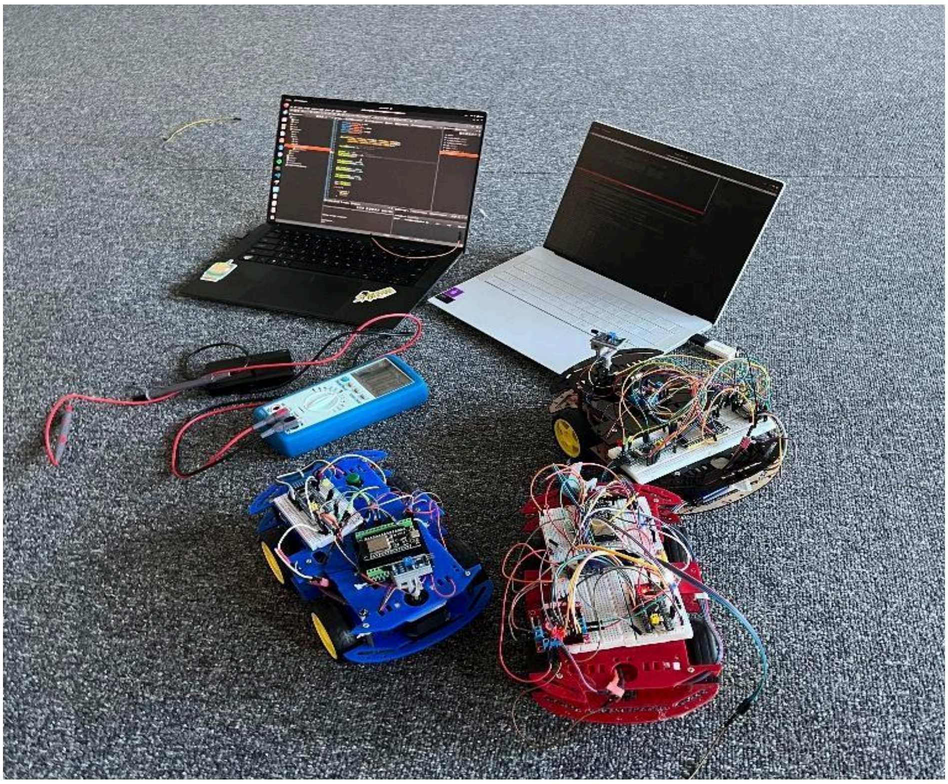
TriFlameX: Cooperative Fire-Fighting Robot Swarm



Course: Mechatronics Engineering (MCTR B501)

Report by:

I.	Mohammed El Said Azab Abdelazim	ID: 13001581
II.	George Halim	ID: 13001582
III.	Mohamed Elsheemy	ID: 13001576

Table of Contents

1. Project Description

TriFlameX is a swarm of three autonomous fire-fighting robots designed to collaboratively detect, locate, and suppress small-scale fires in indoor or controlled outdoor environments. Each robot combines mobility, real-time sensing, wireless communication, and decision-making capabilities. The primary objective is to demonstrate multi-agent coordination and distributed intelligence in emergency response applications. Communication protocols employed include UART for intra-car data transfer and MQTT over Wi-Fi for inter-device communication. This report explains the data flow, communication protocols, packet structures, and overall system logic that govern the TriFlameX system.

2. Methodology

The project aims to develop a cooperative network of autonomous cars equipped with a homemade LiDAR system, enabling them to detect objects (simulating fire sources) and coordinate to dispatch the nearest car for response. The methodology comprises hardware setup, LiDAR scanning, data communication, and network-based decision-making.

3. System Overview

Three autonomous robotic cars were designed to operate collaboratively as part of a network. Each car was equipped with a custom-built LiDAR system using a Time-of-Flight (ToF) distance sensor mounted on a stepper motor for 2D environment scanning. The goal is for the cars to scan their surroundings, detect the nearest target, and identify which car is closest to it. The closest car then moves toward the object, simulating an emergency response to a fire.

Total parts for robots are :

Part Number	Name	Quantity
1	**DC Motors**	12
2	**H-bridge Motor Driver**	3
3	**3s battery**	3
4	**Esp32 Module**	3

5	Stm32 Microcontroller	3

3.1. LiDAR Design and Scanning

Each car performs environmental scanning using a homemade LiDAR system:

- **Mini ToF Sensor was used for accurate distance measurements.**
- **A stepper motor was used to rotate the sensor, allowing for semi-circular scans depending on design constraints.**
- **The scan data (angle and distance) was recorded and used to identify objects in the environment.**

3.2. Communication and Networking

- **The cars communicated via a wireless protocol (Wi-Fi, and UART).**
- **After completing their scans, each car sent its position and detected object data to the server.**
- **All cars participated in a distributed network, allowing them to share scan results and compute the relative distance between each car and the detected object.**

3.4. Target Detection and Decision Making

- **Once a target (fire) was identified based on LiDAR data, each car calculated its own distance to the object (based on LiDAR).**
- **Through communication with the network broker, each car compared its distance to the others.**
- **A simple rule-based protocol (e.g., the car with the lowest distance value acts) was used to decide which car should respond.**
- **Only the car closest to the object proceeded toward it, while the others remained in standby.**

3.5. Navigation and Response

- **The selected car used basic PD control to approach the target.**
- **The movement was based on differential drive control using directional data from the LiDAR scan.**

- **Upon reaching the object, the car would stop and optionally signal that the target had been reached via LED and message broadcast.**

3.6 Simulation of Fire Emergency

- **The object to be found by the cars was treated as a simulated "fire source."**
- **The quick identification and dispatch of the closest car demonstrated the potential use of such a system in fire-fighting, search-and-rescue, or smart surveillance applications.**

4. Mechanical design

The mechanical design focused on building a compact, stable, and modular platform for each autonomous car, capable of supporting sensors, motors, and electronics. Special attention was given to the mounting of the homemade LiDAR system to ensure accurate scanning and minimal vibration.

4.1 Chassis Design

Each car was built on a lightweight but durable chassis, designed to support:

- **Four DC gear motors treated as two with wheels for differential drive mobility.**
- **The chassis was made from PLA for ease of customization and sufficient structural strength.**

4.2 Sensor Mounting

- **The LiDAR system, consisting of a Time-of-Flight (ToF) sensor mounted on a stepper motor, was placed on the front-top of the car.**
- **A custom bracket was designed to hold the sensor firmly while allowing free rotation through the stepper motor shaft.**

4.3 Electronics and Wiring Layout

- **The microcontroller (STM32 and ESP32), motor drivers, and power supply (LiPo battery) were mounted on a multi-level deck to save space and improve heat dissipation.**

- **Careful wire management was implemented using cable ties and heat-shrink tubing to prevent entanglement with moving parts.**

- **All connectors were secured with hot glue or to withstand vibration during motion.**

4.4 Modularity and Maintenance

- **The design ensured that key elements (e.g., battery, microcontroller, LiDAR) could be accessed without disassembling the entire car.**

4.5 Design Considerations

- **Center of gravity was kept low to improve stability during movement and rotation.**

- **The size and weight of the LiDAR were kept minimal to avoid affecting the steering behavior of the car.**

- **Overall car dimensions were optimized to allow tight turns and operation in indoor environments.**

5. Electrical design

The electrical system includes:

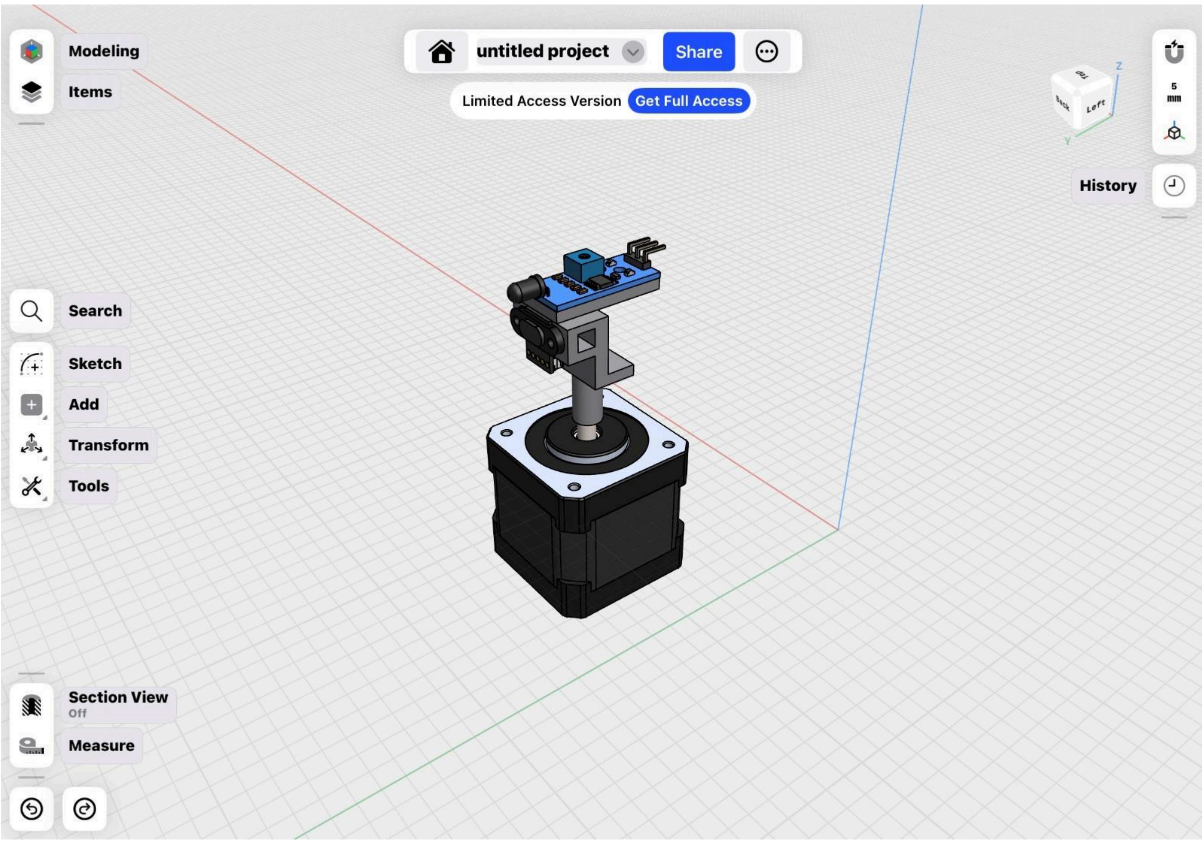
- STM32 microcontroller for real-time motor and sensor control
- ESP32 for wireless communication
- Power regulation via buck converters for 3.3V and 5V lines
- Batteries
- Circuit protection with fuses and diodes
- Nema 17 Stepper motor
- TOF “time of flight sensor”

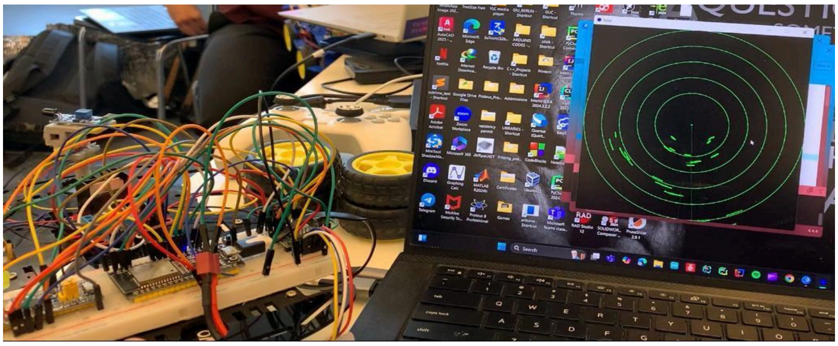
5.2 LIDAR

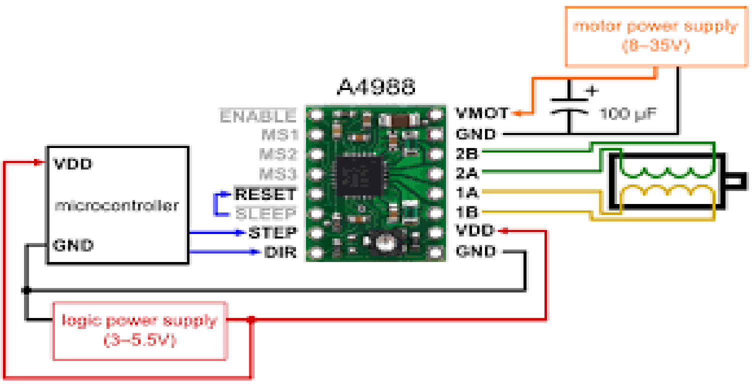
A DIY LiDAR sensor can be built using a Time-of-Flight (ToF) sensor, from waveshare, combined with a NEMA 17 stepper motor to enable 150-degree scanning.

