AND
       UPM01 BITS IN THE USR0C REGISTER
  UCSR0C &= (~(1<<USBS0)); //CHOOSE ONE STOP BIT BY CLEARING USBS0 BIT

  //SET DATA LENGTH TO BE 8 BITS
  UCSR0B &= (~(1<<UCSZ02));
  UCSR0C |= (1<<UCSZ00) | (1<<UCSZ01);  //SET 8BITS 011 BITS IN UCSZ02,UCSZ00,UCSZ01

  //SET THE SPEED FOR TRANSMISSION
  UCSR0A |= (1<<U2X0); //SELSCT HIGH SPEED MODE SETTING U2X0 bit in the UCSR0A register
      to select high-speed mode.
```

```
  // UCSR0A &= ~(1<<U2X0); // low speed

  //BAUDRATE
  UBRR0 = 207; // SETS UBRR0 REGISTER TO 207 TO GET BAUDRATE OF 9600//9600 BAUDRATE FROM
      16MHz cpu clock rate (U2X0 bit is set (double speed mode))
}
```

Listing 2: uart.cpp file

```
#ifndef UART_H
#define UART_H

unsigned char USART_Receive();
void UART_Txchar(char ch);
void trans_string(char str[]);
void trans_num(uint16_t num);
void init_uart();

#endif
```

Listing 3: uart.h file

1.13.3 delay library code

The delay ms function creates a delay by configuring Timer1 in CTC mode with a prescaler of 64. It sets the compare match register to 249, which corresponds to a delay of 1 millisecond. The function loops for the specified number of milliseconds (ms), generating a precise delay by waiting for the timer to reach the compare match value and then clearing the flag to reset the timer. The following are the registers we manipulated:

1. **TCCR1A:**
   - Configures the behavior of Timer1.
   - In the function, it is cleared to 0 to start with a known state.
2. **TCCR1B:**
   - Sets the mode of operation and clock prescaler for Timer1.
   - Configured to CTC mode with a prescaler of 64.
3. **TCNT1:**
   - Holds the current value of Timer1.
   - Reset to 0 to start counting from zero.
4. **OCR1A:**
   - Sets the compare match value for Timer1.
   - Configured to 249 for a 1ms delay.
5. **TIFR1:**
   - Contains flags for Timer1 events.
   - The OCF1A flag indicates when Timer1 reaches the compare match value and is cleared after each iteration.

```
// delay function

//This loop iterates ms times, creating a delay of ms milliseconds by generating a delay
    of 1 millisecond in each iteration.
void delay_ms(unsigned int ms) {
  for (unsigned int i = 0; i < ms; i++) {
      // These lines reset the Timer1 control registers and counter to ensure Timer1 is
          in a known state before configuring it.
      TCCR1A = 0;  // Clear Timer1 Control Register A
```

```
    TCCR1B = 0;  // Clear Timer1 Control Register B
    TCNT1 = 0;   // Clear Timer1 Counter

    //  sets Timer1 to CTC (Clear Timer on Compare Match) mode by setting the WGM12
        bit in the TCCR1B register. It also sets the prescaler to 64 by setting the
        CS11 and CS10 bits.
    TCCR1B |= (1 << WGM12) | (1 << CS11) | (1 << CS10);

    // Set compare match register for 1ms delay
    OCR1A = 249;

    // waits until the Output Compare Flag A (OCF1A) in the Timer/Counter Interrupt
        Flag Register (TIFR1) is set, indicating that the timer has reached the value
        in OCR1A
    while (!(TIFR1 & (1 << OCF1A)));

    // clears the compare match flag by writing a 1 to the OCF1A bit in the TIFR1
        register. This is necessary to reset the flag so that it can be used again in
        the next iteration of the loop.
    TIFR1 |= (1 << OCF1A);
  }
}
```

Listing 4: delay.cpp file

```
#ifndef DELAY_H
#define DELAY_H

void delay_ms(unsigned int ms);

#endif
```

Listing 5: delay.h file

1.13.4 SoftwareSerial library code

```
/**
* -----------------------------------------------------------------------------------+
* @desc SoftwareSerial.h library
* -----------------------------------------------------------------------------------+
* @source
*  Copyright 2020 Florean Shweiger
* Written by Florean Shweiger

* Modified by Nipuni Herath
* Date: 2024/07/20
* Description: Edited for specific project requirements
*/


// Define the serial pins for communication with the ESP-01 module
// #define ESP_TX 2
// #define ESP_RX 3

//SoftwareSerial espSerial(ESP_TX, ESP_RX);   //add baud rate as well
**********************************************************************
#define SOFTWARE_SERIAL_BUFFER_SIZE 64 // Define buffer size (change if needed)
// Define a buffer for incoming data
static uint8_t rxBuffer[SOFTWARE_SERIAL_BUFFER_SIZE];
static volatile uint8_t rxBufferHead = 0;
static volatile uint8_t rxBufferTail = 0;
static uint8_t txPin;
static uint8_t rxPin;
static uint16_t txDelay;

// Initialize SoftwareSerial with a specific baud rate
//keep
void SoftwareSerial_Init(uint8_t receivePin, uint8_t transmitPin, uint32_t baudRate) {
    rxPin = receivePin;
    txPin = transmitPin;

```

```
    // Set the transmit pin as output
    DDRB |= (1 << txPin);
    PORTB |= (1 << txPin); // Set high to idle state

    // Set the receive pin as input with a pull-up resistor
    DDRB &= ~(1 << rxPin);
    PORTB |= (1 << rxPin); // Enable pull-up resistor

    // Calculate timing parameters for the baud rate
    uint16_t bitDelay = (F_CPU / baudRate) / 4;
    txDelay = bitDelay - (15 / 4);

    // Initialize the receive interrupt
    PCICR |= (1 << PCIE0);            // Enable pin change interrupts for PCINT[7:0]
    PCMSK0 |= (1 << rxPin);           // Enable pin change interrupt for receive pin

}



// End SoftwareSerial communication
void SoftwareSerial_End() {
    // Disable the receive interrupt
    PCMSK0 &= ~(1 << rxPin);
    PCICR &= ~(1 << PCIE0);
}

// Read data from the buffer
int SoftwareSerial_Read() {
    if (rxBufferHead == rxBufferTail) {
        return -1; // Buffer empty
    }

    uint8_t data = rxBuffer[rxBufferHead];
    rxBufferHead = (rxBufferHead + 1) % SOFTWARE_SERIAL_BUFFER_SIZE;
    return data;
}

// Check if data is available in the buffer
int SoftwareSerial_Available() {
    return (SOFTWARE_SERIAL_BUFFER_SIZE + rxBufferTail - rxBufferHead) %
        SOFTWARE_SERIAL_BUFFER_SIZE;
}

// Write data to the transmit buffer
void SoftwareSerial_Write(uint8_t data) {
    // Wait until the transmit line is idle
    while (PINB & (1 << txPin)) {
        _delay_us(txDelay); // Delay for bit timing
    }

    // Start bit
    PORTB &= ~(1 << txPin);
    _delay_us(txDelay);

    // Send each bit
    for (uint8_t i = 0; i < 8; ++i) {
        if (data & (1 << i)) {
            PORTB |= (1 << txPin);
        } else {
            PORTB &= ~(1 << txPin);
        }
        _delay_us(txDelay);
    }

    // Stop bit
    PORTB |= (1 << txPin);
    _delay_us(txDelay);
}
```

Listing 6: SoftwareSerial.cpp file

```
#ifndef SOFTWARE_SERIAL_H
```

```
#define SOFTWARE_SERIAL_H

#include <stdint.h>

// Define buffer size
#define SOFTWARE_SERIAL_BUFFER_SIZE 64

// Initialize SoftwareSerial with a specific baud rate
void SoftwareSerial_Init(uint8_t receivePin, uint8_t transmitPin, uint32_t baudRate);

// End SoftwareSerial communication
void SoftwareSerial_End();

// Read data from the buffer
int SoftwareSerial_Read();

// Check if data is available in the buffer
int SoftwareSerial_Available();

// Write data to the transmit buffer
void SoftwareSerial_Write(uint8_t data);

#endif // SOFTWARE_SERIAL_H
```

Listing 7: SoftwareSerial.h file

1.13.5 LiquiCrystal I2C library code

```

/**
* --------------------------------------------------------------------------------+
* @desc LiquidCrystal_I2C.h library
* --------------------------------------------------------------------------------+
* @source
*  Copyright 2020 Frank De Brabander
* Written by Frank De Brabander

* Modified by Nipuni Herath
* Date: 2024/07/20
* Description: Edited for specific project requirements
*/

#include <avr/io.h>
#include <util/delay.h>

// I2C related constants
#define F_SCL 100000UL // I2C clock speed 100 kHz
#define Prescaler 1
#define TWBR_val ((((F_CPU / F_SCL) / Prescaler) - 16 ) / 2)

// LCD commands
#define LCD_CLEARDISPLAY 0x01
#define LCD_RETURNHOME 0x02
#define LCD_ENTRYMODESET 0x04
#define LCD_DISPLAYCONTROL 0x08
#define LCD_CURSORSHIFT 0x10
#define LCD_FUNCTIONSET 0x20
#define LCD_SETCGRAMADDR 0x40
#define LCD_SETDDRAMADDR 0x80

// flags for display entry mode
#define LCD_ENTRYRIGHT 0x00
#define LCD_ENTRYLEFT 0x02
#define LCD_ENTRYSHIFTINCREMENT 0x01
#define LCD_ENTRYSHIFTDECREMENT 0x00

// flags for display on/off control
#define LCD_DISPLAYON 0x04
#define LCD_DISPLAYOFF 0x00
#define LCD_CURSORON 0x02
#define LCD_CURSOROFF 0x00
#define LCD_BLINKON 0x01
```

```
#define LCD_BLINKOFF 0x00

// flags for display/cursor shift
#define LCD_DISPLAYMOVE 0x08
#define LCD_CURSORMOVE 0x00
#define LCD_MOVERIGHT 0x04
#define LCD_MOVELEFT 0x00

// flags for function set
#define LCD_8BITMODE 0x10
#define LCD_4BITMODE 0x00
#define LCD_2LINE 0x08
#define LCD_1LINE 0x00
#define LCD_5x10DOTS 0x04
#define LCD_5x8DOTS 0x00

#define En 0b00000100  // Enable bit
#define Rw 0b00000010  // Read/Write bit
#define Rs 0b00000001  // Register select bit

