# PetCare System

ENG5220: Real Time Embedded Programming

Team7: Bowen Ren 2818474r,  Zhiyu Ren 2869108r,

Yutong Wang 2682750w, Xiaoyu Zhang 2899211z

## 1 Team and Collaboration

| Name | GUID | Description |
|---|---|---|
| Bowen Ren | 2818474r | Unit test, main method, cloud server |
| Zhiyu Ren | 2869108r | Program logic, unit test, main method |
| Yutong Wang | 2682750w | Social media, build the device, cloud server |
| Xiaoyu Zhang | 2899211z | Unit test, GitHub management, report |

Table 1 collaboration

GitHub link: https://github.com/koinzh/PetCare

TikTok link: https://www.tiktok.com/@smartpetcaresyste?_t=8kk3llkyQGP&_r=1

## 2 Introduction

This project uses raspberry PI to make an automatic pet-care system, which has a feeder with real-time monitoring of pet movement and food weight, a water feeder, and transfer the data of feeding activities to aliyun cloud server.

In this system, the ultrasonic sensor detects the distance to the pet, and the feeder starts working when the pet is close. The weight sensor detects the weight of the food in the bowl. When the food weight is less than threshold, the motor starts to rotate the baffle 90 degrees to replenish the food. When the food weight is greater than threshold, the motor rotates backward. User can change the weight threshold through app.

The water sensor detects the level of water in the container. When the water level is less than the setting height, start the water pump to draw water from the water storage tank and add it to the water feeding container. The water pump runs for 1 second.

## 3 Projects components

### 3.1 Configuration requirement

Raspberry Pi 3B+

C++

### 3.2 Equipment

Raspberry Pi 3B+

Power supply (MP1584)

Ultrasonic sensor (HC-SR04)

Weight sensor (HX711)

Step motor (28BYJ-48)

Relay (JQC-3FF-S-Z)

Water sensor

Water pump

Paper boxes

Plastic bottles

### 3.3 Circuit diagram

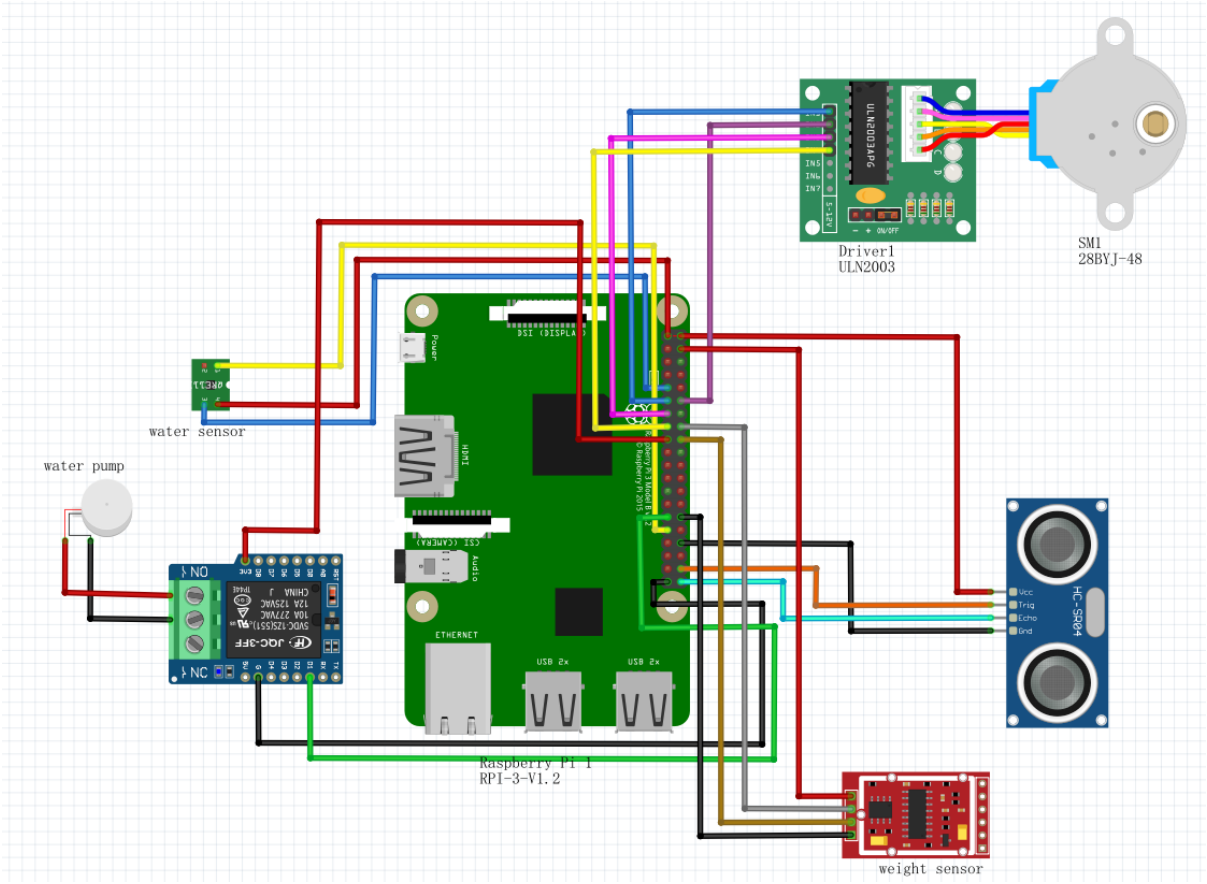


Figure 1 circuit diagram

### 3.4 Principle of sensor

#### 3.4.1 Step motor

This motor turns electrical signals into steps of movement. The number of signals decides how fast it goes and where it stops. Each signal makes the motor move a set amount.

#### 3.4.2 Ultrasonic sensor

A Raspberry Pi sends a short signal to start the sensor. The sensor then sends out sound waves and listens for them to bounce back. How long it takes for the sound to return tells us the distance.

#### 3.4.3 Weight sensor

This sensor has a wire that changes its resistance when stretched. The more it stretches, the more the resistance increases. It measures weight by turning these changes into electrical signal.

### 3.4.4 Water sensor

When a water can detect water levels and adjust its output voltage accordingly. This reliable sensor ensures precise monitoring of water levels, which is crucial for water-feeding systems.

### 3.4.5 Water pump

A miniature water pump serves as an active component in embedded systems, facilitating the flow of water through small conduits. It operates on electrical power, enabling precise control of water delivery.