The ToF sensor provides accurate distance measurements by emitting laser pulses and calculating the time it takes for them to reflect back. Mounted on a rotating platform driven by the NEMA 17 motor, the sensor collects distance data at different angles as it spins.

Using a microcontroller , ESP32, it is controlled by the motor and synchronize distance readings with the angular position to generate a 2D point cloud of the surrounding environment.


Modeling
Items
untitled project
Share
Limited Access Version
Get Full Access
History
Search
Sketch
Add
Transform
Tools
Section View
Off
Measure




motor power supply
(8–35V)
A4988
ENABLE
MS1
MS2
MS3
RESET
SLEEP
STEP
DIR
VMOT
GND
2B
2A
1A
1B
VDD
GND
100 µF
VDD
microcontroller
GND
logic power supply
(3–5.5V)

5.3 LIDAR VISUALIZER

A LiDAR visualizer was developed using the Processing programming language to graphically display real-time data from a DIY LiDAR system.

The visualizer receives distance and angle measurements via serial communication from an ESP microcontroller, which controls a ToF sensor mounted on a rotating platform.

As the ESP transmits data through USB to the computer, Processing reads the serial input, parses the values, and plots them as points in a polar coordinate system. This creates a dynamic 2D visualization of the environment as the sensor rotates, allowing users to observe the LiDAR system in action. The visualizer helps in debugging, tuning, and understanding the performance of the scanning mechanism, making it an essential tool for both development and demonstration purposes.

6. Control

The control system of the autonomous car is responsible for navigating towards the detected object using a PD (Proportional-Derivative) control strategy. Two separate PD controllers were implemented: one for distance control and another for angle control. These controllers ensure that the car approaches the target efficiently and accurately while maintaining correct orientation.

6.1 Distance Control (Linear PD Controller)

The distance controller regulates how fast the car moves toward the object based on the measured distance from the LiDAR scan. The error is defined as the difference between the actual distance and a predefined target distance (e.g., 30 cm). The PD controller calculates a forward or backward speed to minimize this error.

Control Law:
Speed = Kp × (error) + Kd × (error − previous error)

Where:

- **Kp is the distance proportional gain**
- **Kd is the distance derivative gain**

The output speed is clamped within a certain range (e.g., -100 to 100) to prevent excessive movement. When the error is small (i.e., the car is close enough to the target), the speed is set to zero to stop the car smoothly.

Where:

- **KpK_pKp is the proportional gain (initially 0.64)**
- **KdK_dKd is the derivative gain (initially 6.0)**
- **Error is the distance to the target minus the desired minimum distance**

Speed is clamped within [-100, 100] to prevent excessive motor commands. If the error is within a small threshold (e.g., 6 cm), the speed is set to 0 to stop the car near the object.

6.2 Angle Control (Rotational PD Controller)

The angle controller ensures the car faces the object by adjusting the relative speeds of the left and right motors. It uses the difference between the desired angle (90°, i.e., straight ahead) and the actual measured angle.

Control Law:
Steering = Kp_angle × (angle error) + Kd_angle × (angle error - previous angle error)

Where:

- **Kp_angle is the angle proportional gain (e.g., 0.9)**
- **Kd_angle is the angle derivative gain (e.g., 0.0) to minimize calculation error magnification**

The steering correction is added and subtracted from the base speed to determine the individual motor speeds, allowing the car to turn smoothly toward the target. To prevent oversteering, the steering correction is clamped between -60 and +60.

6.3 Motion Logic

Depending on the control output:

- **If both speeds are positive → forward movement**
- **If both are negative → reverse**
- **If speeds have opposite signs → pivot turn (left or right)**
- **If error or angle is within thresholds → car stops**

The CAR_forward, CAR_backwards, CAR_left, and CAR_right functions handle the low-level motor control based on these decisions.

6.4 Dynamic Gain Adjustment

The controller dynamically adjusts gains based on distance. If the car is very close to the object, higher gains are applied to react more quickly and reduce overshoot.

6.5 Safety & Clamping

To avoid unstable or unsafe behavior:

- **Motor speeds are clamped to ±100**
- **A deadband zone near the target prevents jittering**
- **Steering corrections are limited to ±60**

6.6 Programming

STM32: C-based code to read sensor data, send to and receive from ESP32 via UART, drive motors via PWM

ESP32: C++ for communication and decision logic and sensor reading and stepper movement

Laptop: Ubuntu-OS uses Python scripts and Ros2 Framework for decision making and data transmission

You can access the full source and supporting materials at:
https://github.com/RoboMechanix/TriFlameX.git

7. Communication System – Multi-Node Architecture

This project implements a distributed control and sensing system using ESP32 and STM32 microcontrollers, with a central laptop orchestrating the control over MQTT (Wi-Fi) and UART (serial) communication.
The system enables real-time control and monitoring of multiple robotic nodes, using a custom binary protocol between ESP32 and STM32, and a ROS-based MQTT bridge between the Laptop and ESP32.

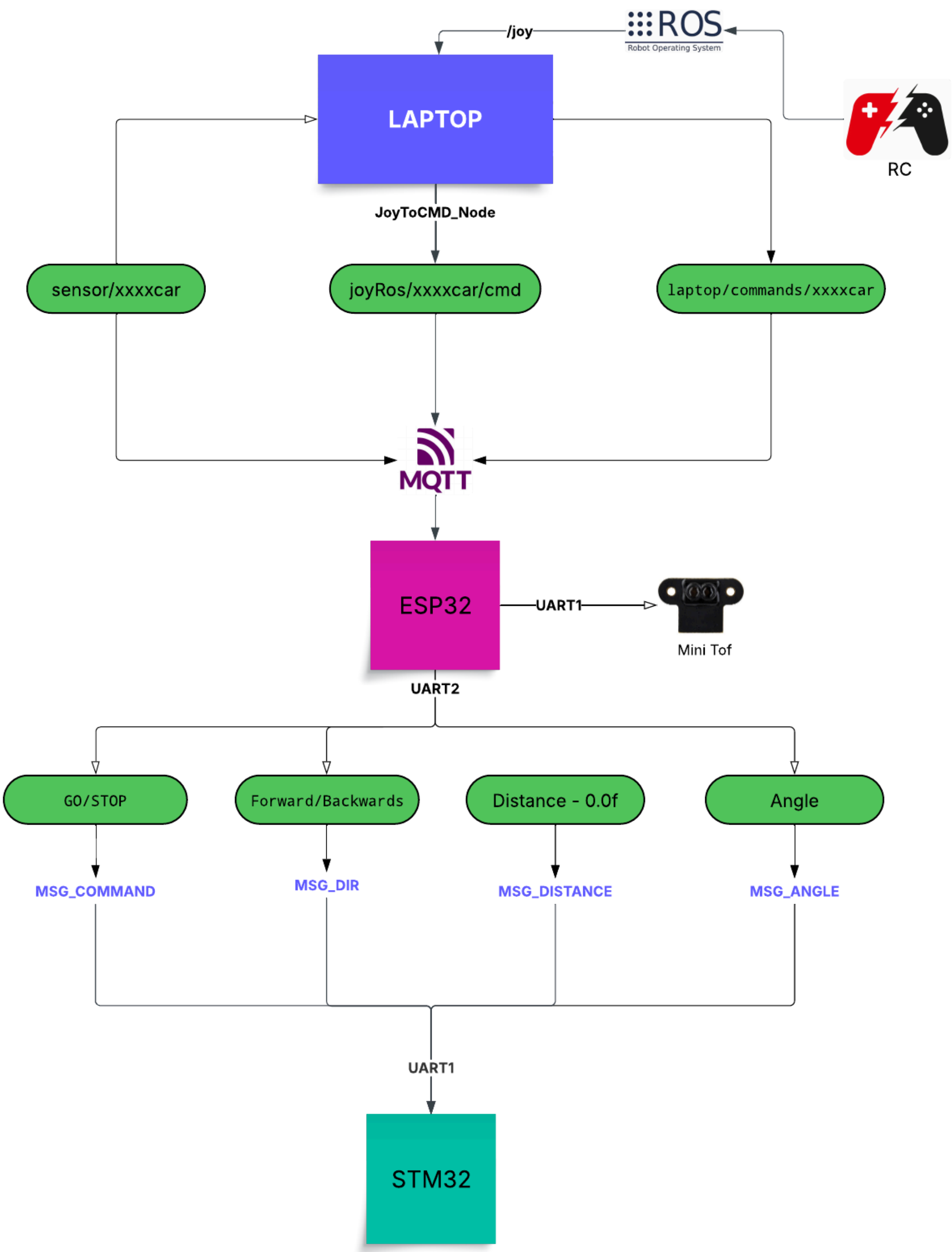
7.1 System Architecture

The TriFlameX system architecture consists of three primary layers: the central laptop controller, ESP32 modules on each car, and STM32 microcontrollers connected to the car's actuators. Communication flows from sensor acquisition to actuation, coordinated through wireless and wired protocols.

7.2 Components and Communication Links

- **Laptop (Central Brain):**
    - Connected via Wi-Fi to ESP32 modules.
    - Runs control logic in Python.
    - Subscribes and publishes MQTT topics for sensor data and commands.
- **ESP32 Modules (One per Car):**
    - Reads Time-of-Flight (ToF) sensor data via UART1.
    - Publishes sensor data to laptop via MQTT.
    - Receives commands from laptop via MQTT.
    - Forwards commands to STM32 over UART2 using a custom binary protocol.
    - Subscribes to joystick commands in manual mode and forwards to STM32.
- **STM32 Microcontrollers:**
    - Connected to ESP32 via UART2.
    - Receives commands, verifies packet integrity.
    - Sends acknowledgments (ACK).
    - Controls motors and servos based on commands.

7.3 Communication Summary Diagram



This figure Show data flows and control flows.

7.4 Communication Protocols

7.4.1 MQTT over Wi-Fi

MQTT provides the backbone wireless communication channel between the laptop and ESP32 modules.

- **Sensor Data Topics:** sensor/xxxxcar
- **Command Topics:** laptop/commands/xxxxcar
- **Manual Joystick Commands:** joyROS/xxxxcar/cmd

The laptop subscribes to sensor data from all cars, processes commands in Python, and publishes control commands accordingly.

7.4.2 UART Communication

Two UART channels per ESP32:

- **UART1:** ToF sensor to ESP32 (sensor data acquisition)
- **UART2:** ESP32 to STM32 (command transmission)

7.4.3 UART Packet Format and Bit Protocol

Communication between ESP32 and STM32 uses a compact 6-byte packet with framing and checksum for reliability.

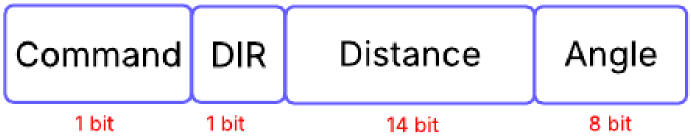
Packet Structure

Byte Index	Field	Description
0	Start Byte	0xAA — start of message
1–3	Payload	3 bytes (24 bits) of packed command data
4	Checksum	XOR of payload bytes (bytes 1–3)
5	End Byte	0x55 — end of message

7.4.4 Checksum Calculation

Checksum is calculated by XOR operation on the three payload bytes to ensure data integrity.

8.3 Payload Bit Layout (24 bits)



Bits	Field	Description
1	Command	1-bit command flag (e.g., go/stop)
1	Direction	Direction bit (0=forward, 1=backward)
14	Distance	Distance from ToF sensor (0–16383)
1	Sign	Sign of angle (0=positive, 1=negative)
7	Angle	Angle magnitude (0–127)

7.4.5 Data Flow and Logic

1. **Sensor Acquisition:**
    - ESP32 reads ToF sensor data via UART1.
    - Packs command, direction, distance, sign, and angle bits into a 3-byte payload.
    - Forms a 6-byte UART packet with framing and checksum.
    - Sends packet to STM32 over UART2.
2. **STM32 Processing:**
    - Validates start/end bytes and checksum.

- Sends ACK (0xCC) if the packet is valid.
- Decodes payload to execute motor and servo commands.