#define LCD_BACKLIGHT 0x08
#define LCD_NOBACKLIGHT 0x00

uint8_t _addr;
uint8_t _cols;
uint8_t _rows;
uint8_t _charsize;
uint8_t _backlightval;
uint8_t _displayfunction;
uint8_t _displaycontrol;
uint8_t _displaymode;

void i2c_init() {
    TWSR = 0x00;
    TWBR = (uint8_t)TWBR_val;
}

void i2c_start() {
    TWCR = (1 << TWSTA) | (1 << TWEN) | (1 << TWINT);
    while (!(TWCR & (1 << TWINT)));
}

void i2c_stop() {
    TWCR = (1 << TWSTO) | (1 << TWEN) | (1 << TWINT);
}

void i2c_write(uint8_t data) {
    TWDR = data;
    TWCR = (1 << TWEN) | (1 << TWINT);
    while (!(TWCR & (1 << TWINT)));
}

void expanderWrite(uint8_t data) {
    i2c_start();
    i2c_write(_addr << 1);
    i2c_write(data | _backlightval);
    i2c_stop();
}

void pulseEnable(uint8_t data) {
    expanderWrite(data | En);     // En high
    _delay_us(1);                 // enable pulse must be >450ns
    expanderWrite(data & ~En);    // En low
    _delay_us(50);                // commands need > 37us to settle
}

void write4bits(uint8_t value) {
    expanderWrite(value);
    pulseEnable(value);
}

void send(uint8_t value, uint8_t mode) {
    uint8_t highnib = value & 0xf0;
```

```c
    uint8_t lownib = (value << 4) & 0xf0;
    write4bits(highnib | mode);
    write4bits(lownib | mode);
}

void command(uint8_t value) {
    send(value, 0);
}

void lcd_write(uint8_t value) {
    send(value, Rs);
}

void lcd_init(uint8_t addr, uint8_t cols, uint8_t rows, uint8_t charsize) {
    _addr = addr;
    _cols = cols;
    _rows = rows;
    _charsize = charsize;
    _backlightval = LCD_BACKLIGHT;

    i2c_init();

    _displayfunction = LCD_4BITMODE | LCD_1LINE | LCD_5x8DOTS;

    if (_rows > 1) {
        _displayfunction |= LCD_2LINE;
    }

    if ((_charsize != 0) && (_rows == 1)) {
        _displayfunction |= LCD_5x10DOTS;
    }

    _delay_ms(50);

    expanderWrite(_backlightval);
    _delay_ms(1000);

    write4bits(0x03 << 4);
    _delay_us(4500);
    write4bits(0x03 << 4);
    _delay_us(4500);
    write4bits(0x03 << 4);
    _delay_us(150);
    write4bits(0x02 << 4);

    command(LCD_FUNCTIONSET | _displayfunction);

    _displaycontrol = LCD_DISPLAYON | LCD_CURSOROFF | LCD_BLINKOFF;
    command(LCD_DISPLAYCONTROL | _displaycontrol);

    command(LCD_CLEARDISPLAY);
    _delay_us(2000);

    _displaymode = LCD_ENTRYLEFT | LCD_ENTRYSHIFTDECREMENT;
    command(LCD_ENTRYMODESET | _displaymode);
}

void lcd_clear() {
    command(LCD_CLEARDISPLAY);
    _delay_us(2000);
}


void lcd_setCursor(uint8_t col, uint8_t row) {
    int row_offsets[] = { 0x00, 0x40, 0x14, 0x54 };
    if (row > _rows) {
        row = _rows - 1;
    }
    command(LCD_SETDDRAMADDR | (col + row_offsets[row]));
}


void lcd_display() {
```

```
    _displaycontrol |= LCD_DISPLAYON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}

void lcd_cursor() {
    _displaycontrol |= LCD_CURSORON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}
```

Listing 8: LiquidCrystal_I2C.cpp file

```
#ifndef LIQUIDCRYSTAL_I2C_H
#define LIQUIDCRYSTAL_I2C_H

#include <avr/io.h>
#include <util/delay.h>

// I2C related constants
#define F_SCL 100000UL // I2C clock speed 100 kHz
#define Prescaler 1
#define TWBR_val ((((F_CPU / F_SCL) / Prescaler) - 16 ) / 2)

// LCD commands
#define LCD_CLEARDISPLAY 0x01
#define LCD_RETURNHOME 0x02
#define LCD_ENTRYMODESET 0x04
#define LCD_DISPLAYCONTROL 0x08
#define LCD_CURSORSHIFT 0x10
#define LCD_FUNCTIONSET 0x20
#define LCD_SETCGRAMADDR 0x40
#define LCD_SETDDRAMADDR 0x80

// flags for display entry mode
#define LCD_ENTRYRIGHT 0x00
#define LCD_ENTRYLEFT 0x02
#define LCD_ENTRYSHIFTINCREMENT 0x01
#define LCD_ENTRYSHIFTDECREMENT 0x00

// flags for display on/off control
#define LCD_DISPLAYON 0x04
#define LCD_DISPLAYOFF 0x00
#define LCD_CURSORON 0x02
#define LCD_CURSOROFF 0x00
#define LCD_BLINKON 0x01
#define LCD_BLINKOFF 0x00

// flags for display/cursor shift
#define LCD_DISPLAYMOVE 0x08
#define LCD_CURSORMOVE 0x00
#define LCD_MOVERIGHT 0x04
#define LCD_MOVELEFT 0x00

// flags for function set
#define LCD_8BITMODE 0x10
#define LCD_4BITMODE 0x00
#define LCD_2LINE 0x08
#define LCD_1LINE 0x00
#define LCD_5x10DOTS 0x04
#define LCD_5x8DOTS 0x00

#define En 0b00000100  // Enable bit
#define Rw 0b00000010  // Read/Write bit
#define Rs 0b00000001  // Register select bit

#define LCD_BACKLIGHT 0x08
#define LCD_NOBACKLIGHT 0x00

#ifdef __cplusplus
extern "C" {
#endif

// I2C functions
void i2c_init();
```

```
void i2c_start();
void i2c_stop();
void i2c_write(uint8_t data);

// LCD control functions
void expanderWrite(uint8_t data);
void pulseEnable(uint8_t data);
void write4bits(uint8_t value);
void send(uint8_t value, uint8_t mode);
void command(uint8_t value);
void lcd_write(uint8_t value);

// LCD initialization
void lcd_init(uint8_t addr, uint8_t cols, uint8_t rows, uint8_t charsize);

// High-level LCD commands
void lcd_clear();
void lcd_setCursor(uint8_t col, uint8_t row);
void lcd_display();
void lcd_cursor();
void lcd_backlight();

#ifdef __cplusplus
}
#endif

#endif
```

Listing 9: LiquidCrystal_I2C.h file

1.13.6 Temperature sensor library

```
/**
* ---------------------------------------------------------------------------------+
* @desc DS18B20.h library
* ---------------------------------------------------------------------------------+
* @source
*  Copyright 2020 Charles Joachim
* Written by Juraj Andrassy

* Modified by Nipuni Herath
* Date: 2024/07/20
* Description: Edited for specific project requirements
*/

#include <avr/io.h>
#include <util/delay.h>

// DS18B20 Commands
#define DS18B20_COMMAND_SKIP_ROM 0xCC
#define DS18B20_COMMAND_CONVERT_T 0x44
#define DS18B20_COMMAND_READ_SCRATCHPAD 0xBE
#define DS18B20_COMMAND_WRITE_SCRATCHPAD 0x4E

// Function prototypes
// void DS18B20_Init(uint8_t pin) {
//     DDRB &= ~(1 << pin);  // Set pin as input
//     PORTB |= (1 << pin);  // Enable internal pull-up resistor
// }
// Reads the temperature from the DS18B20 sensor
float DS18B20_ReadTemperature(uint8_t pin) {
    DS18B20_ConvertTemperature(pin);
    _delay_ms(750);  // Wait for conversion to complete

    uint8_t lsb = DS18B20_ReadScratchPad(pin, 0);
    uint8_t msb = DS18B20_ReadScratchPad(pin, 1);

    int16_t raw = (msb << 8) | lsb;
    return raw * 0.0625;  // Convert to Celsius
}

// Resets the DS18B20 sensor and waits for presence pulse
```

```c
static void DS18B20_Reset(uint8_t pin) {
    DDRB |= (1 << pin);  // Set pin as output
    PORTB &= ~(1 << pin);  // Drive pin low
    _delay_us(500);  // Delay 500 s
    DDRB &= ~(1 << pin);  // Set pin as input
    _delay_us(70);  // Wait for presence pulse
    _delay_us(410);  // Wait for end of reset
}

// Writes a byte to the DS18B20 sensor
static void DS18B20_WriteByte(uint8_t pin, uint8_t byte) {
    for (uint8_t i = 0; i < 8; i++) {
        if (byte & (1 << i)) {
            DDRB |= (1 << pin);  // Drive pin low
            _delay_us(1);  // Delay 1 s
            DDRB &= ~(1 << pin);  // Release pin
            _delay_us(60);  // Wait for 60 s
        } else {
            DDRB |= (1 << pin);  // Drive pin low
            _delay_us(60);  // Wait for 60 s
            DDRB &= ~(1 << pin);  // Release pin
            _delay_us(1);  // Delay 1 s
        }
    }
}

// Reads a byte from the DS18B20 sensor
static uint8_t DS18B20_ReadByte(uint8_t pin) {
    uint8_t byte = 0;
    for (uint8_t i = 0; i < 8; i++) {
        DDRB |= (1 << pin);  // Drive pin low
        _delay_us(1);  // Delay 1 s
        DDRB &= ~(1 << pin);  // Release pin
        _delay_us(10);  // Wait for 10 s
        if (PINB & (1 << pin)) {
            byte |= (1 << i);
        }
        _delay_us(50);  // Wait for 50 s
    }
    return byte;
}

// Sends the SKIP ROM command to the DS18B20 sensor
static void DS18B20_SkipRom(uint8_t pin) {
    DS18B20_Reset(pin);
    DS18B20_WriteByte(pin, DS18B20_COMMAND_SKIP_ROM);
}

// Starts a temperature conversion on the DS18B20 sensor
static void DS18B20_ConvertTemperature(uint8_t pin) {
    DS18B20_SkipRom(pin);
    DS18B20_WriteByte(pin, DS18B20_COMMAND_CONVERT_T);
}

// Writes to the DS18B20 scratchpad memory
static void DS18B20_WriteScratchPad(uint8_t pin, uint8_t th, uint8_t tl, uint8_t config) {
    DS18B20_SkipRom(pin);
    DS18B20_WriteByte(pin, DS18B20_COMMAND_WRITE_SCRATCHPAD);
    DS18B20_WriteByte(pin, th);
    DS18B20_WriteByte(pin, tl);
    DS18B20_WriteByte(pin, config);
}