## 4 Software

### 4.1 Software overview

Our PetCare system uses a Raspberry Pi to control everything. It connects to a cloud server for storing information. There are two main parts: the feeding system and the water system. The feeding system has three parts that work together: a Motor Controller to move the food, a Weight Sensor to check how much food is there, and an Ultrasonic Sensor to know when the pet is near. The water system has a Water Level Controller to keep the water at the right amount. The cloud system uses LinkSDK to help us watch and manage the system from anywhere. The Raspberry Pi makes sure all these parts talk to each other and to the cloud server. The structure of PetCare system is shown in Figure 2.

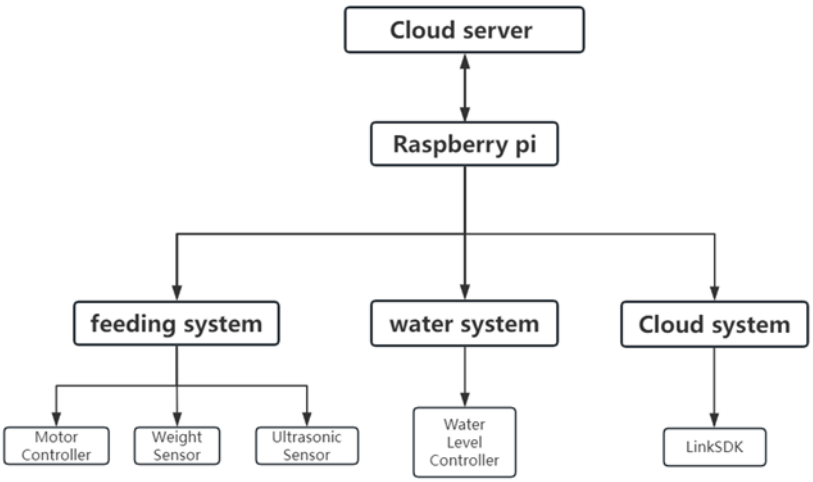


Figure 2 structure of the software components and their interactions

### 4.2 Code explanation

#### 4.2.1 Feeding system

#### (1) MotorController

The MotorController class is responsible for controlling a stepper motor connected to a Raspberry Pi via GPIO pins. It facilitates precise motor movements by executing a sequence of steps in either the forward or backward direction and offers functionalities to stop and reset motor operations.

The method that controls the sequence of operations to rotate the motor:

```
void MotorController::step(int totalSteps, int direction) {
    int stepCount = 8;
    int stepSequence[8][4] = {
        {1,0,0,0}, {1,1,0,0}, {0,1,0,0}, {0,1,1,0},
        {0,0,1,0}, {0,0,1,1}, {0,0,0,1}, {1,0,0,1}
    };
    for (int i = 0; i < totalSteps; ++i) {
        for (int s = 0; s < stepCount; ++s) {
            int idx = (direction > 0) ? s : (stepCount - 1 - s);
            setStep(stepSequence[idx][0],                stepSequence[idx][1],
stepSequence[idx][2], stepSequence[idx][3]);
        }
    }
}
```

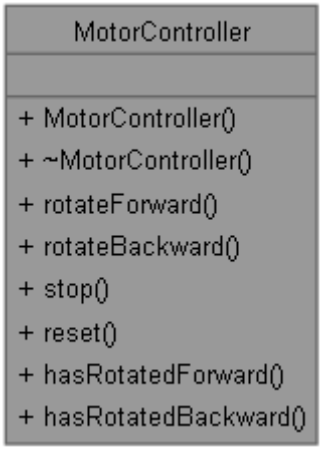


Figure 3 class diagram for MotorController

#### (2) WeightSensor

The WeightSensor class interfaces with a digital weight sensor to measure and report weight. It utilizes GPIO pins for signal communication and is calibrated using a specified coefficient. The sensor supports concurrent operations by reading weight in a separate thread.

The key part of the reading mechanism:

```
void WeightSensor::triggerRead() {
    if (!running) return;
   // int rawWeight = readWeight();
    //updateWeightHistory(rawWeight);
    latestWeight.store(readWeight());  // Update the latest weight
}
```

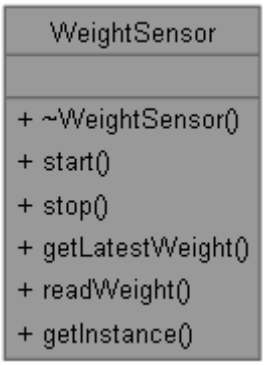


Figure 4 class diagram for WeightSensor

**(3) UltrasonicSensor**

The UltrasonicSensor class manages distance measurement using an ultrasonic sensor connected to a Raspberry Pi. It utilizes a trigger and echo mechanism to determine distances by sending out a pulse and measuring the time it takes for the echo to return.

The way the sensor triggers a pulse and waits for an echo:

```
void UltrasonicSensor::triggerPulse() {
    // generate a 10 microsecond high level pulse
    gpioTrigger(trigPin, 10, 1);
}


void UltrasonicSensor::waitForEcho() {
    std::unique_lock<std::mutex> lock(mtx);
    if (!cv.wait_for(lock, std::chrono::milliseconds(100), [this] { return
this->endTime > this->startTime; })) {
        std::cerr << "Timeout waiting for echo response." << std::endl;
    } else {
        if (distanceCallback) {
```

```
            float distance = (endTime - startTime) * 0.0343 / 2.0;
            distanceCallback(distance);
        }
    }
}
```

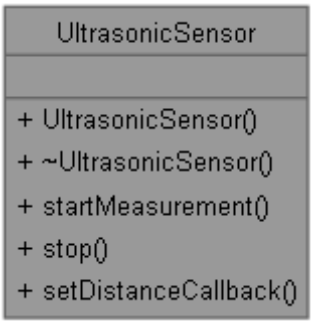


Figure 5 class diagram for UltrasonicSensor

### 4.2.2 Water system

The WaterLevelController class manages the water level in a container by controlling a motor. When the water level drops, the sensor triggers the motor to start, and after a set duration, it stops the motor to avoid overfilling.

Illustrate how the water level check is performed and triggers the motor control:

```
void WaterLevelController::checkWaterLevel(int gpio, int level, uint32_t tick)
{
    if (level == PI_HIGH) {
        startMotor();
    }            else            if            (motorRunning            &&
std::chrono::duration_cast<std::chrono::seconds>(
        std::chrono::steady_clock::now() - motorStartTime).count() >= 2) {
        stopMotor();
    }
}
```

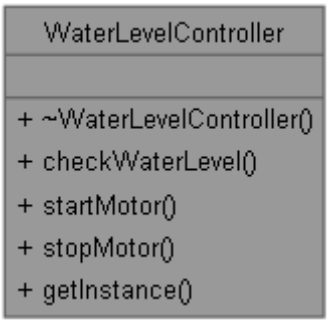


Figure 6 class diagram for Water system

### 4.2.3 Cloud system

A cloud server is built on cloud computing technology. It delivers computing resources and data storage services via the Internet. Users can rent these virtual machines according to their needs, thereby eliminating the requirement to purchase and maintain physical hardware themselves. This approach not only optimizes resource allocation but also significantly enhances cost efficiency. After doing a lot of research and combining various factors, we chose to use the Alibaba Cloud IoT platform to implement the remote data transmission function.