3. **MQTT Communication:**
    - ESP32 publishes ToF data on sensor/xxxxcar.
    - Laptop receives data from all cars and processes commands:
        - Car closest to the target receives “go” command.
        - Others receive “stop” commands.
    - Laptop publishes commands on laptop/commands/xxxxcar.
    - ESP32 forwards these commands to STM32.
4. **Disconnection Handling:**
    - If a car disconnects, laptop assigns max distance value.
    - Corresponding car commanded to stop.
    - Control logic reevaluates remaining cars.
5. **Manual Control Mode:**
    - Joystick node publishes control commands on /joy.
    - Custom node joy_toCMD subscribes to /joy.
    - On command reception, publishes to joyROS/xxxxcar/cmd.
    - ESP32 forwards joystick commands over UART2 to STM32.
    - Manual control uses same binary protocol for command packets.

---

10. Conclusion

This repository contains the complete project hierarchy, including hardware schematics, firmware for the ESP microcontroller, Processing code for real-time visualization, and detailed documentation for setting up and running the DIY LiDAR system. All files are organized for clarity and ease of use, making it simple to replicate or build upon the project.

You can access the full source and supporting materials at:
https://github.com/RoboMechanix/TriFlameX.git

11. References

- https://www.waveshare.com/wiki/TOF_Laser_Range_Sensor_Mini#UART

12.Code

Main.c

```
#include "main.h"

#include "FreeRTOS.h"

#include "task.h"


// === Globals ===

uint64_t current_time_ms = 1;


// === Variables ===


float realangle = 0.0f;


char c;

char uart_rx_buffer[UART_BUFFER_SIZE];

int uart_rx_index = 0;

int x = 0;


volatile uint8_t uart_line_ready = 0;

char uart_rx_buffer_copy[UART_BUFFER_SIZE];  // Used by main loop/task
```

```
void PD_Distance_Task(void *pvParameters);
void PD_Angle_Task(void *pvParameters);

// Dummy delay (for simulation only)
void delay_ms(uint32_t ms) {
    for (volatile uint32_t i = 0; i < ms * 1000; i++) {
        __asm("NOP");
    }
}

int main(void) {
    // Initialize UART
    UART_init(1, BAUDRATE);
    UART1_InterruptsInit();

    // === Left Motor (TIM3, PA4/PA5) ===
    TIM_TypeDef *leftTimer = TIM3;
    uint8_t leftChannel = 2;
    GPIO_TypeDef *leftDir1Port = GPIOA;
    uint8_t leftDir1Pin = 4;
    GPIO_TypeDef *leftDir2Port = GPIOA;
    uint8_t leftDir2Pin = 5;

    // === Right Motor (TIM4, PB6/PB7) ===
    TIM_TypeDef *rightTimer = TIM4;
    uint8_t rightChannel = 2;
    GPIO_TypeDef *rightDir1Port = GPIOA;
    uint8_t rightDir1Pin = 2;
    GPIO_TypeDef *rightDir2Port = GPIOA;
```

```
    uint8_t rightDir2Pin = 3;

    // green led

    GPIO_pinMode(GPIOB, 8, OUTPUT);

    GPIO_digitalWrite(GPIOB, 8, LOW);


    // yellow led

    GPIO_pinMode(GPIOB, 6, OUTPUT);

    GPIO_digitalWrite(GPIOB, 6, LOW);


    // red led

    GPIO_pinMode(GPIOB, 9, OUTPUT);

    GPIO_digitalWrite(GPIOB, 9, LOW);


    // === Init Car Motors ===

    CAR_init(leftTimer, leftChannel, PWM_FREQ_HZ, leftDir1Port,
leftDir2Port,

            leftDir1Pin, leftDir2Pin, rightTimer, rightChannel,
PWM_FREQ_HZ,

            rightDir1Port, rightDir2Port, rightDir1Pin, rightDir2Pin);


    // === Init Millisecond Timer (TIM2 used for timing) ===

    TIM_initMillis(TIM2, 1);  // 1ms resolution

    delay_ms(50);


    // === Initialize PD controllers ===

    PD_init(1.4f, 6.0f);        // Distance PD

    PD_init_angle(0.90f, 0.0f); // Angle control gains


    // === Create FreeRTOS Tasks ===

    xTaskCreate(PD_Distance_Task, "DistanceTask", 256, NULL, 2, NULL);
```

```
    xTaskCreate(PD_Angle_Task, "AngleTask", 256, NULL, 2, NULL);

    // === Start Scheduler ===
    vTaskStartScheduler();

    // Should never reach here
    while (1)
        ;
}

// === Task for PD Distance ===
void PD_Distance_Task(void *pvParameters) {
    while (1) {
        current_time_ms = TIM_Millis();
        if (distance != 0.0f) {
            if (distance > maxDistance)
                PD_init(1.4f, 6.0f);        // Distance PD

            PD_update_from_distance(distance, current_time_ms);
        }
        vTaskDelay(pdMS_TO_TICKS(1)); // Update every 10ms
    }
}

// === Task for PD Angle ===
void PD_Angle_Task(void *pvParameters) {
    while (1) {
        current_time_ms = TIM_Millis();
```

```
        if (!command) {
            CAR_stop();
        } else
            PD_update_angle(angle, current_time_ms);


        vTaskDelay(pdMS_TO_TICKS(1


        )); // Update every 10ms
    }
}
```