// Reads from the DS18B20 scratchpad memory
static uint8_t DS18B20_ReadScratchPad(uint8_t pin, uint8_t index) {
    uint8_t value;
    DS18B20_SkipRom(pin);
    DS18B20_WriteByte(pin, DS18B20_COMMAND_READ_SCRATCHPAD);
    for (uint8_t i = 0; i <= index; i++) {
        value = DS18B20_ReadByte(pin);
    }
    return value;
```

```
}

///only
```

Listing 10: DS18B20.cpp file

```
#ifndef DS18B20_H
#define DS18B20_H

#include <avr/io.h>
#include <util/delay.h>

// DS18B20 Commands
#define DS18B20_COMMAND_SKIP_ROM 0xCC
#define DS18B20_COMMAND_CONVERT_T 0x44
#define DS18B20_COMMAND_READ_SCRATCHPAD 0xBE
#define DS18B20_COMMAND_WRITE_SCRATCHPAD 0x4E

// Function prototypes
void DS18B20_Init(uint8_t pin);
float DS18B20_ReadTemperature(uint8_t pin);
static void DS18B20_Reset(uint8_t pin);
static void DS18B20_WriteByte(uint8_t pin, uint8_t byte);
static uint8_t DS18B20_ReadByte(uint8_t pin);
static void DS18B20_SkipRom(uint8_t pin);
static void DS18B20_ConvertTemperature(uint8_t pin);
static void DS18B20_WriteScratchPad(uint8_t pin, uint8_t th, uint8_t tl, uint8_t config)
    ;
static uint8_t DS18B20_ReadScratchPad(uint8_t pin, uint8_t index);

#endif
```

Listing 11: DS18B20.h file

1.13.7 WifiEspAT library code

```
/**
* ------------------------------------------------------------------------------------+
* @desc WifiEspAT.h library
* ------------------------------------------------------------------------------------+
* @source
*  Copyright 2019 Juraj Andrassy
* Written by Juraj Andrassy

* Modified by Nipuni Herath
* Date: 2024/07/20
* Description: Edited for specific project requirements
*/
#include <iostream>
#include <cstring>
#include <cstdint>
#include <arpa/inet.h>  // For inet_addr()
#include <unistd.h>     // For sleep()
#include <cstdio>

// Register definitions for the WiFi module
#define WIFI_MODULE_BASE_ADDRESS 0x10000000
#define WIFI_MODULE_RESET_PIN 0x01
#define WIFI_MODULE_INIT_REGISTER 0x04
#define WIFI_MODULE_STATUS_REGISTER 0x08
#define WIFI_MODULE_CONFIG_REGISTER 0x10
#define WIFI_MODULE_SSID_REGISTER 0x70

// Function to send an AT command to the WiFi module
bool sendATCommand(const char* cmd, char* response, size_t responseSize) {
    // Write the command to the WiFi module's command register
    *(volatile uint32_t*)(WIFI_MODULE_BASE_ADDRESS + WIFI_MODULE_INIT_REGISTER) = (
        uint32_t)cmd;

    // Wait for the command to complete
    sleep(1);
```

```

    // Read the response from the WiFi module's response register
    if (response != nullptr) {
        strncpy(response, (char*)(WIFI_MODULE_BASE_ADDRESS + WIFI_MODULE_STATUS_REGISTER
            ), responseSize);
    }

    // Check if the command was successful
    return *(volatile uint32_t*)(WIFI_MODULE_BASE_ADDRESS + WIFI_MODULE_STATUS_REGISTER)
        == 0;
}

// Initialize the WiFi module
bool wifi_init(int serialPort) {
    // Initialize the WiFi module using the specified serial port and reset pin
    *(volatile uint32_t*)(WIFI_MODULE_BASE_ADDRESS + WIFI_MODULE_RESET_PIN) = serialPort
        ;
    return sendATCommand("AT+INIT", nullptr, 0);
}

// Get the WiFi connection status
uint8_t wifi_status() {
    // Return the WiFi connection status
    char response[256];
    if (sendATCommand("AT+STATUS", response, sizeof(response))) {
        // Parse the response to determine the status
        return WL_CONNECTED; // Placeholder
    }
    return WL_DISCONNECTED;
}

// Begin a WiFi connection with the specified SSID and passphrase
int wifi_begin(const char* ssid, const char* passphrase) {
    // Begin a WiFi connection with the specified SSID and passphrase
    char cmd[256];
    sprintf(cmd, "AT+CONNECT=%s,%s", ssid, passphrase);
    if (sendATCommand(cmd, nullptr, 0)) {
        strcpy(ssid, ssid);
        strcpy(passphrase, passphrase);
        return WL_CONNECTED;
    }
    return WL_CONNECT_FAILED;
}

// Disconnect from the WiFi network
int wifi_disconnect(bool persistent) {
    // Disconnect from the WiFi network
    char cmd[32];
    sprintf(cmd, "AT+DISCONNECT=%d", persistent ? 1 : 0);
    if (sendATCommand(cmd, nullptr, 0)) {
        return WL_DISCONNECTED;
    }
    return WL_CONNECT_FAILED;
}

// Configure the WiFi module with a static IP address
bool wifi_config(uint32_t local_ip, uint32_t dns_server, uint32_t gateway, uint32_t
    subnet) {
    // Configure the WiFi module with a static IP address
    char cmd[128];
    sprintf(cmd, "AT+CONFIG_IP=%s,%s,%s,%s",
            inet_ntoa(*(struct in_addr*)&local_ip),
            inet_ntoa(*(struct in_addr*)&dns_server),
            inet_ntoa(*(struct in_addr*)&gateway),
            inet_ntoa(*(struct in_addr*)&subnet));
    return sendATCommand(cmd, nullptr, 0);
}

// Set the DNS servers
bool wifi_setDNS(uint32_t dns_server1, uint32_t dns_server2) {
    // Set the DNS servers
    char cmd[64];
    sprintf(cmd, "AT+DNS=%s,%s",
```

```
            inet_ntoa(*(struct in_addr*)&dns_server1),
            inet_ntoa(*(struct in_addr*)&dns_server2));
    return
```

Listing 12: WifiEspAT.cpp file

```
#ifndef WIFI_MODULE_H
#define WIFI_MODULE_H

#include <cstdint>
#include <arpa/inet.h>

#define WIFI_MODULE_BASE_ADDRESS 0x10000000
#define WIFI_MODULE_RESET_PIN 0x01
#define WIFI_MODULE_INIT_REGISTER 0x04
#define WIFI_MODULE_STATUS_REGISTER 0x08
#define WIFI_MODULE_CONFIG_REGISTER 0x10
#define WIFI_MODULE_SSID_REGISTER 0x70

bool sendATCommand(const char* cmd, char* response, size_t responseSize);
bool wifi_init(int serialPort);
uint8_t wifi_status();
int wifi_begin(const char* ssid, const char* passphrase);
int wifi_disconnect(bool persistent);
bool wifi_config(uint32_t local_ip, uint32_t dns_server, uint32_t gateway, uint32_t
    subnet);
bool wifi_setDNS(uint32_t dns_server1, uint32_t dns_server2);