Alibaba Cloud IoT platform provides robust device connectivity, management, and data processing services for businesses. It supports various communication protocols like MQTT and HTTP, enabling easy device integration with the cloud. Key features of the platform include real-time data processing, a customizable rule engine, and stringent security measures to ensure secure data transmission and storage. Alibaba Cloud's IoT platform is widely used in smart manufacturing, smart cities, smart homes, and other fields, helping enterprises optimize operational efficiency and achieve digital transformation. Next, we will introduce our implementation process in detail by dividing it into two parts: the Raspberry Pi side and the cloud server side.

**(1) Raspberry Pi side**

The Alibaba Cloud IoT platform provides a device-side software development kit, Link SDK, which can be used to realise access to various devices. Link SDK currently only supports C, Java, Python, Node.js, Android, iOS, and the C++ we use is not included, and ultimately we can only choose to use the C Link SDK, which is the closest to C++, for porting and development.

Unlike the high encapsulation of SDKs in other languages, C Link SDK requires manual porting and code logic writing to make the device successfully accessible, which is a great challenge for us. We wrote an IOTConnect class modelled on the demo code in the toolkit to enable devices to access and interact with the cloud server. Since there are still many implementation differences between the original C program and the C++ program, we can only keep some public parameters, and public functions in order to ensure the functionality and smooth access.

(2) Cloud server side

The construction of the server side is relatively simple, in the Alibaba Cloud IoT platform, open a public instance, and add the required attributes, respectively, the read-only attributes Weight and Distance, which are updated in real time and displayed on the mobile phone app, as well as the read/write attribute Weightthreshold, which can be set by the user, and a read/write attribute switch for testing and debugging. After that, we open the Raspberry Pi registration and access the Alibaba Cloud Platform based on the MQTT protocol, and then we can use the API provided by the platform to report and receive JSON messages, so as to realise the information transfer between the platform and the Raspberry Pi. The following is a screenshot of the development of the Alibaba Cloud IoT platform. In addition, we have also carried out the development of mobile apps, simple UI design, to meet the user's remote control requirements.