Tim_program.c

```
#include "TIM_interface.h"


volatile uint16_t *preload;
volatile uint16_t preload2;
volatile uint16_t preload3;
volatile uint16_t preload4;
volatile uint16_t *n;
volatile uint16_t n2;
volatile uint16_t n3;
volatile uint16_t n4;
volatile uint16_t *counter;
volatile uint16_t counter2;
```

```
volatile uint16_t counter3;

volatile uint16_t counter4;

volatile void (*callback2)();

volatile void (*callback3)();

volatile void (*callback4)();

uint64_t millis = 0;

uint16_t trigTime_ms_global;

int isFirstTime = 1;


void TIM_initPWM(TIM_TypeDef *TIMX, uint8_t channel, float frequency) {

    if (channel < 1 || channel > 4) {

        return;

    }

    // init clock and corresponding pin in the GPIO

    if (TIMX == TIM2) {

        SET_BIT(RCC->APB1ENR, 0); // Enable TIM2 clock

        RCC->APB2ENR |= RCC_APB2ENR_IOPAEN; // Enable GPIOA clock

        switch (channel) {

        case 1: // PA0

            GPIOA->CRL &= ~(GPIO_CRL_MODE0 | GPIO_CRL_CNF0);

            GPIOA->CRL |= (GPIO_CRL_MODE0_1 | GPIO_CRL_CNF0_1); // 2 MHz, AF
PP

            break;

        case 2: // PA1

            GPIOA->CRL &= ~(GPIO_CRL_MODE1 | GPIO_CRL_CNF1);

            GPIOA->CRL |= (GPIO_CRL_MODE1_1 | GPIO_CRL_CNF1_1);

            break;

        case 3: // PA2

            GPIOA->CRL &= ~(GPIO_CRL_MODE2 | GPIO_CRL_CNF2);
```

```
            GPIOA->CRL |= (GPIO_CRL_MODE2_1 | GPIO_CRL_CNF2_1);
            break;
        case 4: // PA3
            GPIOA->CRL &= ~(GPIO_CRL_MODE3 | GPIO_CRL_CNF3);
            GPIOA->CRL |= (GPIO_CRL_MODE3_1 | GPIO_CRL_CNF3_1);
            break;
        }
    } else if (TIMX == TIM3) {
        SET_BIT(RCC->APB1ENR, 1); // Enable TIM3 clock
        RCC->APB2ENR |= RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN; // Enable
GPIOA & GPIOB
        switch (channel) {
        case 1: // PA6
            GPIOA->CRL &= ~(GPIO_CRL_MODE6 | GPIO_CRL_CNF6);
            GPIOA->CRL |= (GPIO_CRL_MODE6_1 | GPIO_CRL_CNF6_1);
            break;
        case 2: // PA7
            GPIOA->CRL &= ~(GPIO_CRL_MODE7 | GPIO_CRL_CNF7);
            GPIOA->CRL |= (GPIO_CRL_MODE7_1 | GPIO_CRL_CNF7_1);
            break;
        case 3: // PB0
            GPIOB->CRL &= ~(GPIO_CRL_MODE0 | GPIO_CRL_CNF0);
            GPIOB->CRL |= (GPIO_CRL_MODE0_1 | GPIO_CRL_CNF0_1);
            break;
        case 4: // PB1
            GPIOB->CRL &= ~(GPIO_CRL_MODE1 | GPIO_CRL_CNF1);
            GPIOB->CRL |= (GPIO_CRL_MODE1_1 | GPIO_CRL_CNF1_1);
            break;
        }
```

```c
} else if (TIMX == TIM4) {
    SET_BIT(RCC->APB1ENR, 2); // Enable TIM4 clock
    RCC->APB2ENR |= RCC_APB2ENR_IOPBEN; // Enable GPIOB clock
    switch (channel) {
    case 1: // PB6
        GPIOB->CRL &= ~(GPIO_CRL_MODE6 | GPIO_CRL_CNF6);
        GPIOB->CRL |= (GPIO_CRL_MODE6_1 | GPIO_CRL_CNF6_1);
        break;
    case 2: // PB7
        GPIOB->CRL &= ~(GPIO_CRL_MODE7 | GPIO_CRL_CNF7);
        GPIOB->CRL |= (GPIO_CRL_MODE7_1 | GPIO_CRL_CNF7_1);
        break;
    case 3: // PB8
        GPIOB->CRH &= ~(GPIO_CRH_MODE8 | GPIO_CRH_CNF8);
        GPIOB->CRH |= (GPIO_CRH_MODE8_1 | GPIO_CRH_CNF8_1);
        break;
    case 4: // PB9
        GPIOB->CRH &= ~(GPIO_CRH_MODE9 | GPIO_CRH_CNF9);
        GPIOB->CRH |= (GPIO_CRH_MODE9_1 | GPIO_CRH_CNF9_1);
        break;
    }
}

// direction upward
CLEAR_BIT(TIMX->CR1, 4);
// mode 'edge aligned'
CLEAR_BIT(TIMX->CR1, 5);
CLEAR_BIT(TIMX->CR1, 6);
```

```
// set the ARR preload
SET_BIT(TIMX->CR1, 7);
//enable the capture compare corresponding pin
SET_BIT(TIMX->CCER, (4 * (channel - 1)));
// choose the polarity of the pin to active high
CLEAR_BIT(TIMX->CCER, (4 * (channel - 1) + 1));

volatile uint32_t *CCMRX;
uint8_t modChannel = 1;
if (channel <= 2) {
    CCMRX = &TIMX->CCMR1;
    modChannel = channel;
} else {
    CCMRX = &TIMX->CCMR2;
    modChannel = channel - 2;
}
// set the channel mode to be output
CLEAR_BIT(*CCMRX, (8 * (modChannel - 1)));
CLEAR_BIT(*CCMRX, (8 * (modChannel - 1) + 1));
// set the channel preload enable
SET_BIT(*CCMRX, (8 * (modChannel - 1) + 3));
// select PWM mode 1
CLEAR_BIT(*CCMRX, (8 * (modChannel - 1) + 4));
SET_BIT(*CCMRX, (8 * (modChannel - 1) + 5));
SET_BIT(*CCMRX, (8 * (modChannel - 1) + 6));

// setting the psc with zero
TIMX->PSC = 0;
```

```
    // calculating prescaler and arr for specific frequency
    float currentARR = (8000000 / ((frequency * (TIMX->PSC + 1)))) - 1;
    while (currentARR >= 65536) {
        TIMX->PSC += 1;
        currentARR = (8000000 / ((frequency * (TIMX->PSC + 1)))) - 1;
    }
    TIMX->ARR = currentARR;
    SET_BIT(TIMX->EGR, 0);  // UG: Update Generation
    // start counting
    SET_BIT(TIMX->CR1, 0);
}

void TIM_writePWM(TIM_TypeDef *TIMX, uint8_t channel, float dutyCycle) {
    if (channel < 1 || channel > 4 || dutyCycle < 0 || dutyCycle > 100) {
        return;
    }
    volatile uint32_t *CCRX;
    if (channel == 1) {
        CCRX = &TIMX->CCR1;
    } else if (channel == 2) {
        CCRX = &TIMX->CCR2;
    } else if (channel == 3) {
        CCRX = &TIMX->CCR3;
    } else if (channel == 4) {
        CCRX = &TIMX->CCR4;
    }
    *CCRX = (dutyCycle / 100) * (TIMX->ARR);
}
```

```
void TIM_initDelay(TIM_TypeDef *TIMX, uint16_t minTime_ms) {
    enableTimerClock(TIMX);
    uint32_t clk_freq = 8000000; // 8 MHz
    uint32_t target_ticks = minTime_ms * 1000; // Convert minTime_ms to
microseconds

    uint16_t prescaler = 0;
    uint32_t arr = 0;

    // Try to find the smallest prescaler that gives ARR <= 65535
    for (prescaler = 1; prescaler <= 0xFFFF; prescaler++) {
        arr = (clk_freq / prescaler) * minTime_ms / 1000;
        if (arr <= 0xFFFF)
            break;
    }

    if (prescaler > 0xFFFF) {
        return;
    }

    TIMX->CR1 = 0;
    TIMX->PSC = prescaler - 1;
    TIMX->ARR = arr - 1;
    TIMX->EGR = TIM_EGR_UG;
    TIMX->CNT = 0;
    TIMX->SR &= ~TIM_SR_UIF;
    TIMX->CR1 |= TIM_CR1_CEN;
```

```
}

void TIM_delay(TIM_TypeDef *TIMX, uint32_t delay_ms) {
    enableTimerClock(TIMX);
    TIMX->CR1 = 0;
    TIMX->CNT = 0;
    TIMX->PSC = 8000 - 1;
    TIMX->ARR = delay_ms - 1;
    TIMX->CR1 |= TIM_CR1_CEN;
    while (!(TIMX->SR & TIM_SR_UIF))
        ;
    TIMX->SR &= ~TIM_SR_UIF;
    TIMX->CR1 &= ~TIM_CR1_CEN;    // Stop timer
}

void TIM_delay_long(TIM_TypeDef *TIMX, uint32_t delay_ms) { // use it when
delay > 65 Seconds
    while (delay_ms > 0) {
        uint32_t chunk = (delay_ms > 65000) ? 65000 : delay_ms;
        TIM_delay(TIMX, chunk);
        delay_ms -= chunk;
    }
}

void enableTimerClock(TIM_TypeDef *TIMx) {
    switch ((uint32_t) TIMx) {
    case (uint32_t) TIM1:
        RCC->APB2ENR |= RCC_APB2ENR_TIM1EN;
        break;
```

```
    case (uint32_t) TIM2:
        RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
        break;
    case (uint32_t) TIM3:
        RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
        break;
    case (uint32_t) TIM4:
        RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;
        break;
    default:
        break;
    }
}

void TIM_callback(TIM_TypeDef *TIMX, float minTimeMs, float timeMs,
        void (*application)()) { // we can change the callback function to be
anything
    // till now the mintime doesn't do anything
    // init clock and the corresponding timer's interrupt
    if (TIMX == TIM2) {
        SET_BIT(RCC->APB1ENR, 0); // Enable TIM2 clock
        SET_BIT(TIMX->DIER, 0); // Enable update interrupt (UIE)
        // enable interrupt
        NVIC_EnableIRQ(TIM2_IRQn);
        // set interrupt priority
        NVIC_SetPriority(TIM2_IRQn, 1);
        counter = &counter2;
        preload = &preload2;
        n = &n2;
```

```
        callback2 = application;
    } else if (TIMX == TIM3) {
        SET_BIT(RCC->APB1ENR, 1); // Enable TIM3 clock
        SET_BIT(TIMX->DIER, 0); // Enable update interrupt (UIE)
        // enable interrupt
        NVIC_EnableIRQ(TIM3_IRQn);
        // set interrupt priority
        NVIC_SetPriority(TIM3_IRQn, 1);
        counter = &counter3;
        preload = &preload3;
        n = &n3;
        callback3 = application;
    } else if (TIMX == TIM4) {
        SET_BIT(RCC->APB1ENR, 2); // Enable TIM4 clock
        SET_BIT(TIMX->DIER, 0); // Enable update interrupt (UIE)
        // enable NVIC interrupt
        NVIC_EnableIRQ(TIM4_IRQn);
        // set interrupt priority
        NVIC_SetPriority(TIM4_IRQn, 1);
        counter = &counter4;
        preload = &preload4;
        n = &n4;
        callback4 = application;
    }
    // direction upward
    CLEAR_BIT(TIMX->CR1, 4);
    // mode 'edge aligned'
    CLEAR_BIT(TIMX->CR1, 5);
```

```c
    CLEAR_BIT(TIMX->CR1, 6);


//  uint32_t prescaler = ((minTimeMs*4000.0) - 1);

//  uint32_t arr = 1; // the arr should be >= 1

//  if (prescaler > 65535){

//      prescaler = 65535;

//      arr = ((minTimeMs*8000)/(prescaler + 1)) - 1;

//      if (arr > 65535){ // we won't reach this case unless the minimum time
was more than 8 minutes

//          arr = 65535;

//      }

//  }


    uint32_t prescaler = 255;

    uint32_t arr = 100;


    TIMX->ARR = (uint16_t) arr;

    TIMX->PSC = prescaler;

    uint32_t arr_new = ((timeMs * 8000) / (TIMX->PSC + 1)) - 1;

    if (arr_new <= TIMX->ARR) {

        TIMX->ARR = (uint16_t) arr_new;

        *n = 1;

    } else {

        float div = ((float) (arr_new + 1)) / ((float) (TIMX->ARR + 1));

        *n = (uint16_t) div;

        *preload = (uint16_t) (((1 - (div - (float) (*n))))

                * ((float) (TIMX->ARR + 1))); // watch out we need old n

        *n = *n + 1; // due to the preload the interrupt number should
increase by 1
```

```
        TIMX->CNT = *preload;
    }

    // start counting
    SET_BIT(TIMX->CR1, 0);
}
void TIM_initMillis(TIM_TypeDef *TIMX, uint16_t trigTime_ms) {
    enableTimerClock(TIMX);
    TIMX->CR1 = 0;
    trigTime_ms_global = trigTime_ms;
    TIMX->CCER |= TIM_CCER_CC1E;
    TIMX->DIER |= TIM_DIER_CC1IE;
    TIMX->CCMR1 &= ~TIM_CCMR1_CC1S;
    TIMX->CNT = 0;
    TIMX->PSC = 8000 - 1;
    TIMX->ARR = (60000) - 1;
    if (TIMX == TIM2) {
        NVIC_EnableIRQ(TIM2_IRQn);
    } else if (TIMX == TIM3) {
        NVIC_EnableIRQ(TIM3_IRQn);
    } else if (TIMX == TIM4) {
        NVIC_EnableIRQ(TIM4_IRQn);
    }

    TIMX->CR1 |= TIM_CR1_CEN;
    TIMX->CCR1 = TIMX->CNT + trigTime_ms_global;

}
```

```c
uint64_t TIM_Millis() {
    if (isFirstTime) {
        isFirstTime = 0;
        millis = 0;
    }
    return millis;
}

void TIM2_IRQHandler() {
    if (TIM2->SR & TIM_SR_CC1IF) {
        TIM2->SR &= ~TIM_SR_CC1IF;
        TIM2->CCR1 = TIM2->CNT + trigTime_ms_global;
        millis++;
//      if (isFirstTime) {
//          isFirstTime = 0;
//          millis = 0;
//      }
    }
    if ((TIM2->SR & (1 << 0)) == 1) { // check the uif flag
        TIM2->SR &= ~(1 << 0); // clear the uif
        counter2++;
        if (counter2 == n2) {
            counter2 = 0;
            TIM2->CNT = preload2; // set the preload
            callback2();
        }
    }
```

```
}

void TIM3_IRQHandler() {
    if ((TIM3->SR & (1 << 0)) == 1) { // check the uif flag
        if (TIM3->SR & TIM_SR_CC1IF) {
            TIM3->SR &= ~TIM_SR_CC1IF;
            TIM3->CCR1 = TIM3->CNT + trigTime_ms_global;
            millis++;
//          if (isFirstTime) {
//              isFirstTime = 0;
//              millis = 0;
//          }
        }
        if ((TIM3->SR & (1 << 0)) == 1) {
            TIM3->SR &= ~(1 << 0); // clear the uif
            counter3++;
            if (counter3 == n3) {
                counter3 = 0;
                TIM3->CNT = preload3; // set the preload
                callback3();
            }
        }
    }
}

void TIM4_IRQHandler() {
    if ((TIM4->SR & (1 << 0)) == 1) { // check the uif flag
        if (TIM4->SR & TIM_SR_CC1IF) {
```

```
            TIM4->SR &= ~TIM_SR_CC1IF;

            TIM4->CCR1 = TIM4->CNT + trigTime_ms_global;

            millis++;

//          if (isFirstTime) {

//              isFirstTime = 0;

//              millis = 0;

//          }

        }

        if ((TIM4->SR & (1 << 0)) == 1) {

            TIM4->SR &= ~(1 << 0); // clear the uif

            counter4++;

            if (counter4 == n4) {

                counter4 = 0;

                TIM4->CNT = preload4; // set the preload

                callback4();

            }

        }

    }

}
```

Motor.c

```
#include "motor.h"

#include "../MCAL/GPIO/GPIO_interface.h"


void initMotor(Motor *motor, float velocityPercentage,
```

```
              TIM_TypeDef *TIMX,
              GPIO_TypeDef *EN, uint8_t ENNum,
              GPIO_TypeDef *IN1, uint8_t IN1Num,
              GPIO_TypeDef *IN2, uint8_t IN2Num) {

    motor->dir = FORWARD;// default direction
    motor->speed = velocityPercentage;
    motor->EN = EN;
    motor->IN1 = IN1;
    motor->IN2 = IN2;
    motor->ENNum = ENNum;
    motor->IN1Num = IN1Num;
    motor->IN2Num = IN2Num;
    motor-> TIMX = TIMX;
    TIM_initPWM( motor-> TIMX, motor-> Ch, velocityPercentage);
    setDir(motor, FORWARD);
}

void setDir(Motor *motor, DIR dir) {
    motor->dir = dir;
    if (dir == REVERSE) {  // IN1 IN2  -> 01
        GPIO_digitalWrite(motor->IN1, motor->IN1Num, LOW);
        GPIO_digitalWrite(motor->IN2, motor->IN2Num, HIGH);
    } else { // IN1 IN2  -> 10
        GPIO_digitalWrite(motor->IN1, motor->IN1Num, HIGH);
        GPIO_digitalWrite(motor->IN2, motor->IN2Num, LOW);
    }
}
```

```c
DIR getDir(Motor *motor) {
    return motor->dir;
}

void setSpeed(Motor *motor, float velocityPercentage) {
    motor->speed = velocityPercentage;
    TIM_writePWM(motor-> TIMX, motor-> Ch, velocityPercentage);
}

float getSpeed(Motor *motor) {
    return motor->speed;
}

void stop(Motor *motor) {  // IN1 IN2  -> 11
    GPIO_digitalWrite(motor->EN, motor->ENNum, LOW);
    GPIO_digitalWrite(motor->IN1, motor->IN1Num, LOW);
    GPIO_digitalWrite(motor->IN2, motor->IN2Num, LOW);
}
```