#endif  // WIFI_MODULE_H
```

Listing 13: WiFiEspAT.h file

1.14 References

- ATMega328P datasheet
- pH Sensor E-201-C datasheet
- Gravity Analog TDS Sensor Datasheet
- Dallas Waterproof Temperature Sensor Datasheet
- ESP 01 WiFi Module Datasheet

2 Appendix A : Weekly Log Entries

2.1 1st February - 11 February 2024

We examined various projects that could be implemented in the industrial sector and decided on developing a fish tank monitoring and management system. This project was chosen because it has the potential to significantly enhance the efficiency and productivity of aquaculture operations by ensuring optimal conditions for fish health and growth.
The device we build for customers will provide real-time monitoring of crucial parameters such as pH levels, temperature, and total dissolved solids (TDS), and will also include an integrated feeder to automate the feeding process. The following are the sensors we chose:

- PH sensor
- Analog TDS water conductivity sensor
- Waterproof digital temperature sensor

(a) PH sensor (b) TDS sensor (c) Digital Waterproof Temperature sensor

Figure 87: Initial sketches

2.2 12 February - 18 February 2024

(created the self evaluation report)
We focused on solving the industrial problem by analyzing potential consumers and customers in the aquaculture sector. We evaluated challenges and opportunities, emphasizing the need for optimal conditions for fish species in Sri Lanka.
We estimated the cost of essential components and ensured they fit within our budget. We selected the ESP32 micro controller **(we changed the Micro contoller later on the project to ATMEGA)** for its reliability and low power consumption and identified necessary sensors for monitoring pH, temperature, and TDS levels. This week's work established a solid foundation for developing a functional and cost-effective system for fish farmers.

Figure 88: ESP32 Microcontroller

2.3 19 February - 25 February 2024

We identified key engineering principles, issues where Mathematics and Science learned during the last three semesters can be applied to our project, hands-on skills we possess for this project, and the applicability of our project for solving an industrial problem in Sri Lanka. Additionally, we listed the cost of the main items needed for our project.

In week 3, we started the **explore phase** of the Cambridge model by creating a stakeholder map, observing users, and identifying similar products in the market. We assessed the stakeholders' needs that must be met through the product.

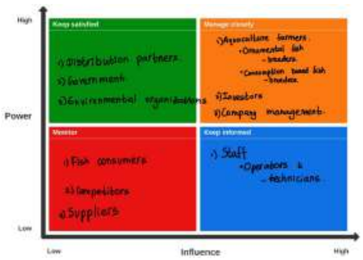


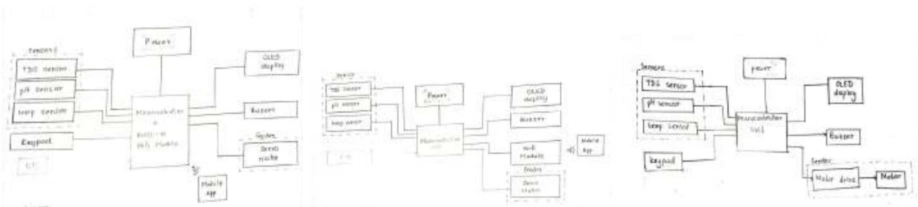
Figure 89: Stake holder map we created

2.4 26 February - 3 March 2024

During week 4, we explored industrial projects to understand how extending Engineering Design Realization (EDR) projects could contribute to the development of the Sri Lankan industry. We discussed the expectations of group formation, focusing on identifying suitable design models, testing methodologies in electronic manufacturing, applying gained knowledge to commercial design projects, designing product enclosures adhering to industry standards, and preparing proper documentation for electronic design. Additionally, we engaged in discussions with other groups working on similar projects to ours to explore potential collaborations and future implementations.

2.5 4 March - 10 March 2024

On this week we started the **create phase.**
We delved into considering possible designs for the prototype, particularly focusing on the product functional bloke diagrams. We compared these designs to existing similar product designs available in the market to gather insights and identify areas for improvement.
We engaged in the creative process by drawing hand-drawn 3D sketches to visualize different design concepts and explore their feasibility and practicality for our project. This step allowed us to brainstorm and refine our ideas before moving forward with the prototype development process.

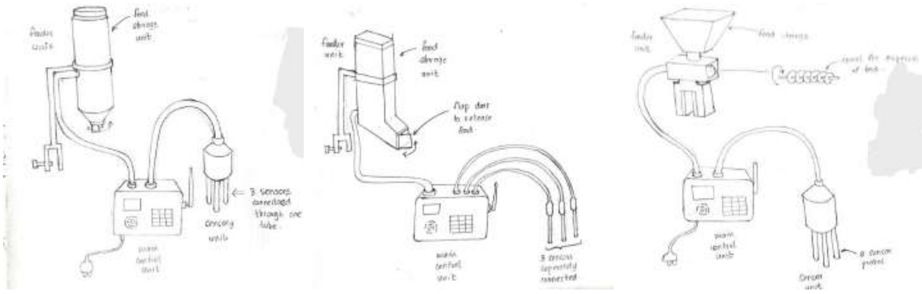


(a) Conceptual design 1 (b) Conceptual design 2 (c) Conceptual design 3

Figure 90: Initial sketches

2.6 11 March - 17 March 2024

We focused on conceptual design, finalizing three different alternatives and drew hand-drawn 3D sketches to visualize the product's physical appearance. Additionally, we designed the electronics in the form of hand-drawn block diagrams to outline the system architecture.



(a) Sketch 1 (b) Sketch 2 (c) Sketch3

Figure 91: Initial sketches

After creating these designs, we evaluated them based on predetermined criteria to assess their feasibility and effectiveness. We cross-checked our designs with other teams to gather feedback and ensure alignment with project goals and requirements. This iterative process allowed us to refine our concepts and move closer to selecting the most suitable design for further development

		Conceptual design 1	**Conceptual design 2**	**Conceptual design 3**
Newly added features		Micro controller with Built-in wifi module Flipping door based feeding mechanism Tube shaped feeding dispenser 3 in one sensory unit	External wifi module for the micro controller Flapping door based feeding mechanism L shaped feeding dispenser 3 sensors separately connected	Micro controller with Built-in wifi module Screw conveyer feeding mechanism 3 in one sensory unit Motor drive and DC motor
Removed features		Sliding door at the bottom of dispenser.	Micro controller with Built-in wifi module 3 in one sensory unit	Servo motor L shaped and tube shaped feeding dispenser
Enclosure design criteria comparison	Functionality	7	5	9
	Durability	5	7	9
	Manufacturing Feasibility	7	8	8
	Assembly and serviceability	8	6	7
	Simplicity	5	6	8
	Scalability	7	8	9
	Safety	9	9	9
	Ergonomics	7	8	8
	User experience	5	6	9

Figure 92: Evaluation of the 3 conceptual designs

2.7 18 March - 24 March 2024

We created a rough circuit for our project and then selected suitable electronic components (buzzer type, display type, feeder, etc.) for our implementation considering the electronic and mechanical properties. The following figure 46 shows some of our initially selected components.

(a) OLED screen (b) Push button (c) WifiModule (d) Piezo buzzer

Figure 93: Components that we initially selected

We decided to use Gear Motor 12V 50RPM as our motor.

Figure 94: DC motor

After selecting the components we started the 3D modeling of the feeder part using Solidworks.

(we did NOT start the design of the enclosure which houses the PCB due to not finalizing our PCB)

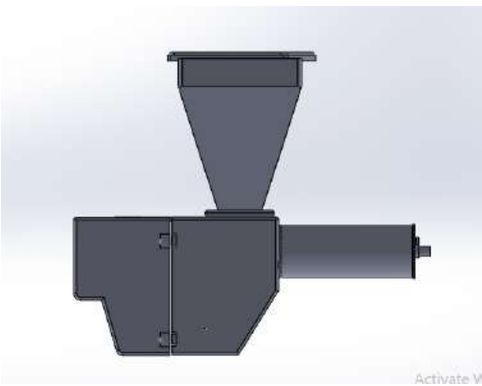


Figure 95: Initial feeder design

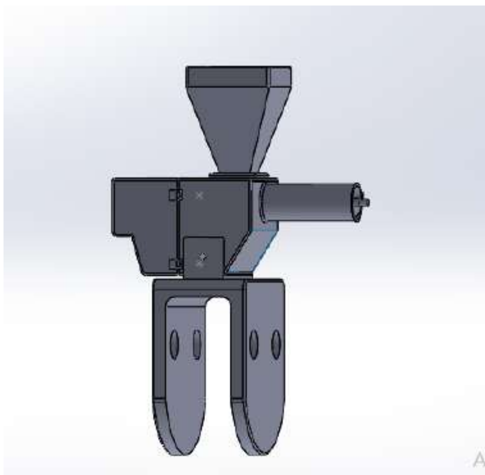


Figure 96: Initial feeder design with stand

In parallel with the 3D modeling, we started designing the circuit for the already discussed and finalized conceptual design.

2.8 25 March - 31 March 2024

During this week, our team focused on reviewing the PCB components required for our aquaculture tank monitoring system. We evaluated the availability and suitability of each component, ensuring they met the specifications outlined in our design.
We decided to use the Micro controller the ATMEGA328P-PU dual-in-line processor chip instead of the Esp32 chip due to its industrial recognition.
Based on our component review, we identified several areas where the initial PCB design could be optimized. We made necessary adjustments to accommodate available components and to enhance the overall functionality of the PCB.

We placed an order with LCSC for the main components. This included a thorough review of the components datasheets and specifications to ensure they met our design criteria. Then we designed the first version of the PCB as a trial run. This allowed us to test our design assumptions and to identify any potential issues in the early stages as we paid close attention to PCB design factors such as trace width, component placement, and overall layout.

After designing the first PCB, we conducted a review, focusing on key PCB design factors to ensure that the PCB size was optimal for the overall design, balancing functionality with compactness. The following Figure shows the image of the firstly designed PCB and some of our initial schematics.

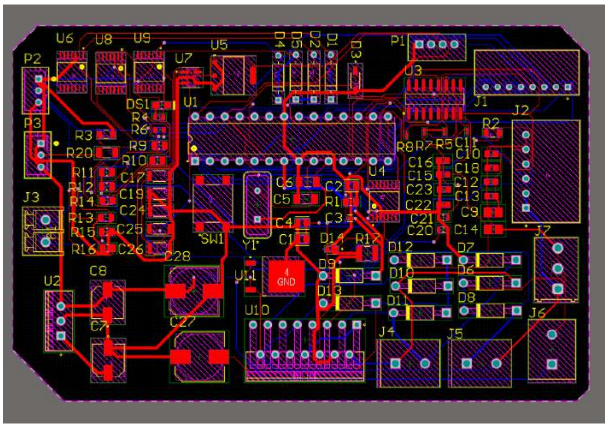


Figure 97: The first PCB

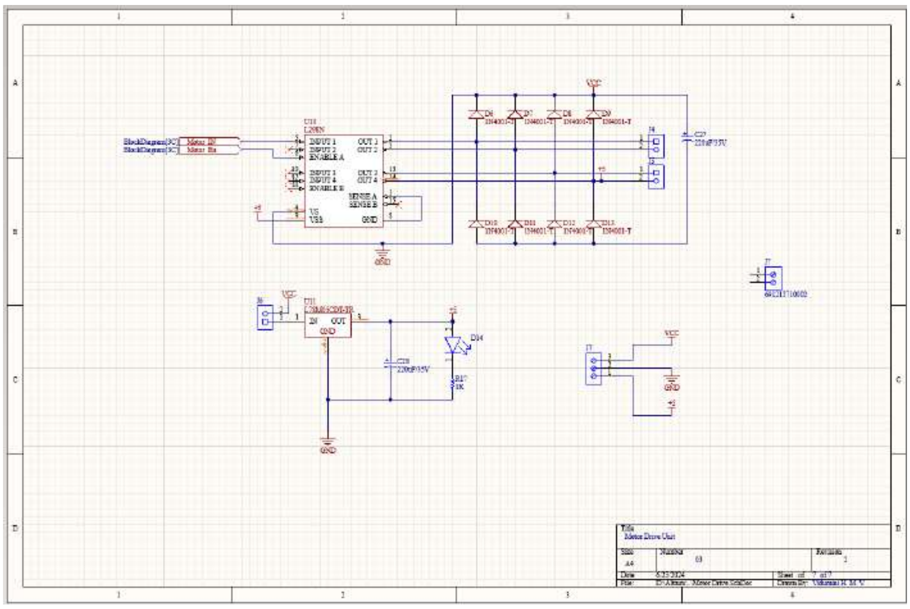


Figure 99: Initial Motor Drive

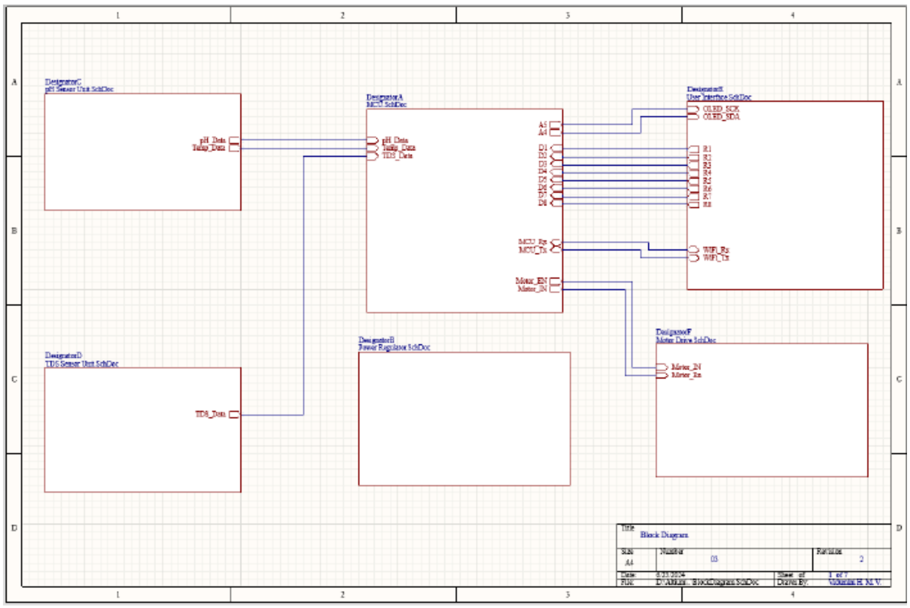
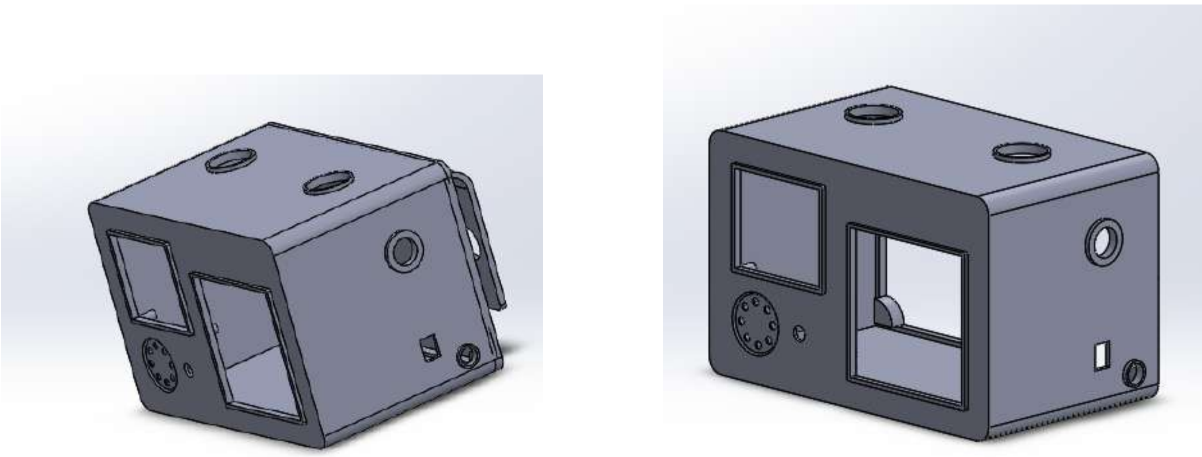


Figure 98: Initial Block Diagram

After completing the PCB design, our next step was to create an initial enclosure specifically tailored to house the PCB, serving as our user interface enclosure.



(a) Initial enclosure which housed the PCB (1)

(b) Initial enclosure which housed the PCB (2)

Figure 100: First User Interface enclosure design

2.9 1 April - 7 April 2024

Building on the insights gained from our trial PCB, we continued to refine and revise our PCB design. This iterative process involved multiple rounds of review and adjustments.

During our reviews, we realized that the initial motor driver circuit was more complex than necessary for our implementation. After careful consideration, we decided to simplify the motor driver circuit, removing unnecessary functions and streamlining the design. This change required us to redesign and review the circuit to ensure it still met our performance requirements.

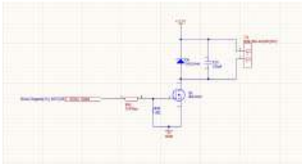


Figure 101: New Motor Driver Circuit

We also realized that the OLED display would be too small for our product therefore we chose the LCD 16X4 LCD display with blue back light with the I2 C module.

Figure 102: 16x4 blue Backlight LCD Display

Additionally, we decided to use the ATMEGA328P-AU SMD microcontroller chip instead of the ATMEGA328P-PU dual-in-line processor chip. This change was made to improve the compactness and efficiency of the PCB design.

Figure 103: ATMEGA328P-AU SMD

Parallel to our hardware design efforts, we developed code sections for the system. We implemented these sections on a simulator for initial testing, focusing on individual parts of the circuit. This allowed us to debug and refine the code in a controlled environment before integrating it with the entire system. Once individual parts of the circuit were tested successfully, we proceeded to test the entire circuit as a whole on software simulation. This comprehensive testing phase was critical to ensure that all components and code sections worked seamlessly together.

Below are some functions we simulated