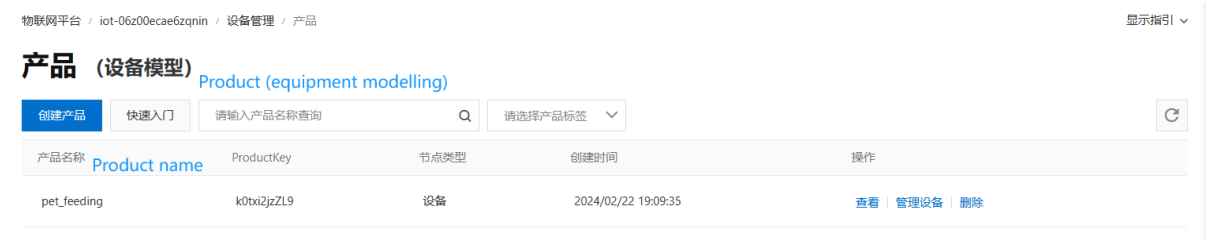


Figure 7 AliCloud Product Page

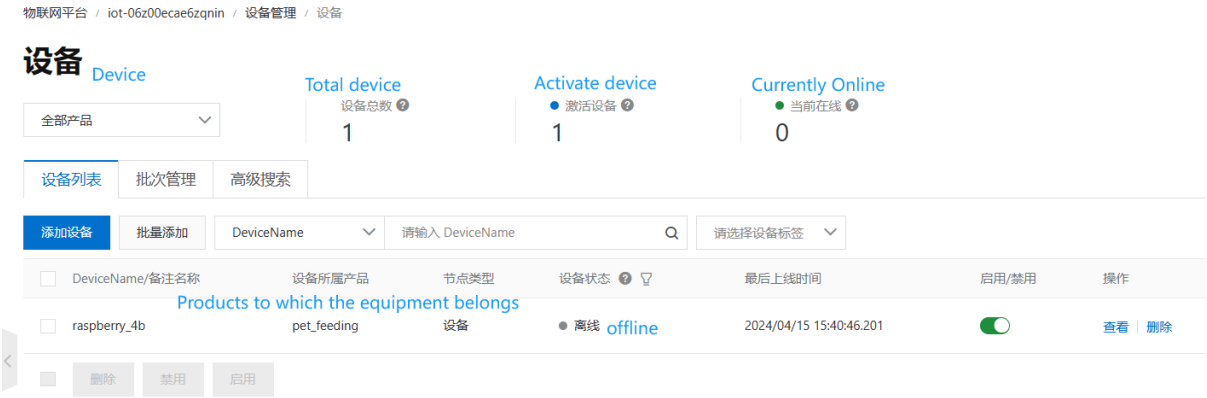


Figure 8 AliCloud Device Page

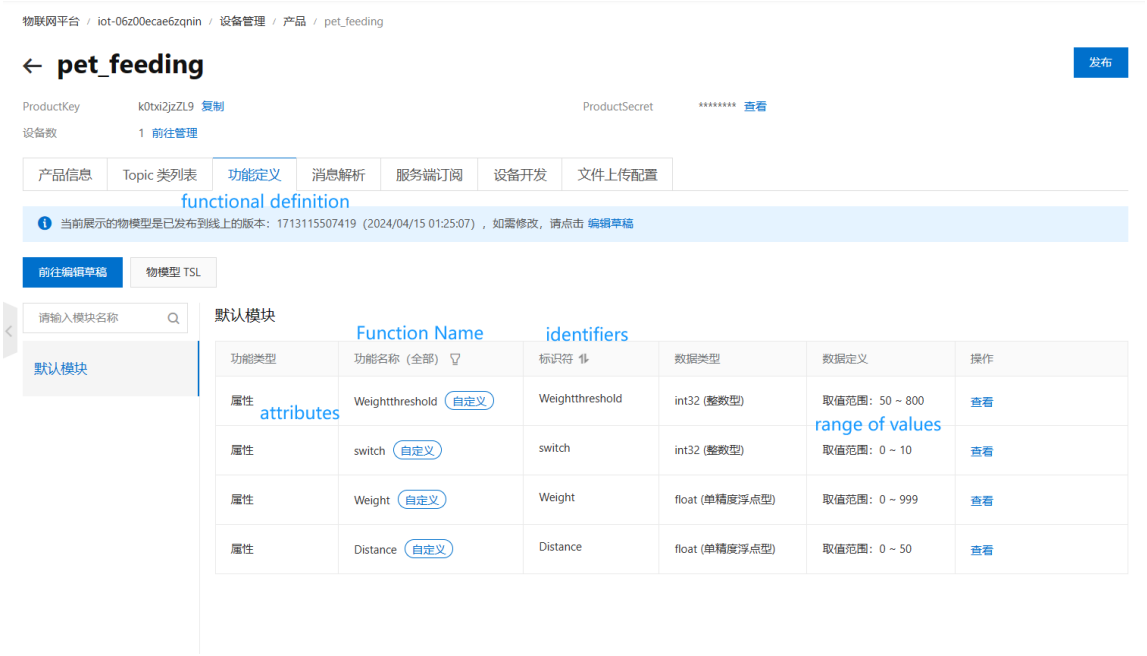


Figure 9 AliCloud Product Function Definition

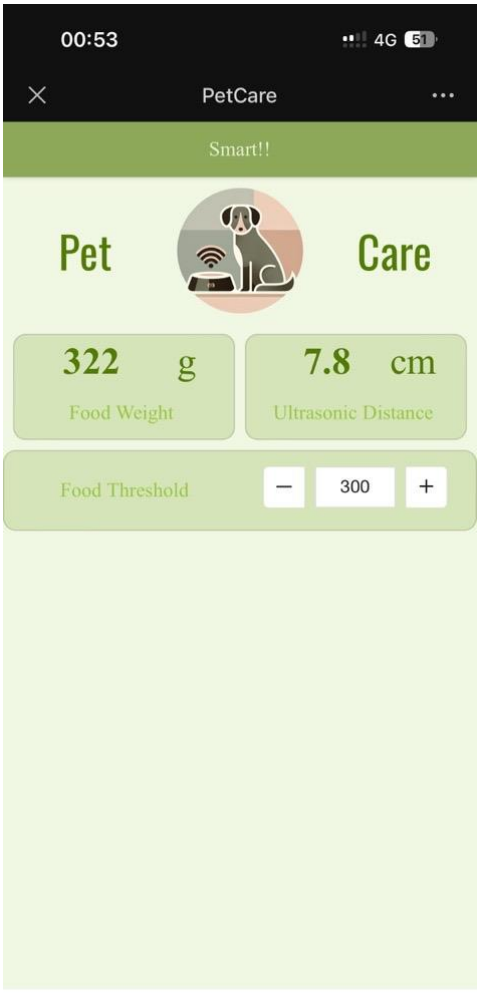


Figure 10 Mobile Application Testing

### 4.2.4 Integration

The main function orchestrates the operation of several interconnected components of an embedded system that monitors and controls pet care activities. It utilizes GPIO for interfacing with hardware, manages threads for concurrent operations, and employs callbacks for real-time monitoring and response.

The system starts by setting up GPIO and registering a signal handler to terminate the program upon receiving a signal like SIGINT. Instantiates MotorController, UltrasonicSensor, WaterLevelController, and WeightSensor with appropriate pin configurations and calibration settings.

For main loop and threads, WeightSensor continuously monitors the weight and updates its value, which is used to control motor movements. Water level monitoring runs in a separate thread to continuously check water levels and ensure the water supply is maintained. The UltrasonicSensor measures the distance. If the distance is below a set threshold and the weight is less than a certain value for three consecutive readings, it triggers the motor to rotate forward. If the weight meets or exceeds the threshold and the motor has previously rotated forward, it triggers the motor to rotate backward and then resets its state to allow future forward rotations. The system ignores invalid distance readings and ensures that motor actions only occur under appropriate conditions.

Upon termination, the system stops all sensors and motors, joins any running threads, and cleanly shuts down the GPIO infrastructure, ensuring all resources are properly released.

## 4.3 Testing

### 4.3.1 Unit testing

For software development projects, conducting unit tests is a critical step in ensuring that each component works as expected. Our approach is to write and execute unit tests for each component in the project. Doing so helps us to catch errors and problems early, avoiding major problems later in the software development process and improving the quality and reliability of the code. Figure 11 to Figure 14 show the test outputs.