CAR_program.c

```c
#include "CAR_interface.h"
#include "../../MCAL/TIMR/TIM_interface.h"
#include "../../MCAL/GPIO/GPIO_interface.h"
```

```c
// Global variables for motor control
TIM_TypeDef *TimxLeft_global;
TIM_TypeDef *TimxRight_global;
uint8_t ChannelLeft_global;
uint8_t ChannelRight_global;

GPIO_TypeDef *DirLeft1_global;
GPIO_TypeDef *DirLeft2_global;
GPIO_TypeDef *DirRight1_global;
GPIO_TypeDef *DirRight2_global;

uint8_t DirLeft1Pin_global;
uint8_t DirLeft2Pin_global;
uint8_t DirRight1Pin_global;
uint8_t DirRight2Pin_global;

void CAR_init(TIM_TypeDef *TimxLeft ,uint8_t ChannelLeft, float FrequencyLeft,
              GPIO_TypeDef *DirLeft1, GPIO_TypeDef *DirLeft2, uint8_t
DirLeft1Pin, uint8_t DirLeft2Pin,
              TIM_TypeDef *TimxRight ,uint8_t ChannelRight, float
FrequencyRight,
              GPIO_TypeDef *DirRight1, GPIO_TypeDef *DirRight2, uint8_t
DirRight1Pin, uint8_t DirRight2Pin) {

    // Store GPIO pointers and pin numbers
    DirLeft1_global = DirLeft1;
    DirLeft2_global = DirLeft2;
    DirRight1_global = DirRight1;
    DirRight2_global = DirRight2;
```

```
    DirLeft1Pin_global = DirLeft1Pin;
    DirLeft2Pin_global = DirLeft2Pin;
    DirRight1Pin_global = DirRight1Pin;
    DirRight2Pin_global = DirRight2Pin;

    // Configure all direction pins as outputs
    GPIO_pinMode(DirLeft1, DirLeft1Pin, OUTPUT);
    GPIO_pinMode(DirLeft2, DirLeft2Pin, OUTPUT);
    GPIO_pinMode(DirRight1, DirRight1Pin, OUTPUT);
    GPIO_pinMode(DirRight2, DirRight2Pin, OUTPUT);

    // Store timer and channel configuration
    TimxLeft_global = TimxLeft;
    TimxRight_global = TimxRight;
    ChannelLeft_global = ChannelLeft;
    ChannelRight_global = ChannelRight;

    // Initialize PWM channels
    TIM_initPWM(TimxLeft, ChannelLeft, FrequencyLeft);
    TIM_initPWM(TimxRight, ChannelRight, FrequencyRight);
}

void CAR_forward(float rightSpeed ,float leftSpeed) {
    // Left motor forward
    GPIO_digitalWrite(GPIOB, 9, HIGH); // yellow high
    GPIO_digitalWrite(GPIOB, 8, LOW); // green low
```

```
    GPIO_digitalWrite(DirLeft1_global, DirLeft1Pin_global, HIGH);
    GPIO_digitalWrite(DirLeft2_global, DirLeft2Pin_global, LOW);
    // Right motor forward
    GPIO_digitalWrite(DirRight1_global, DirRight1Pin_global, HIGH);
    GPIO_digitalWrite(DirRight2_global, DirRight2Pin_global, LOW);

    TIM_writePWM(TimxLeft_global, ChannelLeft_global, leftSpeed);
    TIM_writePWM(TimxRight_global, ChannelRight_global, rightSpeed);
}

void CAR_backwards(float rightSpeed , float leftSpeed) {
    // Left motor backward
    GPIO_digitalWrite(GPIOB, 9, HIGH); // yellow high
    GPIO_digitalWrite(GPIOB, 8, LOW); // green low


    GPIO_digitalWrite(DirLeft1_global, DirLeft1Pin_global, LOW);
    GPIO_digitalWrite(DirLeft2_global, DirLeft2Pin_global, HIGH);
    // Right motor backward
    GPIO_digitalWrite(DirRight1_global, DirRight1Pin_global, LOW);
    GPIO_digitalWrite(DirRight2_global, DirRight2Pin_global, HIGH);

    TIM_writePWM(TimxLeft_global, ChannelLeft_global, leftSpeed);
    TIM_writePWM(TimxRight_global, ChannelRight_global, rightSpeed);
}

void CAR_right(float leftSpeed, float rightSpeed) {
```

```
    // Left motor forward
    GPIO_digitalWrite(GPIOB, 9, HIGH); // yellow high
    GPIO_digitalWrite(GPIOB, 8, LOW); // green low


    GPIO_digitalWrite(DirLeft1_global, DirLeft1Pin_global, HIGH);
    GPIO_digitalWrite(DirLeft2_global, DirLeft2Pin_global, LOW);
    // Right motor backward
    GPIO_digitalWrite(DirRight1_global, DirRight1Pin_global, LOW);
    GPIO_digitalWrite(DirRight2_global, DirRight2Pin_global, HIGH);

    TIM_writePWM(TimxLeft_global, ChannelLeft_global, leftSpeed);
    TIM_writePWM(TimxRight_global, ChannelRight_global, rightSpeed);
}

void CAR_left(float rightSpeed, float leftSpeed) {
    // Left motor backward
    GPIO_digitalWrite(GPIOB, 9, HIGH); // yellow high
    GPIO_digitalWrite(GPIOB, 8, LOW); // green low


    GPIO_digitalWrite(DirLeft1_global, DirLeft1Pin_global, LOW);
    GPIO_digitalWrite(DirLeft2_global, DirLeft2Pin_global, HIGH);
    // Right motor forward
    GPIO_digitalWrite(DirRight1_global, DirRight1Pin_global, HIGH);
    GPIO_digitalWrite(DirRight2_global, DirRight2Pin_global, LOW);

    TIM_writePWM(TimxLeft_global, ChannelLeft_global, leftSpeed);
```

```
    TIM_writePWM(TimxRight_global, ChannelRight_global, rightSpeed);
}

void CAR_stop() {
    GPIO_digitalWrite(GPIOB, 9, LOW); // yellow high

    GPIO_digitalWrite(GPIOB, 8, HIGH); // green low

    // Brake both motors
    GPIO_digitalWrite(DirLeft1_global, DirLeft1Pin_global, LOW);
    GPIO_digitalWrite(DirLeft2_global, DirLeft2Pin_global, LOW);
    GPIO_digitalWrite(DirRight1_global, DirRight1Pin_global, LOW);
    GPIO_digitalWrite(DirRight2_global, DirRight2Pin_global, LOW);

    TIM_writePWM(TimxLeft_global, ChannelLeft_global, 0);
    TIM_writePWM(TimxRight_global, ChannelRight_global, 0);
}
```

CONTROL_program.c

```
#include "CONTROL_interface.h"
float kp_global;
float kd_global;
float prev_error=0;
float prev_time=0;
float speed;
```

```
float kp_angle = 0;
float kd_angle = 0;
float prev_angle_error = 0;
float prev_angle_time = 0;
float servo_output = 0;
#define maxDistance 30 //5cm from target
#define mainAngle 90

void PD_init_angle(float Kp, float Kd) {
    kp_angle = Kp;
    kd_angle = Kd;
}

// Update angle control, currentAngle and targetAngle in degrees
void PD_update_angle(float currentAngle, uint64_t time_ms) {
    float error = mainAngle - currentAngle;  // desired - current
    float dt = (time_ms - prev_angle_time) / 1000.0f;
    if (dt <= 0) dt = 0.001f;

    float derivative = (error - prev_angle_error) ;

    prev_angle_error = error;
    prev_angle_time = time_ms;

    float steering_correction = kp_angle * error + kd_angle * derivative;

    // Clamp correction
    if (steering_correction > 60) steering_correction = 60;
```

```
    else if (steering_correction < -60) steering_correction = -60;


    // Use global forward speed (assumed non-negative)

    float base_speed = speed;

    float right_motor_speed;

    float left_motor_speed;

    if (speed==0)

CAR_stop();


    if (error < 0) {

        // Turn left: right motor faster, left motor slower

        left_motor_speed = base_speed + steering_correction;

        right_motor_speed = base_speed - steering_correction;

    } else if (error > 0) {

        // Turn right: left motor faster, right motor slower

        left_motor_speed = base_speed - (-steering_correction);  //
steering_correction negative here

        right_motor_speed = base_speed + (-steering_correction);

    }


    // Clamp motor speeds to [0, 100]

    if (left_motor_speed > 100) left_motor_speed = 100;

    if (left_motor_speed < -100) left_motor_speed = -100;

    if (right_motor_speed > 100) right_motor_speed = 100;

    if (right_motor_speed < -100) right_motor_speed = -100;

    if(fabs(error)<1){

        right_motor_speed=0;

        left_motor_speed=0;

    }
```

```
    // Drive motors forward with computed speeds

    if(right_motor_speed>0 && left_motor_speed>0){
    CAR_forward(right_motor_speed, left_motor_speed);
    }
    else if(right_motor_speed<0 && left_motor_speed<0){
        CAR_backwards(-right_motor_speed, -left_motor_speed);
    }
    else if(right_motor_speed>0 && left_motor_speed<0){
            CAR_left(right_motor_speed, -left_motor_speed);
        }
    else if(right_motor_speed<0 && left_motor_speed>0){
            CAR_right(left_motor_speed, -right_motor_speed);
        }
}



void PD_init( float Kp, float Kd)
{
kp_global=Kp;
kd_global=Kd;

}
void PD_update_from_distance(float actualDistance, uint64_t time_ms)
{
```

```
    float error = actualDistance - maxDistance;
    if(error<0){
        kp_global=2;
        kd_global=8;

    }
    float p = kp_global * error;
    float d = kd_global*(error - prev_error) ;
    prev_error = error;
    prev_time = time_ms;

    speed = p + d;

    // Clamp speed to [-100, 100]
    if (speed > 100.0f) {
        speed = 100.0f;
    } else if (speed < -100.0f) {
        speed = -100.0f;
    }

    // Apply deadband threshold
//    if (speed > 0.0f && speed < 30.0f) {
//        speed = 30.0f;  // Minimum forward speed
//    } else if (speed < 0.0f && speed > -30.0f) {
//        speed = -30.0f; // Minimum backward speed
//    }
if(fabs(error)<6){
    speed=0;
```

```
}
    // Movement logic
//    if (speed > 0) {
//        CAR_forward(speed,speed);
//    } else if (speed < 0) {
//        CAR_backwards(-speed,-speed);
//    } else {
//        CAR_stop();
//    }
}
```

GPIO_program.c

```
#include "GPIO_interface.h"

void GPIO_pinMode(GPIO_TypeDef *GPIOX, uint8_t pinNumber, GPIO_MODE mode){
    if (pinNumber < 0 || pinNumber > 15){
        return;
    }
    // Initialize the clock of port x
    if (GPIOX == GPIOA) {
        SET_BIT(RCC->APB2ENR, 2);
    } else if (GPIOX == GPIOB) {
        SET_BIT(RCC->APB2ENR, 3);
    } else if (GPIOX == GPIOC) {
        SET_BIT(RCC->APB2ENR, 4);
```

```
    }
    volatile uint8_t pinIndex = pinNumber % 8;


    volatile uint32_t *CRX;
    if (pinNumber < 8 && pinNumber >= 0){
        CRX = &GPIOX->CRL;
    }else if(pinNumber >= 8 && pinNumber < 16){
        CRX = &GPIOX->CRH;
    }
    // Zero the CRX register's specific pin mode not the whole register
    *CRX &= ~(0xF << (4*(pinIndex)));
    if (mode == OUTPUT){
        *CRX |= (0x2 << (4*(pinIndex)));
    }else if (mode == INPUT_FLOAT){
        *CRX |= (0x4 << (4*(pinIndex)));
    }else if (mode == INPUT_PULLUP){
        *CRX |= (0x8 << (4*(pinIndex)));
        SET_BIT(GPIOX->ODR, pinNumber);
    }else if (mode == INPUT_PULLDOWN){
        *CRX |= (0x8 << (4*(pinIndex)));
        CLEAR_BIT(GPIOX->ODR, pinNumber);
    }else if (mode == AF_PP){
        *CRX |= (0xB << (4*(pinIndex)));


    }
}


void GPIO_digitalWrite(GPIO_TypeDef *GPIOX, uint8_t pinNumber, PIN_LEVEL
level){
```

```
    if (pinNumber < 0 || pinNumber > 15){
        return;
    }
    if (level == HIGH){
        SET_BIT(GPIOX->ODR, pinNumber);
    }else if(level == LOW){
        CLEAR_BIT(GPIOX->ODR, pinNumber);
    }
}

uint8_t GPIO_digitalRead(GPIO_TypeDef *GPIOX, uint8_t pinNumber){
    if (pinNumber < 0 || pinNumber > 15){
        return 99;
    }
    return READ_BIT(GPIOX->IDR, pinNumber);
}

void GPIO_digitalToggle(GPIO_TypeDef *GPIOX, uint8_t pinNumber){
    if (pinNumber < 0 || pinNumber > 15){
        return;
    }
    TOGGLE_BIT(GPIOX->ODR, pinNumber);
}
```