```
void go_to_menu() {
  while (digitalRead(PB_cancel) == HIGH) {
    lcd.clear();
    print_line(modes[current_mode], 0, 0, 2);
    int pressed = wait_for_button_press();
    delay(1000);

    if (pressed == PB_UP) {
      delay(200);
      current_mode -= 1;
      if (current_mode < 0) {
        current_mode = max_modes - 1;
      }
    } else if (pressed == PB_DOWN) {
      delay(200);
      current_mode += 1;
      current_mode = current_mode % max_modes;
    } else if (pressed == PB_OK) {
      delay(200);
      run_mode(current_mode); // Execute selected mode
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }
  lcd.clear();
}
```

Listing 14: Function to navigate the menu

```
// Function to handle different modes
void run_mode(int current_mode) {
  lcd.clear();
  print_line("running mode " + String(current_mode), 0, 0, 1);
  delay(2000);

  if (current_mode == 0) {
    set_temp_range(); // Set temperature range
  } else if (current_mode == 1) {
    set_ph_range(); // Set pH range
  } else if (current_mode == 2) {
    set_tds_range(); // Set TDS range
  } else if (current_mode == 3) {
    set_motor_activation_time(); // Set motor activation time
  } else if (current_mode == 4) {
    set_motor_duration(); // Set motor duration
  }
}
```

Listing 15: Function to run modes in the menu

```
int wait_for_button_press() {
  while (true) {
    if (digitalRead(PB_UP) == LOW) {
      return PB_UP;
    } else if (digitalRead(PB_DOWN) == LOW) {
      return PB_DOWN;
    } else if (digitalRead(PB_OK) == LOW) {
      return PB_OK;
    } else if (digitalRead(PB_cancel) == LOW) {
      return PB_cancel;
    }
  }
}
```

Listing 16: Button Input Handling Function

```
// Function to wait for button press
int wait_for_button_press() {
  while (true) {
    if (digitalRead(PB_UP) == LOW) {
      return PB_UP;
    } else if (digitalRead(PB_DOWN) == LOW) {
      return PB_DOWN;
    } else if (digitalRead(PB_OK) == LOW) {
      return PB_OK;
    } else if (digitalRead(PB_cancel) == LOW) {
      return PB_cancel;
    }
  }
}
```

Listing 17: Button Input Handling Function

```

// Function to ring the alarm
bool should_ring_alarm = true;

void ring_alarm() {
    bool break_happened = false;
    while (should_ring_alarm && digitalRead(PB_cancel) == HIGH) {
        for (int i = 0; i < n_notes; i++) {
            if (digitalRead(PB_cancel) == LOW) {
                delay(200);
                should_ring_alarm = false; // Set flag to false
                break_happened = true;
                noTone(BUZZER); // Turn off the buzzer
                break;
            }
            tone(BUZZER, notes[i]);
            delay(500);
            noTone(BUZZER);
            delay(2);
        }
    }