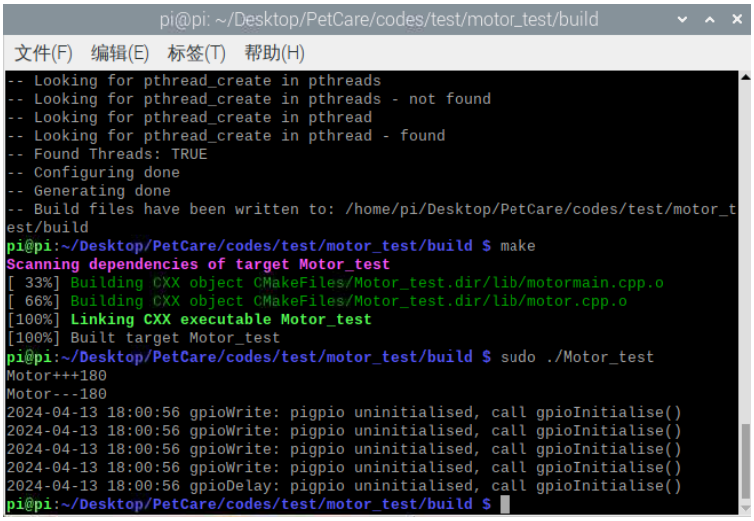


Figure 11 motor test

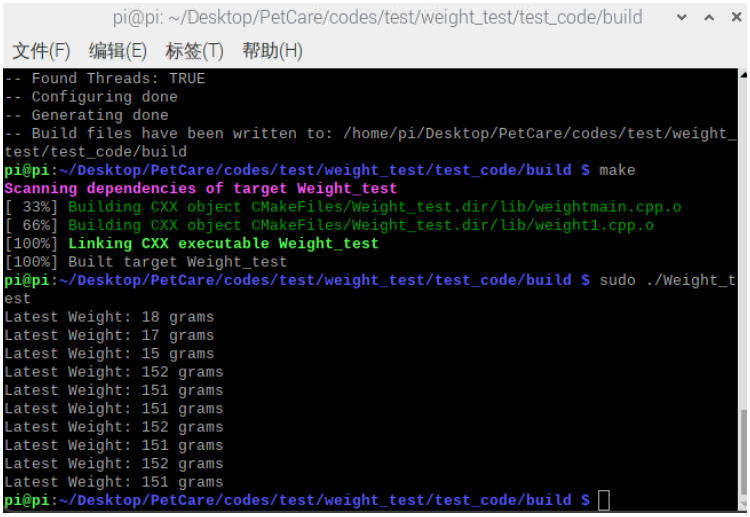


Figure 12 weight sensor test

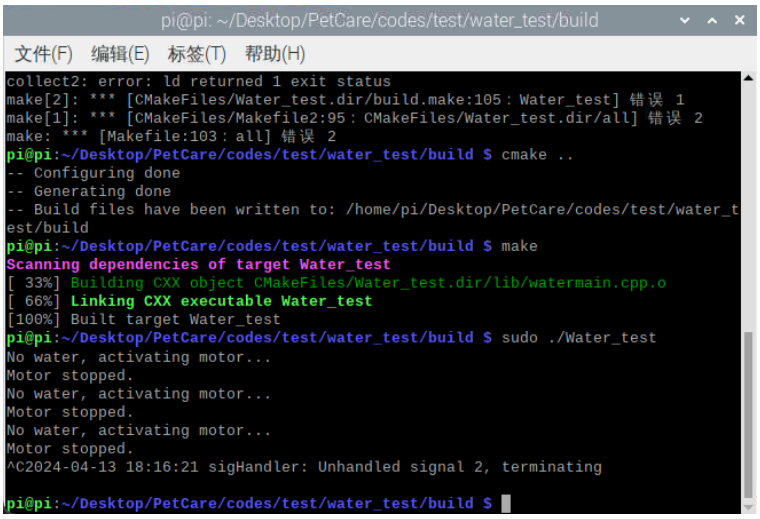


Figure 13 water test

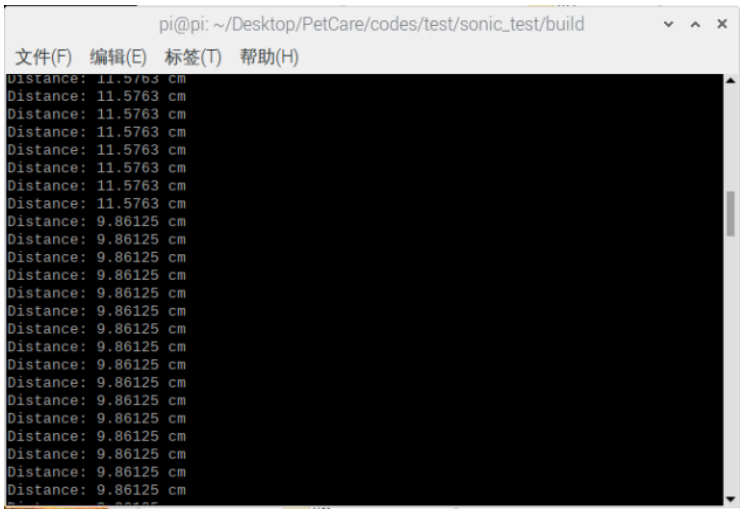


Figure 14 ultrasonic test

### 4.3.2 System testing

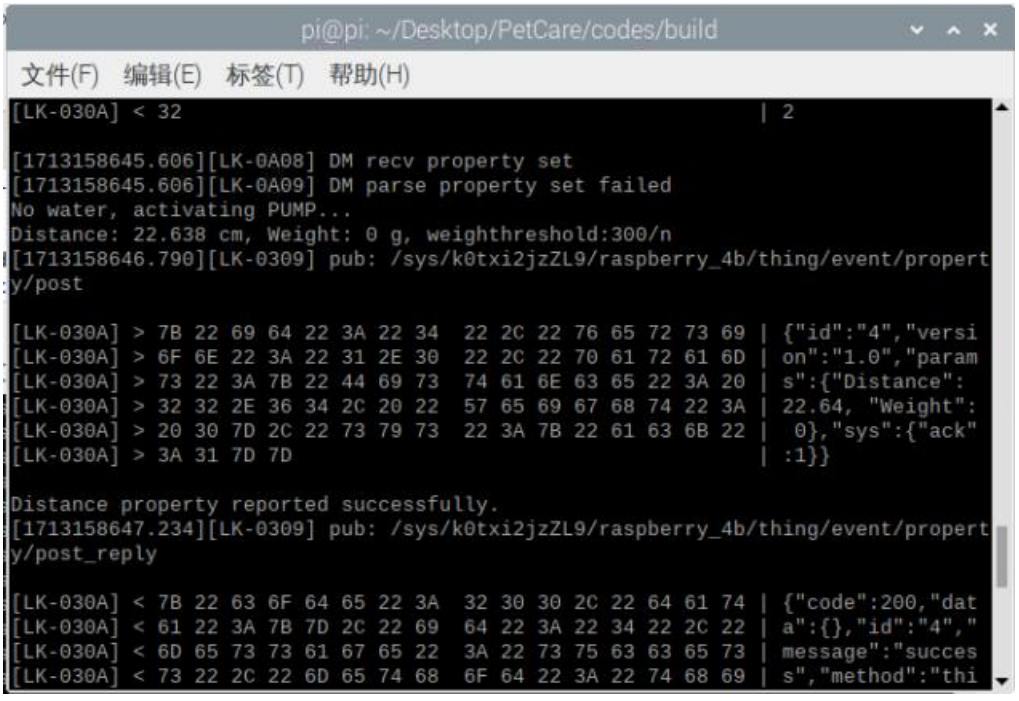


Figure 15 System Testing

## 5 Further Development

Due to time constraints, we have only completed the basic functions of the feeder only. Here are some features that can be improved in the future.

The current mobile app only allows for very simple remote interaction, we plan to develop a

full mobile app with more attributes and interaction logic to better suit the user's needs, such as manual feeding, fault alerts, etc. The app will allow pet owners to enter specific feeding instructions and customise the time and amount of food their pet receives. At the same time, the feeding and watering logs will be added to the database for storage, so that the user can better understand the operation of the feeder.