UART_program.c

```
#include "UART_interface.h"
#include "stm32f103xb.h"
```

```c
#include "../../UTIL/BIT_MATH.h"

#include "../GPIO/GPIO_interface.h"


#define RX_BUFFER_LEN 64


// Internal parser state variables

static enum { WAIT_START, READ_DATA, READ_CHECKSUM, WAIT_END } state =
WAIT_START;

static uint8_t buffer[3];

static uint8_t data_index = 0;

static uint8_t checksum = 0;

volatile UARTMessage received_msg;

volatile uint8_t message_ready = 0;


// === Variables ===

volatile int distance = 0;

volatile int angle = 0;

volatile uint8_t command = 0;

volatile int dir = 0;




void UART_init(int UART_pref_num, int baudrate)

{

    USART_TypeDef *USARTx;

    uint32_t pclk = 8000000; // 8 MHz clock

    //uint32_t brr_value; //unused var


    switch (UART_pref_num)
```

```
{
case 1:
    SET_BIT(RCC->APB2ENR, 14);            // USART1
    SET_BIT(RCC->APB2ENR, 2);             // GPIOA
    GPIO_pinMode(GPIOA, 9, AF_PP);        // TX
    GPIO_pinMode(GPIOA, 10, INPUT_FLOAT); // RX
    USARTx = USART1;
    break;

case 2:
    SET_BIT(RCC->APB1ENR, 17);           // USART2
    SET_BIT(RCC->APB2ENR, 2);            // GPIOA
    GPIO_pinMode(GPIOA, 2, AF_PP);       // TX
    GPIO_pinMode(GPIOA, 3, INPUT_FLOAT); // RX
    USARTx = USART2;
    break;

case 3:
    SET_BIT(RCC->APB1ENR, 18);            // USART3
    SET_BIT(RCC->APB2ENR, 3);             // GPIOB
    GPIO_pinMode(GPIOB, 10, AF_PP);       // TX
    GPIO_pinMode(GPIOB, 11, INPUT_FLOAT); // RX
    USARTx = USART3;
    break;

default:
    return;
}
```

```
    // Baud Rate Calculation for 8 MHz clock
    float usartdiv = (float)pclk / (16.0f * baudrate);
    uint16_t mantissa = (uint16_t)usartdiv;
    uint16_t fraction = (uint16_t)((usartdiv - mantissa) * 16.0f + 0.5f); //
rounded

    USARTx->BRR = (mantissa << 4) | (fraction & 0xF);

    // Enable USART, TX, RX
    SET_BIT(USARTx->CR1, 13); // UE
    SET_BIT(USARTx->CR1, 3);  // TE
    SET_BIT(USARTx->CR1, 2);  // RE
}

void UART1_InterruptsInit(void) {
    // USART1 configuration...
    USART1->CR1 |= USART_CR1_RXNEIE; // Enable RX interrupt
    NVIC_EnableIRQ(USART1_IRQn);     // Enable USART1 interrupt in NVIC
}


void UART_send(int UART_pref_num, int data)
{
    USART_TypeDef *USARTx;

    switch (UART_pref_num)
    {
    case 1: USARTx = USART1; break;
```

```
    case 2: USARTx = USART2; break;

    case 3: USARTx = USART3; break;

    }


    while (!(USARTx->SR & (1 << 7)));

    USARTx->DR = (data & 0xFF);

}


int UART_Receive(int UART_pref_num)

{

    USART_TypeDef *USARTx;


    switch (UART_pref_num)

    {

    case 1: USARTx = USART1; break;

    case 2: USARTx = USART2; break;

    case 3: USARTx = USART3; break;

    }


    while (!(USARTx->SR & (1 << 5)));


    return USARTx->DR & 0xFF;

}


UARTMessage UART_receive_message(int UART_pref_num) {

    USART_TypeDef *USARTx;

    switch (UART_pref_num) {

        case 1: USARTx = USART1; break;
```

```c
        case 2: USARTx = USART2; break;

        case 3: USARTx = USART3; break;

        default: return (UARTMessage){ .type = MSG_NONE };

    }


    if (!(USARTx->SR & USART_SR_RXNE)){

           if (USART1->SR & USART_SR_ORE) {

                  volatile uint32_t tmp = USART1->DR;  // Clear ORE by reading
DR

                  (void)tmp;

              }

        return (UARTMessage){ .type = MSG_NONE };

    }


    uint8_t byte = USARTx->DR & 0xFF;


    switch (state) {

        case WAIT_START:

            if (byte == 0xAA) {

                data_index = 0;

                checksum = 0;

                state = READ_DATA;

            }

            break;


        case READ_DATA:

            buffer[data_index++] = byte;

            checksum ^= byte;

            if (data_index == 3) state = READ_CHECKSUM;
```

```
            break;

        case READ_CHECKSUM:
            if (byte == checksum) {
                state = WAIT_END;
            } else {
                state = WAIT_START;
            }
            break;

        case WAIT_END:
            if (byte == 0x55) {
                while (!(USARTx->SR & USART_SR_TXE));
                USARTx->DR = 0xCC;

                uint32_t packed = (buffer[0] << 16) | (buffer[1] << 8) |
buffer[2];
                int command = (packed >> 23) & 0x01;
                int dir = (packed >> 22) & 0x01;
                int distance = (packed >> 8) & 0x3FFF;
                int angle = packed & 0xFF;

                UARTMessage msg = {
                    .type = MSG_COMMAND_DISTANCE_ANGLE,
                    .command = command,
                    .dir = dir,
                    .distance = distance,
                    .angle = angle
                };
```

```
                state = WAIT_START;
                return msg;
            } else {
                state = WAIT_START;
            }
            break;
    }

    return (UARTMessage){ .type = MSG_NONE };
}

void USART1_IRQHandler(void) {

    UARTMessage msg = UART_receive_message(1);
    if (msg.type != MSG_NONE) {
        received_msg = msg;
        message_ready = 1;
        distance = msg.distance;
        dir = msg.dir;
        angle = msg.angle;
        command = msg.command;
        distance = dir? -distance : distance;

    }
}
```

Main.cpp

```cpp
#include <main.h>


const char* ssid = "SSH";

const char* password = "AzabSSH359";

const char* mqtt_server = "192.168.0.69"; //IP Address


const char* mqtt_client_id = "ESP32_" TOSTRING(CAR_COLOUR) "Car";

const char* mqtt_sub_laptopCMD = "laptop/commands/" TOSTRING(CAR_COLOUR)
"car";

const char* mqtt_pub_topic = "sensor/" TOSTRING(CAR_COLOUR) "car";

const char* mqtt_sub_joyRos = "joyROS/" TOSTRING(CAR_COLOUR) "car/cmd";




WiFiClient espClient;

PubSubClient client(espClient);


HardwareSerial stm32Serial(2); // UART2: TX2=17, RX2=16


SemaphoreHandle_t xSharedDataMutex;


u16_t dummydistance_cm = 12;

volatile bool go_command = false;

bool isAutonomous = true;

volatile int Sensordistance= 9999;

volatile int Sensorangle = 90;
```

```
void setup() {
  Serial.begin(115200);
  setup_led();

  xSharedDataMutex = xSemaphoreCreateMutex();
    if (xSharedDataMutex == NULL) {
      while (1) {
        Serial.println("Failed to create mutex");
      }
    }
    // Blocking the flow till the wi-fi is connected
    connectToWiFi(ssid, password);
    tofInit();
    stepperInit();
    delay(2000); // Give time for Serial to connect

    setupSTM32Serial(stm32Serial, rx_pin, tx_pin);
    setupMQTT(mqtt_server, mqtt_client_id, mqtt_sub_laptopCMD,
mqtt_sub_joyRos);

    // Comms on core 0
    xTaskCreatePinnedToCore(WiFiTask, "WiFiTask", 4096, NULL, 1, NULL, 0);
    xTaskCreatePinnedToCore(MQTTTask, "MQTTTask", 8192, NULL, 2, NULL, 0);
    xTaskCreatePinnedToCore(SerialTask, "SerialTask", 4096, NULL, 2, NULL, 0);

    xTaskCreatePinnedToCore(
          tof_task,          // Function name
          "tof",             // Task name
```

```
        2048,               // Stack size
        NULL,               // Parameter
        1,                  // Priority
        NULL,               // Task handle
        1                   // Core 1
    );

    xTaskCreatePinnedToCore(
        stepper_task,       // Function name
        "stepper",          // Task name
        4096,               // Stack size
        NULL,               // Parameter
        2,                  // Priority
        NULL,               // Task handle
        1                   // Core 1
    );

}


void loop(){}
```

Lidar.cpp

```
#include "lidar.h"
```

```
#include <main.h>


void tof_task(void *parameter){

  while(1){

    TOF_Active_Decoding(); // Query and decode TOF data

  }

}


void stepper_task(void *parameter){

  float span = 126.0;

  float min_angle_steps = 0, max_angle_steps = 0;

  bool mask = false, mask2 = false;

  int counter = 0;

  float min_angle = 0.0 + ((180.0-span)/2.0);

  float max_angle = 180.0 - ((180.0-span)/2.0);

  float angle1 = 0.0, angle2 = 0.0, dist1 = 0.0, dist2 = 0.0, prev_distance =
0.0, prev_angle = 0.0;

  float current_angle;

  TickType_t xLastWakeTime;

  while(1){

    //////////////////////////////////////// Calibration
//////////////////////////////////////

    if (!digitalRead(calibration_pin) && !mask){

      counter ++;

      mask = true;

    }else if(digitalRead(calibration_pin) && mask){

      mask = false;

    }

    if (counter % 2 == 0 && !mask2){
```

```
    digitalWrite(en_pin, HIGH); // deactivate the stepper
    Serial.println("stepper off");
    min_angle_steps = 0;
    max_angle_steps = 0;
    mask2 = true;
  }else if(counter % 2 != 0 && mask2){
    digitalWrite(en_pin, LOW); // reactivate the stepper
    Serial.println("stepper on");
    // transforming angle to steps
    min_angle_steps = min_angle/resolution;
    max_angle_steps = max_angle/resolution;
    digitalWrite(dir_pin, HIGH); // AntiClockwise
    vTaskDelay(50 / portTICK_PERIOD_MS);  // for stability
    for (int i = 0; i < min_angle_steps; i++){ // to go to the min angle
      digitalWrite(step_pin, HIGH);
      vTaskDelay(2 / portTICK_PERIOD_MS);
      digitalWrite(step_pin, LOW);
      vTaskDelay(2 / portTICK_PERIOD_MS);
    }
    xLastWakeTime = xTaskGetTickCount();
    mask2 = false;
  }

////////////////////////////////////////////////////////////////////////////////////
  float angle_at_min_dist = 181;
  uint32_t min_dist = 9999;  // set high starting value
  digitalWrite(dir_pin, HIGH); // AntiClockwise
  vTaskDelay(5 / portTICK_PERIOD_MS);  // for stability
```

```
for (int i = min_angle_steps; i <= max_angle_steps; i ++){
  digitalWrite(step_pin, HIGH);
  vTaskDelayUntil(&xLastWakeTime, 1 / portTICK_PERIOD_MS);
  digitalWrite(step_pin, LOW);
  vTaskDelayUntil(&xLastWakeTime, 1 / portTICK_PERIOD_MS);
  current_angle = i*resolution;
  uint32_t current_dist = TOF_0.dis/10;  // Get stable snapshot / by 10

  if (current_dist < min_dist) {
    min_dist = current_dist;
    angle_at_min_dist = current_angle;
  }
}

angle1 = angle_at_min_dist;
dist1 = min_dist;
angle1 = CLAMP(angle_at_min_dist, (min_angle + shifting_angle_factor),
(max_angle - shifting_angle_factor));

if (TOF_0.dis_status == 0){
  dist1 = 9999; // If the distance is not valid, set it to 6999
}
xSemaphoreTake(xSharedDataMutex, portMAX_DELAY);
Sensordistance = dist1;
Sensorangle = angle1;
xSemaphoreGive(xSharedDataMutex);

Serial.print("(");
Serial.print(angle1);
```

```
    Serial.print(",");
    Serial.print(dist1);
    Serial.println(")");

    angle_at_min_dist = 181;
    min_dist = 9999;  // set high starting value
    digitalWrite(dir_pin, LOW); // clockwise
    vTaskDelay(5 / portTICK_PERIOD_MS);  // for stability
    for (int i = max_angle_steps; i >= min_angle_steps; i --){
      digitalWrite(step_pin, HIGH);
      vTaskDelayUntil(&xLastWakeTime, 1 / portTICK_PERIOD_MS);
      digitalWrite(step_pin, LOW);
      vTaskDelayUntil(&xLastWakeTime, 1 / portTICK_PERIOD_MS);
      current_angle = i*resolution + shifting_angle_factor;

      uint32_t current_dist = TOF_0.dis/10;  // Get stable snapshot / by 10

      if (current_dist <= min_dist) {
        min_dist = current_dist;
        angle_at_min_dist = current_angle;
      }
    }
    angle2 = angle_at_min_dist;
    dist2 = min_dist;
    angle2 = CLAMP(angle_at_min_dist, (min_angle + shifting_angle_factor),
(max_angle - shifting_angle_factor));

    if (TOF_0.dis_status == 0){
      dist2 = 9999; // If the distance is not valid, set it to 6999
```

```
    }
    xSemaphoreTake(xSharedDataMutex, portMAX_DELAY);
    Sensordistance = dist2;
    Sensorangle = angle2;
    xSemaphoreGive(xSharedDataMutex);

    Serial.print("(");
    Serial.print(angle2);
    Serial.print(",");
    Serial.print(dist2);
    Serial.println(")");

  }
}

void stepperInit(){
  pinMode(dir_pin, OUTPUT);
  pinMode(step_pin, OUTPUT);
  pinMode(calibration_pin, INPUT_PULLUP);
  pinMode(en_pin, OUTPUT);
  digitalWrite(en_pin, HIGH); // deactivate the stepper
}

void tofInit(){
  TOF_UART.begin(1500000, SERIAL_8N1, TOF_RX_PIN, TOF_TX_PIN);
}
```