    // Add a delay after the loop to prevent immediate restart
    delay(1000);
}
```

Listing 18: Function to ring alaram

2.10 15 April - 21 April 2024

During this week, we focused on further developing and refining the code. We implemented the code on a breadboard, allowing us to test and debug the working prototype in a flexible and adjustable setup.

Following the recent changes to the motor drive circuit, we had to re-evaluate and iterate our design process to ensure compatibility and functionality. This involved revisiting the circuit design, updating the code accordingly, and conducting testing to verify the revised motor drive circuit worked as intended within the overall system.

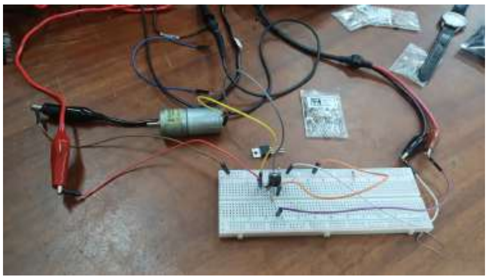


Figure 104: DC motor controlling circuit

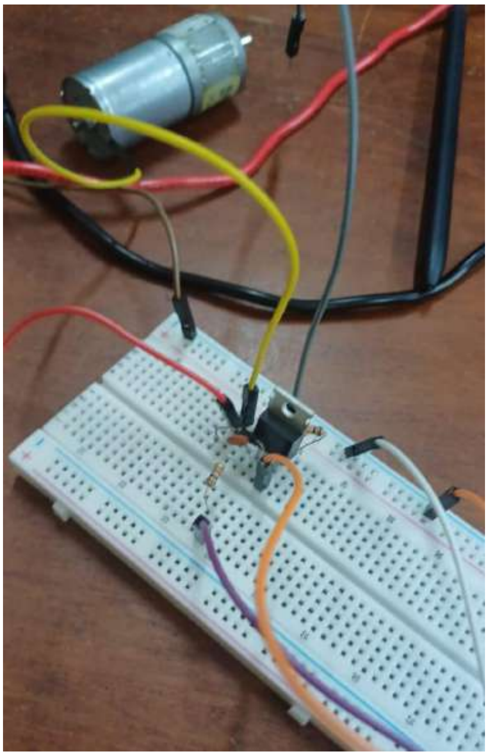


Figure 105: Circuit

After reviews and adjustments, we redesigned and finalized the PCB design. This final version incorporated all the necessary changes and optimizations from our iterative design process. We conducted a review to ensure the design met all specifications and standards. Once satisfied, we sent the finalized PCB design for manufacturing and placed orders for any additional components required for the production phase.

The following Figure shows the finalized PCB design.

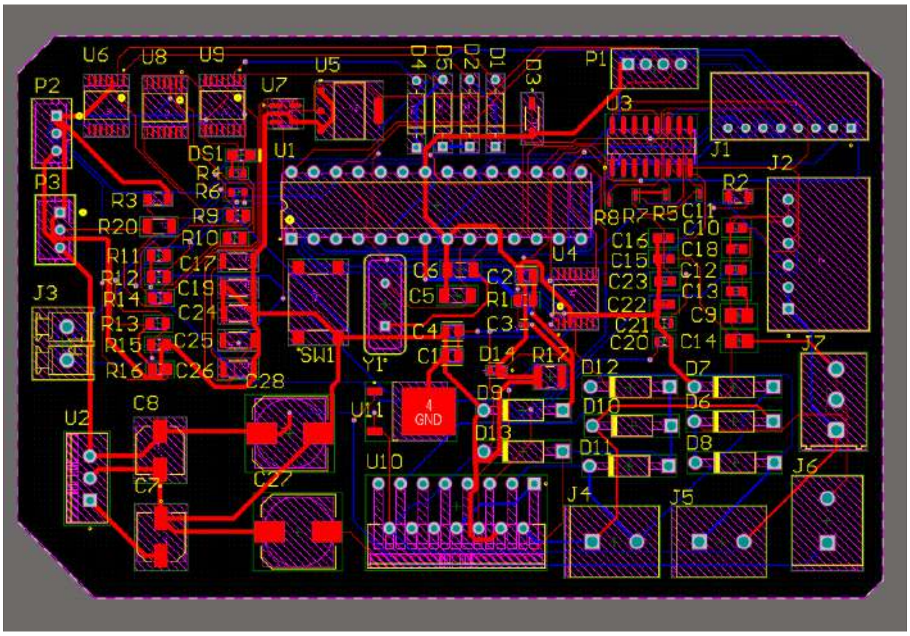


Figure 106: Initially Designed PCB

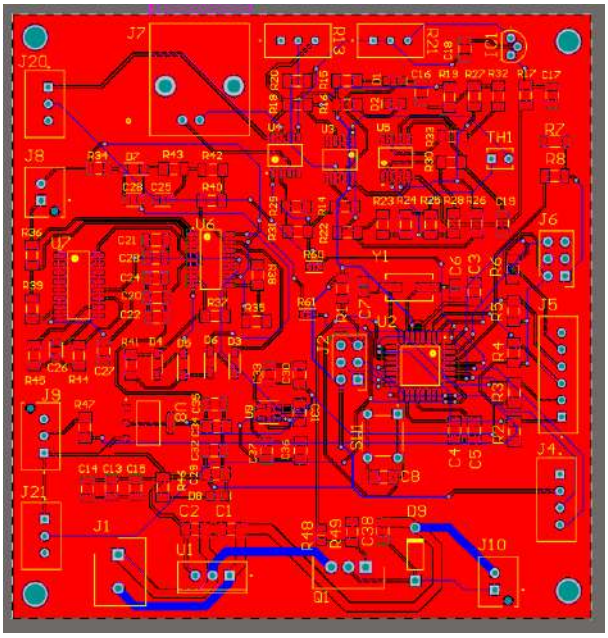


Figure 107: Finalized PCB Design

Simultaneously, we continued to refine the SolidWorks design for the display component of our system. This included making detailed adjustments to ensure the display was well-integrated into the overall design, both functionally and aesthetically. We also procured the necessary components for the display, ensuring all parts were ready for the next phase of assembly and testing.

2.11 22 April - 28 April 2024

After finalizing the PCB design, our focus shifted to creating a detailed 3D model for the main unit of the system. Utilizing the given dimensions of the PCB and the precise placements of its components, we began the 3D modeling process. This step was crucial for ensuring that all components would fit correctly within the enclosure and that the overall design was both functional and aesthetically pleasing. The 3D model helped us visualize the final product and identify any potential issues with component placement or spacing.

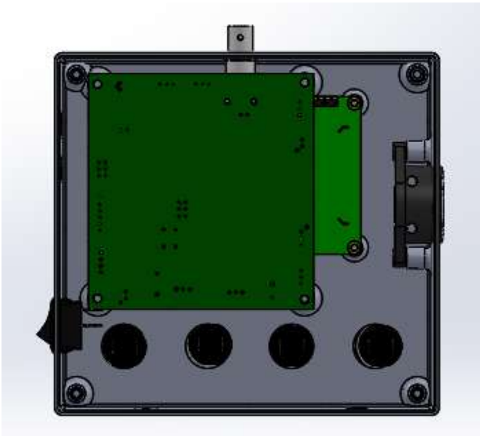


Figure 108: Finalized PCB housing enclosure

Figure 109: Finalized user interface enclosure

In parallel with the 3D modeling, we began developing the detailed code for the system. This involved writing comprehensive and optimized code to handle all functionalities of the aquaculture tank monitoring system. To verify the accuracy and effectiveness of the code, we conducted breadboard implementations.

2.12 29 April - 5 May 2024

During this week, the PCB and the ordered components arrived. We conducted a continuity test to ensure the integrity and correctness of the PCB traces and connections.This test methodically assessed the integrity of PCB traces and connections, ensuring that every pathway and circuit was properly established and free from defects. By meticulously probing each connection point and trace pathway, we aimed to identify any potential issues early on and preemptively address them to prevent costly delays

in the development process.
Upon receiving the components, our team meticulously unpacked and inspected each item to ensure they met our stringent quality standards and specifications.

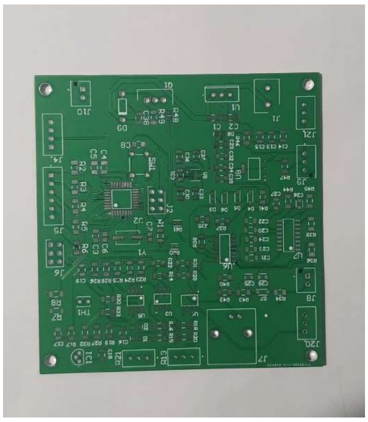


Figure 110: Printed circuit board (front)

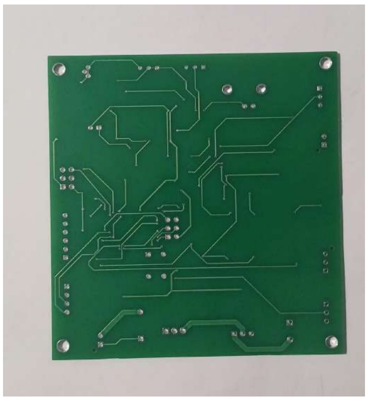


Figure 111: Printed circuit board (back)

We tested the 3 sensors and integrated their functions into the code.

2.13 6 May - 12 May 2024

During this week, we undertook a thorough redesign of the feeder due to concerns over its initial design being excessively bulky and costly. Recognizing the need for a more streamlined and economically viable solution, we made the strategic decision to transition from the originally planned plastic stand to a robust metal stand.

This adjustment not only addresses the durability concerns posed by the previous design but also aligns more closely with our project's budgetary considerations.

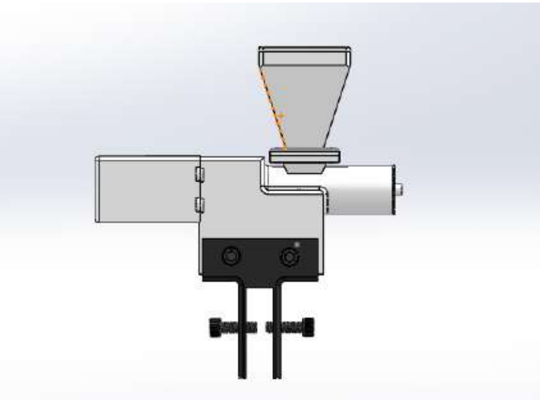


Figure 112: Final feeder design

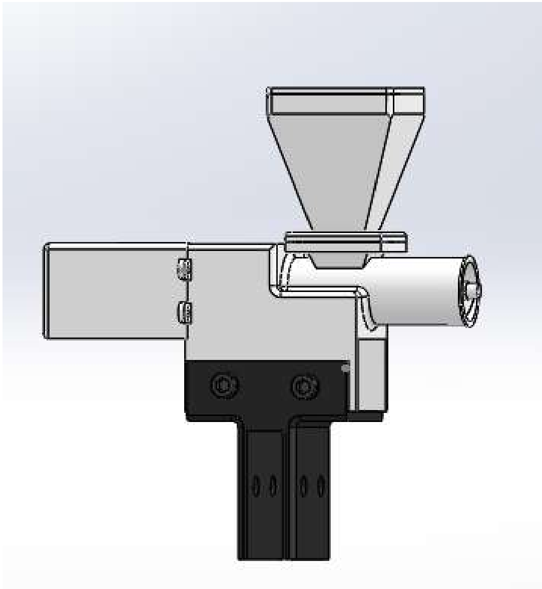


Figure 113: Final feeder design assembly

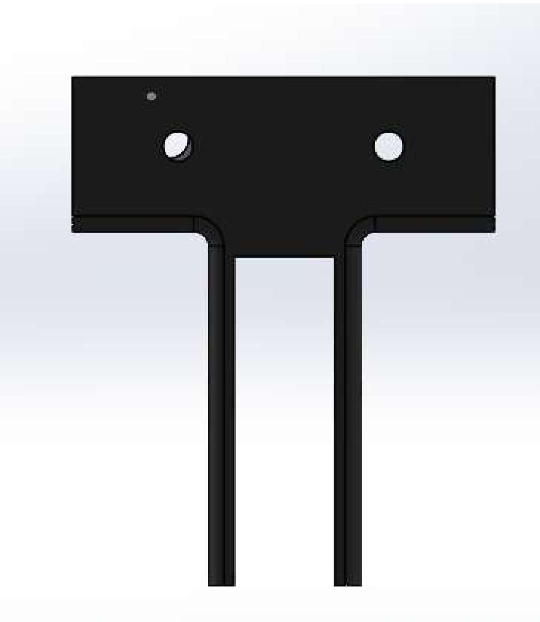


Figure 114: New stand design

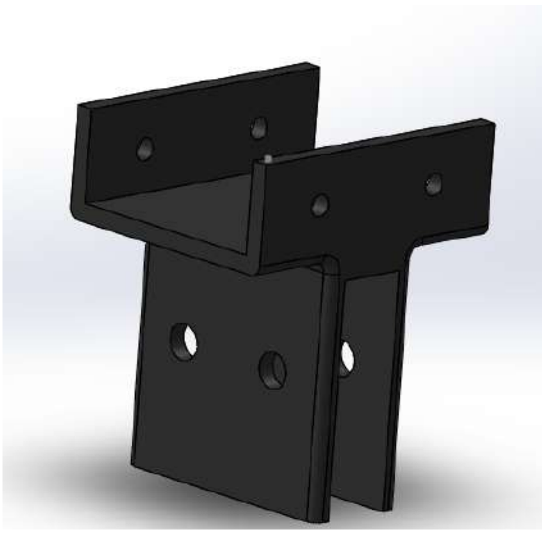


Figure 115: New stand design

2.14 6 May - 12 May 2024

We updated our Arduino code to retrieve time from an NTP server using a WiFi module. Additionally, we conducted comprehensive testing of pH, temperature, and TDS sensor functionalities and integrated them into our main code base.