## 6 Project management

### 6.1 Timetable

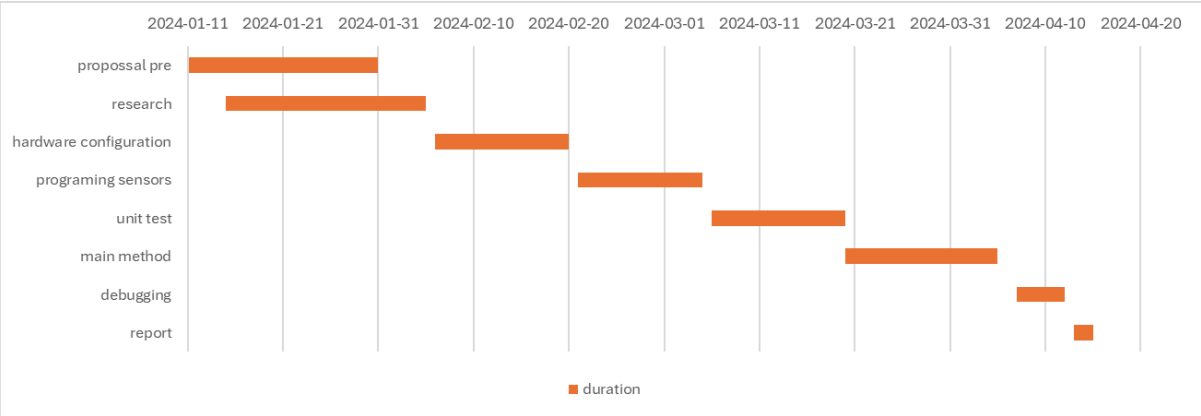


Figure 16 Timetable

### 6.2 Budget

| Item | Quantity | Price/ £ |
|---|---|---|
| Power supply(MP1584) | 1 | 7.99 |
| Ultrasonic sensor (HC-SR04) | 1 | 2.00 |
| Weight sensor (HX711) | 2 | 2.41 |
| Step motor (28BYJ-48) | 2 | 4.64 |
| Water sensor | 1 | 9.99 |
| Water pump | 1 | 3.99 |
| Total | 8 | 38.07 |

Table 2 Budget

# Appendix

motor.h

```
#ifndef MOTORCONTROLLER_H
#define MOTORCONTROLLER_H

#include <atomic>
#include "pigpio.h"

class MotorController {
public:
    MotorController(int pin1, int pin2, int pin3, int pin4);
    ~MotorController();
    void rotateForward(int steps);
    void rotateBackward(int steps);
    void stop();
    void reset();
    bool hasRotatedForward() const;
    bool hasRotatedBackward() const;

private:
    int pins[4];
    std::atomic<bool> rotatedForward;
    std::atomic<bool> rotatedBackward;
    void setStep(int w1, int w2, int w3, int w4);
    void step(int totalSteps, int direction);
    //Timer* timer;
};

#endif // MOTORCONTROLLER_H
```

motor.cpp

```
#include "motor.h"
#include <chrono>
#include <thread>
#include <functional>


MotorController::MotorController(int pin1, int pin2, int pin3, int pin4)
: rotatedForward(false), rotatedBackward(false) {
    pins[0] = pin1;
    pins[1] = pin2;
    pins[2] = pin3;
    pins[3] = pin4;
    for (int i = 0; i < 4; ++i) {
        gpioSetMode(pins[i], PI_OUTPUT);
```

```
    }


}



void MotorController::rotateForward(int steps) {
    if (!rotatedForward.exchange(true)) {
        step(steps, 1);
    }
    stop();
}

void MotorController::rotateBackward(int steps) {
    if (!rotatedBackward.exchange(true)) {
        step(steps, -1);
    }
    stop();
}

void MotorController::stop() {
    setStep(0, 0, 0, 0);
}

void MotorController::reset() {
    rotatedForward = false;
    rotatedBackward = false;
}

bool MotorController::hasRotatedForward() const {
    return rotatedForward.load();
}

bool MotorController::hasRotatedBackward() const {
    return rotatedBackward.load();
}

void MotorController::setStep(int w1, int w2, int w3, int w4) {
    gpioWrite(pins[0], w1);
    gpioWrite(pins[1], w2);
    gpioWrite(pins[2], w3);
    gpioWrite(pins[3], w4);
    gpioDelay(1000);


}
```

```
void MotorController::step(int totalSteps, int direction) {
    int stepCount = 8;
    int stepSequence[8][4] = {
        {1,0,0,0},
        {1,1,0,0},
        {0,1,0,0},
        {0,1,1,0},
        {0,0,1,0},
        {0,0,1,1},
        {0,0,0,1},
        {1,0,0,1}
    };
    for (int i = 0; i < totalSteps; ++i) {
        for (int s = 0; s < stepCount; ++s) {
            int idx = (direction > 0) ? s : (stepCount - 1 - s);
            setStep(stepSequence[idx][0],                stepSequence[idx][1],
stepSequence[idx][2], stepSequence[idx][3]);
        }
    }
}

MotorController::~MotorController() {
    stop();
}
```

sonic.h

```
#ifndef ULTRASONICSENSOR_H
#define ULTRASONICSENSOR_H

#include <functional>
#include <mutex>
#include <condition_variable>
#include "pigpio.h"

class UltrasonicSensor {
public:
    UltrasonicSensor(int triggerPin, int echoPin);
    ~UltrasonicSensor();
    void startMeasurement();
    void stop();
    void setDistanceCallback(std::function<void(float)> callback);

private:
    int trigPin, echoPin;
    bool running;
    std::mutex mtx;
    std::condition_variable cv;
```

```
    std::function<void(float)> distanceCallback;
    uint32_t startTime, endTime;
    static void echoInterrupt(int gpio, int level, uint32_t tick, void* user);
    void triggerPulse();
    void waitForEcho();
};

#endif // ULTRASONICSENSOR_H
```

sonic.cpp

```
#include "sonic.h"
#include <iostream>
#include <thread>

UltrasonicSensor::UltrasonicSensor(int triggerPin, int echoPin) :
trigPin(triggerPin), echoPin(echoPin), running(false) {
    gpioSetMode(trigPin, PI_OUTPUT);
    gpioSetMode(echoPin, PI_INPUT);
    gpioSetAlertFuncEx(echoPin, echoInterrupt, this);
}

UltrasonicSensor::~UltrasonicSensor() {
    stop();
    gpioSetAlertFuncEx(echoPin, nullptr, nullptr);
}

void UltrasonicSensor::startMeasurement() {
    running = true;
    std::thread([&]() {
        while (running) {
            triggerPulse();
            waitForEcho();
        }
    }).detach();
}

void UltrasonicSensor::stop() {
    running = false;
    cv.notify_all();
}

void UltrasonicSensor::setDistanceCallback(std::function<void(float)> callback)
{
    distanceCallback = callback;
}

void UltrasonicSensor::triggerPulse() {
    // generate a 10 microsecond high level pulse
```

```
    gpioTrigger(trigPin, 10, 1);
}


void UltrasonicSensor::waitForEcho() {
    std::unique_lock<std::mutex> lock(mtx);
    if (!cv.wait_for(lock, std::chrono::milliseconds(100), [this] { return
this->endTime > this->startTime; })) {
        std::cerr << "Timeout waiting for echo response." << std::endl;
    } else {
        if (distanceCallback) {
            float distance = (endTime - startTime) * 0.0343 / 2.0;
            distanceCallback(distance);
        }
    }
}

void UltrasonicSensor::echoInterrupt(int gpio, int level, uint32_t tick, void*
user) {
    UltrasonicSensor* sensor = static_cast<UltrasonicSensor*>(user);
    if (level == 1) {
        sensor->startTime = tick;
    } else if (level == 0) {
        sensor->endTime = tick;
        sensor->cv.notify_one();
    }
}
```