Comms.cpp

```
#include <main.h>

#include "comm.h"


void WiFiTask(void *pvParameters) {

  while (true) {

    if (WiFi.status() != WL_CONNECTED) {

      connectToWiFi(ssid, password);

    }

    vTaskDelay(pdMS_TO_TICKS(1000));

  }

}


void MQTTTask(void *pvParameters) {

  while (true) {

    if (!client.connected()) {

      connect_mqttServer();

    }


    client.loop();

    int dis = 0;

    xSemaphoreTake(xSharedDataMutex,portMAX_DELAY);

    dis = Sensordistance;

    xSemaphoreGive(xSharedDataMutex);


    String message = String(dis);

    publishMessage(mqtt_pub_topic, message);
```

```
    vTaskDelay(pdMS_TO_TICKS(100));
  }
}

void SerialTask(void *pvParameters) {
  while (true) {
    bool command = false;
    xSemaphoreTake(xSharedDataMutex,portMAX_DELAY);
    command = go_command;
    bool autonomous = isAutonomous;
    xSemaphoreGive(xSharedDataMutex);

    if (command && autonomous ){
      int dis = 0;
      int angle = 0;
      xSemaphoreTake(xSharedDataMutex,portMAX_DELAY);
      dis = Sensordistance;
      angle = Sensorangle;
      xSemaphoreGive(xSharedDataMutex);
      sendPackedToSTM32(false ,dis, angle);
    }
    else if (autonomous){
      sendPackedToSTM32(false, defaultStopDistance, defaultStopAngle);
    }
    vTaskDelay(pdMS_TO_TICKS(150));
  }
}
```

ledAsIndicator.cpp

```
#define ledPin 2
#include <main.h>

void blink_led(unsigned int times, unsigned int duration) {
    for (unsigned int i = 0; i < times; i++) {
        digitalWrite(ledPin, HIGH);
        delay(duration);
        digitalWrite(ledPin, LOW);
        vTaskDelay(pdMS_TO_TICKS(200));
    }
}

void setup_led() {
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);
}
```

Mqtt_utils.cpp

```
#include <main.h>

long lastMsg = 0;
bool firstTime = true;

void setupMQTT(const char* server, const char* client_id, const char* topic,
const char* topic2) {
```

```
    client.setServer(server, 1883);
    client.setCallback(mqttCallback);

    while (!client.connected()) {
        Serial.print("Connecting to MQTT...");

        if (client.connect(client_id)) {
            Serial.println("connected.");
            client.subscribe(topic);
            client.subscribe(topic2);
        } else {
            Serial.print("failed. rc=");
            Serial.print(client.state());
            Serial.println(" trying again in 2 seconds");

            blink_led(3,200); //blink LED three times (200ms on duration) to
show that MQTT server connection attempt failed

            delay(2000);

        }
    }
}

void connect_mqttServer() {
    if (!client.connected()) {
        setupMQTT(mqtt_server, mqtt_client_id, mqtt_sub_laptopCMD,
mqtt_sub_joyRos);
    }
```

```
    client.loop();

    delay(50);

}


void mqttCallback(char* topic, byte* message, unsigned int length) {

    String msg = "";

    for (unsigned int i = 0; i < length; i++) {

      msg += (char)message[i];

    }


    if (strcmp(topic, mqtt_sub_laptopCMD) == 0) {

        if (msg == "GO") {

            setCommandSTM32(MOVECOMMAND::GO);

        }

        else if (msg == "STOP") {

            setCommandSTM32(MOVECOMMAND::STOP);


        }

        else if (msg == "Manual Mode") {

            setCommandSTM32(MOVECOMMAND::ManualMode);

        }

    }

    if (strcmp(topic, mqtt_sub_joyRos) == 0) {

        unsigned long raw = msg.toInt();


        int command = (raw >> 23) & 0x01;

        int direction = (raw >> 22) & 0x01;

        int distance = (raw >> 8) & 0x3FFF;
```

```
        //int angleSign = (raw >> 7) & 0x01;
        int angleMag = raw & 0xFF;

        //int angle = angleSign ? -angleMag : angleMag;

        // xSemaphoreTake(xSharedDataMutex, portMAX_DELAY);
        // go_command = (command == 1);
        // xSemaphoreGive(xSharedDataMutex);

        bool dir = (direction == 1);

        sendPackedToSTM32Manual(command, direction, distance, angleMag);
    }

}

void publishMessage(const char* topic, const String& payload) {

    long now = millis();
    if (client.connected() && (now - lastMsg > 100)) {
        lastMsg = now;
        client.publish(topic, payload.c_str());
    }
}
```

Serial_utils.cpp

```
#include <main.h>

void setupSTM32Serial(HardwareSerial& serial, int rxPin, int txPin) {
    serial.begin(baudrate, SERIAL_8N1, rxPin, txPin);
    setCommandSTM32(MOVECOMMAND::STOP);
}

void sendPackedToSTM32(bool direction, u16_t distance, u8_t angle) {
    if (distance > 16383 || angle > 255 || angle < 0) {
        Serial.println("Invalid distance or angle range");
        sendPackedToSTM32(direction,10,90);
        return;
    }

    uint8_t buffer[6];
    buffer[0] = START_BYTE;

    // Pack bits: [command:1][dir:1][distance:14][angle:8]
    uint32_t packed = 0;
    packed |= ((go_command ? 1 : 0) & 0x01) << 23;      // Bit 23
    packed |= ((direction ? 1 : 0) & 0x01) << 22;       // Bit 22
    packed |= (distance & 0x3FFF) << 8;                 // Bits 21-8 (14 bits)
    packed |= (abs(angle) & 0xFF);                      // Bits 7-0 (8 bits)

    buffer[1] = (packed >> 16) & 0xFF;
```

```
    buffer[2] = (packed >> 8) & 0xFF;

    buffer[3] = packed & 0xFF;

    buffer[5] = END_BYTE;


    // Checksum = XOR of payload bytes

    buffer[4] = buffer[1] ^ buffer[2] ^ buffer[3];


    stm32Serial.write(buffer, sizeof(buffer));


    // Wait for ACK

    unsigned long start = millis();

    while (millis() - start < 100) {

        if (stm32Serial.available()) {

            uint8_t ack = stm32Serial.read();

            if (ack == ACK_BYTE) {

                Serial.println("ACK received from STM32");

                return;

            }

        }

    }

    Serial.println("⚠️ No ACK received");

}


void sendPackedToSTM32Manual(bool command,bool direction, u16_t distance, u8_t
angle) {

    if (distance > 16383 || angle > 255 || angle < 0) {

        Serial.println("Invalid distance or angle range");

        sendPackedToSTM32(direction,10,90);

        return;
```

```
    }


    uint8_t buffer[6];
    buffer[0] = START_BYTE;


    // Pack bits: [command:1][dir:1][distance:14][angle:8]
    uint32_t packed = 0;
    packed |= ((command ? 1 : 0) & 0x01) << 23;      // Bit 23
    packed |= ((direction ? 1 : 0) & 0x01) << 22;       // Bit 22
    packed |= (distance & 0x3FFF) << 8;                 // Bits 21-8 (14 bits)
    packed |= (abs(angle) & 0xFF);                      // Bits 7-0 (8 bits)


    buffer[1] = (packed >> 16) & 0xFF;
    buffer[2] = (packed >> 8) & 0xFF;
    buffer[3] = packed & 0xFF;
    buffer[5] = END_BYTE;


    // Checksum = XOR of payload bytes
    buffer[4] = buffer[1] ^ buffer[2] ^ buffer[3];


    stm32Serial.write(buffer, sizeof(buffer));


    // Wait for ACK
    unsigned long start = millis();
    while (millis() - start < 100) {
        if (stm32Serial.available()) {
            uint8_t ack = stm32Serial.read();
            if (ack == ACK_BYTE) {
```

```
                Serial.println("ACK received from STM32");

                return;

            }

        }

    }

    Serial.println("⚠️ No ACK received");

}




void setCommandSTM32(MOVECOMMAND command) {

    switch (command) {


    case MOVECOMMAND::GO:

        xSemaphoreTake(xSharedDataMutex, portMAX_DELAY);

        go_command = true;

        isAutonomous = true;

        xSemaphoreGive(xSharedDataMutex);

        break;


    case MOVECOMMAND::STOP:

        xSemaphoreTake(xSharedDataMutex, portMAX_DELAY);

        go_command = false;

        isAutonomous = true;

        xSemaphoreGive(xSharedDataMutex);

        break;


    case MOVECOMMAND::ManualMode:
```

```
        xSemaphoreTake(xSharedDataMutex,portMAX_DELAY);

        isAutonomous = false;

        xSemaphoreGive(xSharedDataMutex);

        break;


    default:

        Serial.println("Unknown command");

  }

}
```

Wifi_utils.cpp

```
#include <main.h>



void connectToWiFi(const char* ssid, const char* password) {

    delay(50);

    Serial.println();

    Serial.print("Connecting to ");

    Serial.println(ssid);


    WiFi.begin(ssid, password);


    char c = 0; //counter for number of attempts
```

```
    while (WiFi.status() != WL_CONNECTED) {
        //blink LED twice (for 200ms ON time) to indicate that wifi not
connected
        blink_led(2,200);
        delay(1000);
        Serial.print(".");
        c++;
        if (c > 20) {
            Serial.println("\nFailed to connect to WiFi. Please check your
credentials.");
            ESP.restart();
            return;
        }
    }
    Serial.println("\nWiFi connected: " + WiFi.localIP().toString());
}
```

Joy_to_cmd_node.py

```
import rclpy
import time
from rclpy.node import Node
from sensor_msgs.msg import Joy
from triflamex_ros2_pkg.UTIL import pack_payload, Car
from triflamex_ros2_pkg.UTIL import ENDC, COLOR_CODES
from triflamex_ros2_pkg.UTIL import reliable_publish
from triflamex_ros2_pkg.UTIL import MQTT_BROKER as MQTT_BROKER
```

```python
speed_array = [100, 110, 165]

angle_array = [100, 110, 125]


class JoyToCmd(Node):

    def __init__(self):

        super().__init__('joy_to_cmd')

        self.selected_car = None

        self.first_run = True

        self.speed = speed_array[0]

        self.index = 0

        self.prev_rb_state = 0

        self.get_logger().info('JoyToCmd Node has been started.')


        self.subscription = self.create_subscription(

            Joy,

            '/joy',

            self.joy_callback,

            10)

        self.subscription


    def joy_callback(self, msg):


        if msg.buttons[5]:

            reliable_publish("calibration/enable", "Enable")


        if self.selected_car is None and self.first_run:

            self.get_logger().info('No car selected. Press LB to select a 
car.')
```

```python
            self.first_run = False

        if not msg.buttons[4]: #LB
            self.selected_car = None
            self.index = 0
            self.speed = speed_array[self.index]
            #self.get_logger().info('Car selection has been cleared.')
            return

        prev_selected_car = self.selected_car
        # Button mapping
        if msg.buttons[2]:  # X
            self.selected_car = Car.BLUE
        elif msg.buttons[0]:  # A
            self.selected_car = Car.RED
        elif msg.buttons[1]:  # B
            self.selected_car = Car.BLACK

        if self.selected_car is None:
            return

        current_rb_state = msg.buttons[5]
        if current_rb_state and not self.prev_rb_state:
            # Button was just pressed (rising edge)
            self.index = (self.index + 1) % len(speed_array)
            self.speed = speed_array[self.index]
            self.get_logger().info(f'Speed level:  {self.index+1}')
        self.prev_rb_state = current_rb_state
```

```
        if prev_selected_car != self.selected_car:
            color = COLOR_CODES.get(self.selected_car, "")
            self.get_logger().info(f'{color}Selected car:
{self.selected_car.name}{ENDC}')
            self.get_logger().info(f'Speed level:  {self.index+1}')



        # Validate and clamp joystick axes
        raw_throttle = msg.axes[1]
        raw_angle = msg.axes[3]

        # Dead zone
        if abs(raw_throttle) < 0.1:
            raw_throttle = 0.0
        if abs(raw_angle) < 0.1:
            raw_angle = 0.0

        # Convert to meaningful values
        throttle = int(abs(raw_throttle) * 8500)
        dir = 0 if raw_throttle >= 0 else 1
        angle = int(abs(raw_angle) * 90)
        sign = 0 if raw_angle >= 0 else 1
        command = 1 if abs(raw_throttle) > 0.1 or abs(raw_angle) > 0.1 else 0

        throttle = speed_array[self.index] if throttle > 50 else 10

        if command:
```

```
            angle = 92

        if (angle > 10):
            #throttle = 10
            angle = 60 if (sign) else 120

        if not dir :
            #throttle *2.4
            pass


        #throttle = speed_array[self.index] if throttle > 50 else 30 if
abs(raw_angle) > 0.1 else 10
        #angle = angle_array[self.index] if angle > 10 else 90

        #angle = angle - 90 if (sign) else angle

        # angle = 120 if (sign) else 50 if (raw_angle == 0) else 90
        # #angle = 0 if (raw_angle==0) else 90

        # angle = 92 #60 and 120
        # throttle = 110

        try:
            packed_data = pack_payload(command, dir, throttle, angle)
            payload = str(packed_data)
            topic = f"joyROS/{self.selected_car.name.lower()}car/cmd"

            reliable_publish(topic, payload)
```

```python
            if command == 1:
                time.sleep(0.5)
            else:
                time.sleep(0.5)

        except Exception as e:
            self.get_logger().error(f"Failed to connect to MQTT broker
'{MQTT_BROKER}': {e}")




def main(args=None):
    rclpy.init(args=args)
    node = JoyToCmd()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

config.py