```
// Function to check sensors and handle alarms
void check_sensors() {
  sensors.requestTemperatures();
  float tempC = sensors.getTempCByIndex(0);

  int phValue = analogRead(PhSensorPin);
  phTemp += phValue;
  for (int i = 0; i < 9; i++) {
    phBufferArr[i] = phBufferArr[i + 1];
  }
  phBufferArr[9] = phTemp / 10;

  // Sort array for median value
  for (int i = 0; i < 10; i++) {
    for (int j = i + 1; j < 10; j++) {
      if (phBufferArr[i] > phBufferArr[j]) {
        int temp = phBufferArr[i];
        phBufferArr[i] = phBufferArr[j];
        phBufferArr[j] = temp;
      }
    }
  }
  float phAverage = 0;
  for (int i = 2; i < 8; i++) {
    phAverage += phBufferArr[i];
  }
  phAverage = (float)phAverage * 5.0 / 1024 / 6;
  float phValueCalc = -5.70 * phAverage + phCalibrationValue;

  int tdsAnalog = analogRead(TdsSensorPin);
  tdsAnalogBuffer[tdsAnalogBufferIndex] = tdsAnalog;
  tdsAnalogBufferIndex++;
  if (tdsAnalogBufferIndex >= SCOUNT) {
    tdsAnalogBufferIndex = 0;
  }
  for (int i = 0; i < SCOUNT; i++) {
    tdsAnalogBufferTemp[i] = tdsAnalogBuffer[i];
  }
  for (int i = 0; i < SCOUNT - 1; i++) {
    for (int j = i + 1; j < SCOUNT; j++) {
      if (tdsAnalogBufferTemp[i] > tdsAnalogBufferTemp[j]) {
        int temp = tdsAnalogBufferTemp[i];
        tdsAnalogBufferTemp[i] = tdsAnalogBufferTemp[j];
        tdsAnalogBufferTemp[j] = temp;
      }
    }
  }
  tdsAverageVoltage = 0;
  for (int i = 8; i < 22; i++) {
    tdsAverageVoltage += tdsAnalogBufferTemp[i];
  }
  tdsAverageVoltage = tdsAverageVoltage / 14;
  tdsAverageVoltage = tdsAverageVoltage * (float)VREF / 1024.0;
  tdsValue = (133.42 * tdsAverageVoltage * tdsAverageVoltage * tdsAverageVoltage -
      255.86 * tdsAverageVoltage * tdsAverageVoltage + 857.39 * tdsAverageVoltage) *
      (1.0 + 0.02 * (temperature - 25.0));

  // Check temperature
  if (tempC > h_temp || tempC < l_temp) {
    should_ring_alarm = true;
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("Temp Alarm:");
    lcd.setCursor(0, 1);
    lcd.print(tempC);
    lcd.print("C");
    delay(1000);
```

```
  }

  // Check pH
  if (phValueCalc > phMaxValue || phValueCalc < phMinValue) {
    should_ring_alarm = true;
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("pH Alarm:");
    lcd.setCursor(0, 1);
    lcd.print(phValueCalc);
    delay(1000);
  }

  // Check TDS
  if (tdsValue > tdsMaxValue || tdsValue < tdsMinValue) {
    should_ring_alarm = true;
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("TDS Alarm:");
    lcd.setCursor(0, 1);
    lcd.print(tdsValue);
    delay(1000);
  }

  // Ring alarm if any sensor is out of range
  if (should_ring_alarm) {
    ring_alarm();
     int pressed = wait_for_button_press();
    if(pressed==PB_cancel){
      should_ring_alarm = false;
  }
}
```

Listing 19: Function to check if any sensor triggers a warning range

2.15 June 21 - June 27 2024

During this week, we focused on integrating and enhancing the final Arduino code. Our efforts were centered on ensuring the system's full functionality, including real-time monitoring of temperature, pH, and TDS values, and incorporating an alarm system to alert when these levels exceed predefined thresholds. We implemented user interface controls such as buttons for up, down, back, cancel, and motor control. Additionally, we integrated an LCD display with an I2C module and enabled WiFi communication for accurate time synchronization and remote motor control based on timing. Throughout the process, we encountered compatibility issues between some libraries, which required us to modify the code and replace certain libraries to achieve a seamless integration.
By the end of the week, we reviewed the 3D designs of our enclosure parts and sent them for 3D printing. Concurrently, we collaborated with a metal workshop to custom-make the stand for our system.

2.16 June 28- July 6 2024

This week was dedicated to the assembly and testing of physical components. We received the 3D printed and metal enclosure parts, and upon assembling them, we verified the dimensions and made necessary adjustments. We also procured additional parts such as screws and other hardware required for the final assembly. Subsequently, we proceeded with soldering the entire PCB, ensuring all connections were secure and functional. This phase was crucial in bringing together all the individual components and validating the overall design.

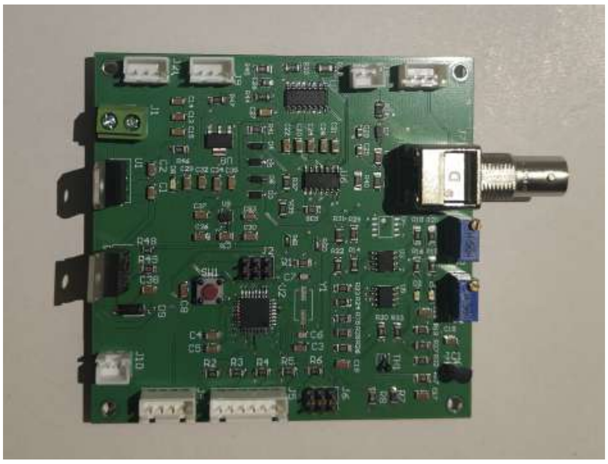


Figure 116: The Soldered PCB

Figure 117: User Interface Enclosure

Figure 118: Feeder Enclosure

2.17 July 16 - July 22 2024

This week, we began the process of converting our Arduino code to AVR-compatible code using C++ in Microchip Studio. This involved translating the main code to operate at the register level for improved efficiency and made it Arduino-independent for industrial standards. We custom-made several libraries, such as the UART and the delay function to replace the Arduino-specific delay function. We also incorporated some existing libraries, such as sensor libraries, ensuring to credit the original authors. Additionally, we made updates to the design documentation to accurately reflect the changes and improvements done within the last weeks.

3 Appendix B : Previously Used Arduino Code

```
    #include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#include <NTPClient.h>
#include <WiFiEspAT.h>
//#include <SoftwareSerial.h>

// Define LCD pins
#define LCD_SDA PC4
#define LCD_SCL PC5

// Pushbuttons
#define PB_cancel PD2
#define PB_OK PD3
#define PB_UP PD4
#define PB_DOWN PD5

#define MOTOR_BUTTON PD6

// Buzzer pin
#define BUZZER PB3

// Feeder Motor pin
#define MOTOR_PIN PD7

// TDS pin
#define TdsSensorPin PC2

// PH pin
#define PhSensorPin PC3

// Temp pin
#define ONE_WIRE_BUS PB0

// WiFi pins
#define ESP_TX PD0
#define ESP_RX PD1

//SoftwareSerial espSerial(ESP_TX, ESP_RX);

// Ranges
int h_temp = 25;
int l_temp = 2;

float phMinValue = 6.5;
float phMaxValue = 8.0;

float tdsMinValue = 300;
float tdsMaxValue = 1000;

// WiFi and NTP settings
const char* ssid = "YourSSID";
const char* password = "YourPassword";
const char* ntpServerName = "pool.ntp.org";

WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, ntpServerName);

// Motor control variables
int motor_hour = 0;             // Hour for motor activation
int motor_minute = 0;           // Minute for motor activation
bool motor_triggered = false;   // Flag to track motor activation
int motor_duration = 10;        // Default motor rotation duration in seconds

// Menu
int current_mode = 0, max_modes = 4;  // Increased max_modes for additional sensor
String modes[] = { "1- Temp range", "2- pH range", "3- TDS range", "4- Set Alarm 2" };

```

```
// For temperature sensor
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

// For pH sensor
float phCalibrationValue = 21.34;
int phBufferArr[10], phTemp;

// For TDS sensor
#define VREF 5.0
#define SCOUNT 30
int tdsAnalogBuffer[SCOUNT];
int tdsAnalogBufferTemp[SCOUNT];
int tdsAnalogBufferIndex = 0;
int tdsCopyIndex = 0;
float tdsAverageVoltage = 0;
float tdsValue = 0;
float temperature = 16;  // Initial temperature for compensation

// Initialize LCD object
LiquidCrystal_I2C lcd(0x27, 16, 2);  // Adjust the address and dimensions according to
    your LCD module

bool should_ring_alarm = false;  // Flag to control alarm ringing

// Notes for the buzzer melody
int notes[] = { 262, 294, 330, 349, 392, 440, 494 };

// Number of notes in the melody
int n_notes = 7;

void setup() {
  Serial.begin(9600);  //init_uart
  pinMode(PB_cancel, INPUT);
  pinMode(PB_OK, INPUT);
  pinMode(PB_UP, INPUT);
  pinMode(PB_DOWN, INPUT);
  pinMode(MOTOR_BUTTON, INPUT);

  pinMode(TdsSensorPin, INPUT);  // Set TDS sensor pin as input
  pinMode(PhSensorPin, INPUT);   // Set pH sensor pin as input
  sensors.begin();               // Initialize the temperature sensor

  pinMode(BUZZER, OUTPUT);  // Set buzzer pin as output
  pinMode(MOTOR_PIN, OUTPUT);

  // Initialize LCD
  lcd.init();        // initialize the lcd
  lcd.backlight();   // Turn on the backlight

  // Connect to WiFi
  connectToWiFi();

  // Initialize time client
  timeClient.begin();

  sensors.begin();
  lcd.clear();
  print_line("welcome!", 0, 0, 2);  // Assuming print_line is adjusted for LCD
  delay(2000);                      //delay_ms
  lcd.clear();
}

void loop() {
  update_time_with_check_motor();
  check_sensors();
  if (digitalRead(PB_OK) == LOW) {  //input
    delay(200);                     //delay_ms
    go_to_menu();
  }
}

void print_line(String text, int col, int row, int txt_size) {
```

```
  lcd.setCursor(col, row);
  lcd.print(text);
}

void connectToWiFi() {
  // Initialize the software serial connection with the ESP-01 module
  //espSerial.begin(9600);

  // Initialize the WiFiEsp library
  //WiFi.init(&espSerial);

  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("WiFi connected");
}

// Function to update time and check for motor activation
void update_time_with_check_motor() {
  timeClient.update();
  int hours = timeClient.getHours();
  int minutes = timeClient.getMinutes();

  // Print time on LCD
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print(hours);
  lcd.print(":");
  lcd.print(minutes);

  //Check if the manual control is pressed for the motor
  int pressed = wait_for_button_press();
  if (pressed = MOTOR_PIN){
    // If the Motor is working when MOTOR_BUTTON is pressed, turn it off
    if (digitalRead(MOTOR_PIN) == HIGH){
      // Start the motor
      digitalWrite(MOTOR_PIN, HIGH);
      motor_triggered = true;
      lcd.clear();
      lcd.setCursor(0., 0.);

      lcd.setCursor(6, 0);
      lcd.print("Motor On");

    // If the Motor is off when MOTOR_BUTTON is pressed, turn it on
    } else if (digitalRead(MOTOR_PIN) == LOW){
      // Start the motor
      digitalWrite(MOTOR_PIN, HIGH);
      motor_triggered = true;
      lcd.clear();
      lcd.setCursor(0., 0.);

      lcd.setCursor(6, 0);
      lcd.print("Motor On");