water.h

```
#ifndef WATER_LEVEL_CONTROLLER_H
#define WATER_LEVEL_CONTROLLER_H

#include <chrono>
#include <pigpio.h>
#include <iostream>

class WaterLevelController {
public:

 static WaterLevelController* instance;

    WaterLevelController(int waterSensorPin, int motorPin);
    ~WaterLevelController();

    void checkWaterLevel(int gpio, int level, uint32_t tick);

private:
    int waterSensorPin;
```

```
    int motorPin;
    bool motorRunning;
    std::chrono::steady_clock::time_point motorStartTime;

    void initialize();
    void startMotor();
    void stopMotor();
};

#endif // WATER_LEVEL_CONTROLLER_H
```

water.cpp

```
#include "water.h"


WaterLevelController* WaterLevelController::instance = nullptr;

WaterLevelController::WaterLevelController(int waterSensorPin, int motorPin)
    : waterSensorPin(waterSensorPin), motorPin(motorPin), motorRunning(false)
{
    initialize();

    instance = this;  // Set instance pointer in the constructo


}

WaterLevelController::~WaterLevelController() {
    gpioTerminate();

     instance = nullptr;  // Clear instance pointer in the destructor

}

void WaterLevelController::checkWaterLevel(int gpio, int level, uint32_t tick)
{
    if (level == PI_HIGH) {
        startMotor();
    }            else            if            (motorRunning            &&
std::chrono::duration_cast<std::chrono::seconds>(
        std::chrono::steady_clock::now() - motorStartTime).count() >= 2) {
        stopMotor();
    }
}

void WaterLevelController::initialize() {
    if (gpioInitialise() < 0) {
        std::cerr << "pigpio initialization failed." << std::endl;
```

```
        exit(1);
    }
    gpioSetMode(waterSensorPin, PI_INPUT);
    gpioSetMode(motorPin, PI_OUTPUT);
    gpioWrite(motorPin, PI_LOW);
    gpioSetAlertFunc(waterSensorPin, [](int gpio, int level, uint32_t tick) {
        WaterLevelController::instance->checkWaterLevel(gpio, level, tick);
    });
}

void WaterLevelController::startMotor() {
    if (!motorRunning) {
        std::cout << "No water, activating PUMP..." << std::endl;
        gpioWrite(motorPin, PI_HIGH);
        motorStartTime = std::chrono::steady_clock::now();
        motorRunning = true;
    }
}

void WaterLevelController::stopMotor() {
    if (motorRunning) {
        gpioWrite(motorPin, PI_LOW);
        std::cout << "PUMP stopped." << std::endl;
        motorRunning = false;
    }
}
```

weight.h

```
#ifndef WEIGHTSENSOR_H
#define WEIGHTSENSOR_H

#include <atomic>
#include <thread>
#include <pigpio.h>
#include <queue>
#include <numeric>  // Include this header file for use std::accumulate


class WeightSensor {
public:
    WeightSensor(int SCK, int SDA, int calibration, float coefficient);
    ~WeightSensor();
    void start();
    void stop();
    int getLatestWeight() const;

    static WeightSensor* instance;
```

```
private:
    int SCK, SDA;
    int calibration;
    float coefficient;
    std::atomic<bool> running;
    std::atomic<int> latestWeight;
    std::thread weightThread;
    //std::queue<int> weightHistory;
  //  const size_t weightHistorySize = 10;
    std::deque<int> weightHistory;
    const size_t weightHistorySize = 10;  // Maintain 10 data points for averaging

    void triggerRead();
    int readWeight();
    int readRawWeight();
    void updateWeightHistory(int newWeight);
    int getSmoothedWeight();
};

#endif // WEIGHTSENSOR_H
```