    }
  }

  // Check if it's time to activate the motor
  if (!motor_triggered && hours == motor_hour && minutes == motor_minute) {
    // Start the motor
    digitalWrite(MOTOR_PIN, HIGH);
    motor_triggered = true;
    lcd.clear();
    lcd.setCursor(0., 0.);

    lcd.setCursor(6, 0);
    lcd.print("Motor On");

```

```
    delay(motor_duration * 1000);  // Run motor for motor_duration seconds
    digitalWrite(MOTOR_PIN, LOW);  // Stop the motor
    motor_triggered = false;
    lcd.clear();
    lcd.setCursor(0., 0.);

    lcd.setCursor(6, 0);
    lcd.print("Motor Off");
  }

  // Check if motor should be stopped
  if (motor_triggered && (hours != motor_hour || minutes != motor_minute)) {
    // Stop the motor
    digitalWrite(MOTOR_PIN, LOW);
    motor_triggered = false;
    lcd.clear();
    lcd.setCursor(0., 0.);
    lcd.setCursor(6, 0);
    lcd.print("Motor Off");
  }
}

int wait_for_button_press() {
  while (true) {
    if (digitalRead(PB_UP) == LOW) {
      return PB_UP;
    } else if (digitalRead(PB_DOWN) == LOW) {
      return PB_DOWN;
    } else if (digitalRead(PB_OK) == LOW) {
      return PB_OK;
    } else if (digitalRead(PB_cancel) == LOW) {
      return PB_cancel;
    }else if (digitalRead(MOTOR_BUTTON) == LOW) {
      return MOTOR_BUTTON;
    }
  }
}

void go_to_menu() {
  while (digitalRead(PB_cancel) == HIGH) {
    lcd.clear();
    print_line(modes[current_mode], 0, 0, 2);
    int pressed = wait_for_button_press();
    delay(1000);

    if (pressed == PB_UP) {
      delay(200);
      current_mode -= 1;
      if (current_mode < 0) {
        current_mode = max_modes - 1;
      }
    } else if (pressed == PB_DOWN) {
      delay(200);
      current_mode += 1;
      current_mode = current_mode % max_modes;
    } else if (pressed == PB_OK) {
      delay(200);
      run_mode(current_mode);
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }
  lcd.clear();
}

// Function to set motor activation time
void set_motor_activation_time() {
  int entered_hour = 0;
  int entered_minute = 0;

  while (true) {
    lcd.clear();
```

```
    lcd.setCursor(0., 0.);

    lcd.setCursor(6, 0);
    lcd.print("Set Motor Time:");
    lcd.setCursor(0, 1);
    lcd.print(String(entered_hour, DEC) + ":" + String(entered_minute, DEC));


    int pressed = wait_for_button_press();
    if (pressed == PB_UP) {
      delay(200);
      entered_minute = (entered_minute + 1) % 60;
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_minute = (entered_minute - 1 + 60) % 60;
    } else if (pressed == PB_OK) {
      delay(200);
      motor_hour = entered_hour;
      motor_minute = entered_minute;
      lcd.clear();

      lcd.setCursor(0., 0.);

      lcd.setCursor(6, 0);
      lcd.print("Motor Time Set");
      lcd.setCursor(0, 1);
      lcd.print(String(motor_hour, DEC) + ":" + String(motor_minute, DEC));

      delay(2000);
      break;
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
    delay(100);
  }
}

// Function to set motor duration
void set_motor_duration() {
  int entered_duration = motor_duration;

  while (true) {
    lcd.clear();
    lcd.setCursor(0., 0.);
    lcd.setCursor(6, 0);
    lcd.print("Set Motor Duration:");
    lcd.setCursor(0, 1);
    lcd.print(String(entered_duration, DEC) + " sec");


    int pressed = wait_for_button_press();
    if (pressed == PB_UP) {
      delay(200);
      entered_duration += 1;
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_duration = max(entered_duration - 1, 1);  // Ensure duration is at least 1
         second
    } else if (pressed == PB_OK) {
      delay(200);
      motor_duration = entered_duration;
      lcd.clear();
      lcd.setCursor(0., 0.);

      lcd.setCursor(6, 0);
      lcd.print("Motor Duration Set");
      lcd.setCursor(0, 1);
      lcd.print(String(motor_duration, DEC) + " sec");

      delay(2000);
      break;
    } else if (pressed == PB_cancel) {
```

```
      delay(200);
      break;
    }
    delay(100);
  }
}

// Function to display motor status
void display_motor_status() {
  lcd.clear();
  lcd.setCursor(0., 0.);
  lcd.print("Motor ");
  lcd.setCursor(6, 0);
  lcd.print(motor_triggered ? "On" : "Off");
  delay(2000);
}

void set_temp_range() {
  int entered_h_temp = 27;
  while (true) {
    lcd.clear();
    print_line("high temp: " + String(entered_h_temp), 0, 0, 2);
    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay(200);
      entered_h_temp += 1;
      entered_h_temp = entered_h_temp % 31;
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_h_temp -= 1;
      if (entered_h_temp < 25) {
        entered_h_temp = 30;
      }
    } else if (pressed == PB_OK) {
      delay(200);
      h_temp = entered_h_temp;
      break;
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }

  int entered_l_temp = 27;
  while (true) {
    lcd.clear();
    print_line("lowest temp: " + String(entered_l_temp), 0, 0, 2);
    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay(200);
      entered_l_temp += 1;
      entered_l_temp = entered_l_temp % 31;
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_l_temp -= 1;
      if (entered_l_temp < 25) {
        entered_l_temp = 30;
      }
    } else if (pressed == PB_OK) {
      delay(200);
      l_temp = entered_l_temp;
      break;
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }
}

void set_ph_range() {
  float entered_ph_min = 6.5;
```

```
  while (true) {
    lcd.clear();
    print_line("pH Min: " + String(entered_ph_min), 0, 0, 2);
    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay(200);
      entered_ph_min += 0.1;
      if (entered_ph_min > 14.0) {
        entered_ph_min = 6.5;
      }
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_ph_min -= 0.1;
      if (entered_ph_min < 0.0) {
        entered_ph_min = 0.0;
      }
    } else if (pressed == PB_OK) {
      delay(200);
      phMinValue = entered_ph_min;
      break;
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }

  float entered_ph_max = 8.0;
  while (true) {
    lcd.clear();
    print_line("pH Max: " + String(entered_ph_max), 0, 0, 2);
    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay(200);
      entered_ph_max += 0.1;
      if (entered_ph_max > 14.0) {
        entered_ph_max = 8.0;
      }
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_ph_max -= 0.1;
      if (entered_ph_max < 0.0) {
        entered_ph_max = 0.0;
      }
    } else if (pressed == PB_OK) {
      delay(200);
      phMaxValue = entered_ph_max;
      break;
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }
}

void set_tds_range() {
  int entered_tds_min = 300;
  while (true) {
    lcd.clear();
    print_line("TDS Min: " + String(entered_tds_min), 0, 0, 2);
    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay(200);
      entered_tds_min += 10;
      if (entered_tds_min > 2000) {
        entered_tds_min = 300;
      }
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_tds_min -= 10;
      if (entered_tds_min < 0) {
```

```
        entered_tds_min = 0;
      }
    } else if (pressed == PB_OK) {
      delay(200);
      tdsMinValue = entered_tds_min;
      break;
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }

  int entered_tds_max = 1000;
  while (true) {
    lcd.clear();
    print_line("TDS Max: " + String(entered_tds_max), 0, 0, 2);
    int pressed = wait_for_button_press();

    if (pressed == PB_UP) {
      delay(200);
      entered_tds_max += 10;
      if (entered_tds_max > 2000) {
        entered_tds_max = 1000;
      }
    } else if (pressed == PB_DOWN) {
      delay(200);
      entered_tds_max -= 10;
      if (entered_tds_max < 0) {
        entered_tds_max = 0;
      }
    } else if (pressed == PB_OK) {
      delay(200);
      tdsMaxValue = entered_tds_max;
      break;
    } else if (pressed == PB_cancel) {
      delay(200);
      break;
    }
  }
}

void run_mode(int mode) {
  if (mode == 0) {
    set_temp_range();
  } else if (mode == 1) {
    set_ph_range();
  } else if (mode == 2) {
    set_tds_range();
  } else if (mode == 3) {
    int x = 1;  // Placeholder for future functionalities
  }
}

void check_sensors() {
  sensors.requestTemperatures();
  float temperatureC = sensors.getTempCByIndex(0);

  if (temperatureC > h_temp) {
    print_line("TEMP HIGH", 0, 40, 1);
    ring_alarm(BUZZER);
    delay(200);

  } else if (temperatureC < l_temp) {
    print_line("TEMP LOW", 0, 40, 1);
    ring_alarm(BUZZER);
    delay(200);
  }

  float phValue = analogRead(PhSensorPin);
  phValue = (float)phValue * 5.0 / 1024.0;
  phValue = 3.5 * phValue + phCalibrationValue;

  if (phValue > phMaxValue || phValue < phMinValue) {
```

```
    print_line("pH OUT OF RANGE", 0, 50, 1);
    ring_alarm(BUZZER);
    delay(200);
  }

  float tdsValue = analogRead(TdsSensorPin);
  tdsAnalogBuffer[tdsAnalogBufferIndex] = tdsValue;
  tdsAnalogBufferIndex++;
  if (tdsAnalogBufferIndex == SCOUNT) {
    tdsAnalogBufferIndex = 0;
    for (int i = 0; i < SCOUNT; i++) {
      tdsAverageVoltage += tdsAnalogBuffer[i];
    }
    tdsAverageVoltage /= SCOUNT;
    tdsValue = (133.42 * pow(tdsAverageVoltage / VREF, 3) - 255.86 * pow(
        tdsAverageVoltage / VREF, 2) + 857.39 * (tdsAverageVoltage / VREF)) * 0.5;
  }

  if (tdsValue > tdsMaxValue || tdsValue < tdsMinValue) {
    print_line("TDS OUT OF RANGE", 0, 50, 1);
    ring_alarm(BUZZER);
    delay(200);
  }
}

void ring_alarm(int buzzerPin) {
  bool break_happened = false;
  while (should_ring_alarm && digitalRead(PB_cancel) == HIGH) {
    for (int i = 0; i < n_notes; i++) {
      if (digitalRead(PB_cancel) == LOW) {
        delay(200);
        should_ring_alarm = false;  // Set flag to false
        break_happened = true;
        noTone(buzzerPin);  // Turn off the buzzer
        break;
      }
      tone(buzzerPin, notes[i]);
      delay(500);
      noTone(buzzerPin);
      delay(2);
    }
  }

  // Add a delay after the loop to prevent immediate restart
  delay(1000);
}
```

Listing 20: Previously Used Arduino Code

This report is reviewed by,

(D.E. U Jayathilaka) D. E. U. Jayathilake	Dasuni. H.M.D.P. Herath H. M. D. P. Herath
210037U - Amarathunga D.N. D. N. Amarathunga	210446J - Pasira I.P.M. I. P. M. Pasira