weight.cpp

```
#include "weight.h"
#include <iostream>
#include <queue>


WeightSensor* WeightSensor::instance = nullptr;

WeightSensor::WeightSensor(int SCK, int SDA, int calibration, float coefficient)
    : SCK(SCK), SDA(SDA), calibration(calibration), coefficient(coefficient),
running(false), latestWeight(0) {
    gpioInitialise();
    gpioSetMode(SCK, PI_OUTPUT);
    gpioSetMode(SDA, PI_INPUT);
    gpioWrite(SCK, PI_LOW);
    instance = this;  // Set the instance pointer in the constructor.
}

WeightSensor::~WeightSensor() {
    stop();
    gpioTerminate();
    instance = nullptr;  // Clear the instance pointer.
}

void WeightSensor::start() {
    running = true;
    gpioSetAlertFunc(SDA, [](int gpio, int level, uint32_t tick) {
```

```
        if (level == 0 && WeightSensor::instance) {  // make sure the instance
is useful
            WeightSensor::instance->triggerRead();
        }
    });
}

void WeightSensor::stop() {
    running = false;
    gpioSetAlertFunc(SDA, nullptr);  // Remove the interrupt handler function
}

int WeightSensor::getLatestWeight() const {
    return latestWeight.load();
}

void WeightSensor::triggerRead() {
    if (!running) return;
   // int rawWeight = readWeight();
    //updateWeightHistory(rawWeight);
    latestWeight.store(readWeight());  // Update the latest weight


int WeightSensor::readWeight() {
    int rawWeight = readRawWeight();
    int  calculatedWeight  =  static_cast<int>(static_cast<float>(rawWeight  -
calibration) / coefficient);


    if (calculatedWeight == 342) {
        return latestWeight.load();  // hardware defect filtering
    }

    return calculatedWeight;
}


int WeightSensor::readRawWeight() {
    long value = 0;
    for (int i = 0; i < 24; ++i) {
        gpioWrite(SCK, PI_HIGH);
        gpioWrite(SCK, PI_LOW);
        if (gpioRead(SDA)) {
            value = (value << 1) + 1;
        } else {
            value <<= 1;
        }
    }
```

```
    gpioWrite(SCK, PI_HIGH); // Additional pulse to end the read
    gpioWrite(SCK, PI_LOW);

    if (value & 0x800000) {
        value |= ~0xFFFFFF;
    }
    return static_cast<int>(value);
}

void WeightSensor::updateWeightHistory(int newWeight) {
    if (weightHistory.size() >= weightHistorySize) {
        weightHistory.pop_front();  // If the deque is full, remove the oldest data
    }
    weightHistory.push_back(newWeight);  // Add a new weight record
}

int WeightSensor::getSmoothedWeight() {
    if (weightHistory.empty()) return 0;
    int sum = std::accumulate(weightHistory.begin(), weightHistory.end(), 0);
    return sum / weightHistory.size();  // Calculate and return the average weight
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(PetCare)

set(CMAKE_CXX_STANDARD 11)
set(LINKSDK_DIR "${CMAKE_SOURCE_DIR}/external/LinkSDK")


execute_process(
    COMMAND make
    WORKING_DIRECTORY ${LINKSDK_DIR}
    RESULT_VARIABLE MAKE_RESULT
)

if(NOT MAKE_RESULT EQUAL "0")
    message(FATAL_ERROR "Building LinkSDK failed with ${MAKE_RESULT}")
endif()

include_directories(${LINKSDK_DIR}/output/include)

find_package(Threads REQUIRED)
find_library(PIGPIO_LIBRARY pigpio /usr/local/lib)
find_library(PIGPIO_THREAD pthread /usr/local/lib)

include_directories(${CMAKE_SOURCE_DIR}/lib)
```

```
set(SOURCE_FILES
    ../src/main.cpp
    ../lib/motor.cpp
    ../lib/motor.h
    ../lib/sonic.cpp
    ../lib/sonic.h
    ../lib/weight.h
    ../lib/weight.cpp
    ../lib/water.h
    ../lib/water.cpp
    ../lib/IOTConnect.h
    ../lib/IOTConnect.cpp

)

add_library(link_sdk STATIC IMPORTED)
set_property(TARGET        link_sdk        PROPERTY        IMPORTED_LOCATION
${LINKSDK_DIR}/output/lib/libaiot.a)

add_executable(PetCare ${SOURCE_FILES})
target_link_libraries(PetCare link_sdk)
target_link_libraries(PetCare       ${PIGPIO_LIBRARY}       ${PIGPIO_THREAD}
${CMAKE_THREAD_LIBS_INIT})
```

main.cpp

```
#include "sonic.h"
#include "motor.h"
#include "weight.h"
#include "water.h"
#include "pigpio.h"
#include "IOTConnect.h"
#include <iostream>
#include <csignal>
#include <atomic>
#include <mutex>
#include <chrono>

std::atomic<bool> terminateProgram{false};
std::mutex mtx;  // Mutexes are used to protect shared variables

std::chrono::time_point<std::chrono::steady_clock> lastReverseTime;
std::chrono::time_point<std::chrono::steady_clock> lastReverseTime2;
const std::chrono::seconds reverseCooldown(30);  // cool for 30s
const std::chrono::seconds reverseCooldown2(2);
void signalHandler(int signum) {
    terminateProgram = true;
}
```

```
int main() {
    if (gpioInitialise() < 0) {
        std::cerr << "GPIO initialization failed。" << std::endl;
        return 1;
    }

    signal(SIGINT, signalHandler);   // gpio initialization
    MotorController motor(17, 18, 27, 22);//set motor GPIO
    UltrasonicSensor sensor(20, 21);//set sonic GPIO
    WaterLevelController waterLevelController(6, 5);//set water GPIO
    WeightSensor weightSensor(23, 24, -151774, 442.54f); //set weight GPIO
    IOTConnect iotconnect;

    int countBelowThreshold = 0;
    const int thresholdCount = 3; //detect three times distance <
thresholdDistance and weight < weightThreshold
    const float thresholdDistance = 10.0;
    const float minValidDistance = 0.1;
    int weightThreshold = iotconnect.weight1;

    weightSensor.start();
    std::thread waterThread([&](){while (!terminateProgram.load())
{}});//water
    int weight1;

    sensor.setDistanceCallback([&](float distance) {
        std::lock_guard<std::mutex> lock(mtx);  // protect shared variables
        auto now = std::chrono::steady_clock::now();
        auto now2 = std::chrono::steady_clock::now();
        weight1 = weightSensor.getLatestWeight();
        if(now2 > lastReverseTime2 + reverseCooldown2 ){
            weightThreshold = iotconnect.weight1;
            std::cout << "Distance: " << distance << " cm, Weight: " << weight1
<< " g, weightthreshold:" << weightThreshold << "g"<<std::endl;
            iotconnect.reportDistance(distance,weight1);


            lastReverseTime2 = std::chrono::steady_clock::now();
            }

        if (distance < minValidDistance || distance > 500) {  // ignore meanless
data
            return;
        }

        int currentWeight = weightSensor.getLatestWeight();
        if (distance < thresholdDistance && currentWeight < weightThreshold &&
```

```
now > lastReverseTime + reverseCooldown) {
            countBelowThreshold++;
            if          (countBelowThreshold          >=          thresholdCount
&& !motor.hasRotatedForward()) {
                std::cout << "Triggering Forward rotation." << std::endl;
                std::cout << "currentWeight"<< currentWeight<< std::endl;
                std::cout << "Weighthold:"<< weightThreshold << std::endl;
                motor.rotateForward(128);  // ++128==motor 90 angle
            }
        } else {
            countBelowThreshold = 0; // reset this: detect three times distance
< thresholdDistance and weight < weightThreshold
        }

        // if weight> weightThreshold and motor has been Forwarded
        if (currentWeight >= weightThreshold && motor.hasRotatedForward()) {
            std::cout << "Triggering Reverse rotation." << std::endl;
            std::cout << "currentWeight"<< currentWeight<< std::endl;
            std::cout << "Weighthold:"<< weightThreshold << std::endl;
            motor.rotateBackward(128);  //  --128==motor 90 angle
            motor.reset();  // reset motor and allowed motor Forward again
            lastReverseTime = std::chrono::steady_clock::now();
        }
    });

    sensor.startMeasurement();

    while (!terminateProgram) {

    }

    waterThread.join();
    weightSensor.stop();
    sensor.stop();
    gpioTerminate();
    std::cout << "PROGRAM STOPED" << std::endl;
    return 0;
}
```