```python
# === Configuration ===
REQUIRED_SSID = "SSH"
MQTT_BROKER = "192.168.0.69"
INTERFACE = "wlp0s20f3"
REQUIRED_IP =  MQTT_BROKER
MQTT_PORT = 1883
MQTT_TOPIC_SUB_BLUE = "sensor/bluecar"
MQTT_TOPIC_SUB_RED = "sensor/redcar"
MQTT_TOPIC_SUB_BLACK = "sensor/blackcar"
MQTT_TOPIC_SUB_BLUE_ROS = "joyROS/bluecar/cmd"
MQTT_TOPIC_SUB_RED_ROS = "joyROS/redcar/cmd"
MQTT_TOPIC_SUB_BLACK_ROS = "joyROS/blackcar/cmd"
MQTT_TOPIC_PUB_BLUE = "laptop/commands/bluecar"
MQTT_TOPIC_PUB_RED = "laptop/commands/redcar"
MQTT_TOPIC_PUB_BLACK = "laptop/commands/blackcar"
MQTT_CLIENT_ID = "Ubuntu_Client"
isMQTTEnabled = False


# === ESP Data ===
blueCar_data = 9999
redCar_data = 9999
blackCar_data = 9999
isBlueCar_live = False
isRedCar_live = False
isBlackCar_live = False
```

```python
# === Mode Data ===
isBlueCarAutonomous = True
isRedCarAutonomous = True
isBlackCarAutonomous = True


# === Enable Mode Data ===
command = True
#enable_topic = "mqtt/enable"
enable_topic = "calibration/enable"
```

checkwifi.py

```python
import os
import sys
import subprocess
import netifaces


from config import (
    REQUIRED_IP,
    REQUIRED_SSID,
    INTERFACE
)


# === Get current IP address ===
```

```python
def get_current_ip(interface):
    try:
        ip = netifaces.ifaddresses(interface)[netifaces.AF_INET][0]['addr']
        return ip
    except Exception as e:
        print(f"❌ Failed to get IP address on {interface}: {e}")
        return None


# === Get current Wi-Fi SSID ===
def get_current_ssid():
    try:
        result = subprocess.check_output(["nmcli", "-t", "-f", "active,ssid",
"dev", "wifi"], text=True)
        for line in result.strip().split("\n"):
            if line.startswith("yes:"):
                return line.split(":")[1]
    except Exception as e:
        print(f"❌ Failed to get SSID: {e}")
    return None


# === Safety check before continuing ===
def validate_network():
    current_ssid = get_current_ssid()
    current_ip = get_current_ip(INTERFACE)

    print(f"📶 Connected SSID: {current_ssid}")
    print(f"🌐 IP Address on {INTERFACE}: {current_ip}")
```

```python
    if current_ssid != REQUIRED_SSID:
        print(f"❌ SSID mismatch! Expected: {REQUIRED_SSID}, Found: 
{current_ssid}")
        if current_ip != REQUIRED_IP:
            print(f"❌ IP mismatch! Expected: {REQUIRED_IP}, Found: 
{current_ip}")
        sys.exit(1)


    if current_ip != REQUIRED_IP:
        print(f"❌ IP mismatch! Expected: {REQUIRED_IP}, Found: 
{current_ip}")
        sys.exit(1)
```

[enableMQTT.py]

```python
import os
import sys
import subprocess
import netifaces



from config import (
    REQUIRED_IP,
    REQUIRED_SSID,
    INTERFACE
)
```

```python
# === Get current IP address ===
def get_current_ip(interface):
    try:
        ip = netifaces.ifaddresses(interface)[netifaces.AF_INET][0]['addr']
        return ip
    except Exception as e:
        print(f"❌ Failed to get IP address on {interface}: {e}")
        return None


# === Get current Wi-Fi SSID ===
def get_current_ssid():
    try:
        result = subprocess.check_output(["nmcli", "-t", "-f", "active,ssid", "dev", "wifi"], text=True)
        for line in result.strip().split("\n"):
            if line.startswith("yes:"):
                return line.split(":")[1]
    except Exception as e:
        print(f"❌ Failed to get SSID: {e}")
    return None


# === Safety check before continuing ===
def validate_network():
    current_ssid = get_current_ssid()
    current_ip = get_current_ip(INTERFACE)
```

```
    print(f"📶 Connected SSID: {current_ssid}")
    print(f"🌐 IP Address on {INTERFACE}: {current_ip}")


    if current_ssid != REQUIRED_SSID:
        print(f"❌ SSID mismatch! Expected: {REQUIRED_SSID}, Found:
{current_ssid}")
        if current_ip != REQUIRED_IP:
            print(f"❌ IP mismatch! Expected: {REQUIRED_IP}, Found:
{current_ip}")
        sys.exit(1)


    if current_ip != REQUIRED_IP:
        print(f"❌ IP mismatch! Expected: {REQUIRED_IP}, Found:
{current_ip}")
        sys.exit(1)
```

Laptop_pub.py

```
import time
from config import (
    MQTT_TOPIC_PUB_BLACK,
    MQTT_TOPIC_PUB_RED,
    MQTT_TOPIC_PUB_BLUE
)
import config
firstTime = True
```

```
blueCar_prevState = False

redCar_prevState = False

blackCar_prevState = False


def on_publish(client, userdata, mid):

    global firstTime

    global blueCar_prevState, redCar_prevState, blackCar_prevState

    if firstTime:

        firstTime = False

        blueCar_prevState = config.isBlueCar_live

        redCar_prevState = config.isRedCar_live

        blackCar_prevState = config.isBlackCar_live

        #print_status()

    else:

        check_status()


    #print("message published")

    #pass



def publish_message(client):

    while True:

        if not config.isMQTTEnabled:

            pubMsg("STOP", MQTT_TOPIC_PUB_BLUE, client)

            pubMsg("STOP", MQTT_TOPIC_PUB_RED, client)

            pubMsg("STOP", MQTT_TOPIC_PUB_BLACK, client)

        else:

            if is_all_autonomous():
```

```
            winner = get_lowest_data_car()
            if winner == "blue":
                pubMsg("GO", MQTT_TOPIC_PUB_BLUE, client)
                pubMsg("STOP", MQTT_TOPIC_PUB_RED, client)
                pubMsg("STOP", MQTT_TOPIC_PUB_BLACK, client)
            elif winner == "red":
                pubMsg("STOP", MQTT_TOPIC_PUB_BLUE, client)
                pubMsg("GO", MQTT_TOPIC_PUB_RED, client)
                pubMsg("STOP", MQTT_TOPIC_PUB_BLACK, client)
            else:
                pubMsg("STOP", MQTT_TOPIC_PUB_BLUE, client)
                pubMsg("STOP", MQTT_TOPIC_PUB_RED, client)
                pubMsg("GO", MQTT_TOPIC_PUB_BLACK, client)

        elif is_two_autonomous():
            if config.isBlueCarAutonomous and config.isRedCarAutonomous:
                if config.blueCar_data <= config.redCar_data:
                    pubMsg("GO", MQTT_TOPIC_PUB_BLUE, client)
                    pubMsg("STOP", MQTT_TOPIC_PUB_RED, client)
                else:
                    pubMsg("STOP", MQTT_TOPIC_PUB_BLUE, client)
                    pubMsg("GO", MQTT_TOPIC_PUB_RED, client)

            elif config.isBlueCarAutonomous and
config.isBlackCarAutonomous:
                if config.blueCar_data <= config.blackCar_data:
                    pubMsg("GO", MQTT_TOPIC_PUB_BLUE, client)
                    pubMsg("STOP", MQTT_TOPIC_PUB_BLACK, client)
                else:
```

```
                        pubMsg("STOP", MQTT_TOPIC_PUB_BLUE, client)

                        pubMsg("GO", MQTT_TOPIC_PUB_BLACK, client)


                elif config.isRedCarAutonomous and
config.isBlackCarAutonomous:

                    if config.redCar_data <= config.blackCar_data:

                        pubMsg("GO", MQTT_TOPIC_PUB_RED, client)

                        pubMsg("STOP", MQTT_TOPIC_PUB_BLACK, client)

                    else:

                        pubMsg("STOP", MQTT_TOPIC_PUB_RED, client)

                        pubMsg("GO", MQTT_TOPIC_PUB_BLACK, client)


            else:

                if config.isBlueCarAutonomous:

                    pubMsg("GO", MQTT_TOPIC_PUB_BLUE, client)

                elif config.isRedCarAutonomous:

                    pubMsg("GO", MQTT_TOPIC_PUB_RED, client)

                elif config.isBlackCarAutonomous:

                    pubMsg("GO", MQTT_TOPIC_PUB_BLACK, client)


            if not config.isBlueCarAutonomous:

                pubMsg("Manual Mode", MQTT_TOPIC_PUB_BLUE, client)

            if not config.isRedCarAutonomous:

                pubMsg("Manual Mode", MQTT_TOPIC_PUB_RED, client)

            if not config.isBlackCarAutonomous:

                pubMsg("Manual Mode", MQTT_TOPIC_PUB_BLACK, client)


        time.sleep(0.5)
```

```
def pubMsg(msg, topic, client):
    pubMsg = client.publish(
            topic=topic,
            payload=msg.encode('utf-8'),
            qos=0,
        )
    pubMsg.wait_for_publish()


def check_status():
    global blueCar_prevState, redCar_prevState, blackCar_prevState
    changed = False
    if blueCar_prevState != config.isBlueCar_live:
        blueCar_prevState = config.isBlueCar_live
        changed = True
    if redCar_prevState != config.isRedCar_live:
        redCar_prevState = config.isRedCar_live
        changed = True
    if blackCar_prevState != config.isBlackCar_live:
        blackCar_prevState = config.isBlackCar_live
        changed = True
    if changed:
        print_status()
    if not config.isBlueCar_live:
        config.redCar_data = 9999
    if not config.isBlackCar_live:
        config.blackCar_data= 9999
```

```
    if not config.isRedCar_live:

        config.blueCar_data= 9999


def print_status():

    print()

    print ("Blue Car is live ✅" if config.isBlueCar_live else "Blue Car is
Dead 🛑")

    print ("Red Car is live ✅" if config.isRedCar_live else "Red Car is Dead
🛑")

    print ("Black Car is live ✅" if config.isBlackCar_live else "Black Car
is Dead 🛑")




def is_all_autonomous():

    return config.isBlueCarAutonomous and config.isRedCarAutonomous and
config.isBlackCarAutonomous


def is_two_autonomous():

    count = 0

    if config.isBlueCarAutonomous:

        count += 1

    if config.isRedCarAutonomous:

        count += 1

    if config.isBlackCarAutonomous:

        count += 1

    return count == 2


def get_lowest_data_car():

    if config.blueCar_data <= config.redCar_data and config.blueCar_data <=
config.blackCar_data:
```

```
        return "blue"

    elif config.redCar_data <= config.blueCar_data and config.redCar_data <=
config.blackCar_data:

        return "red"

    else:

        return "black"
```

Laptop_sub.py

```
import paho.mqtt.client as mqtt

import time

import threading


from config import (

    MQTT_TOPIC_SUB_BLUE,

    MQTT_TOPIC_SUB_RED,

    MQTT_TOPIC_SUB_BLACK,

    MQTT_TOPIC_SUB_BLUE_ROS,

    MQTT_TOPIC_SUB_RED_ROS,

    MQTT_TOPIC_SUB_BLACK_ROS,

    enable_topic

)


import config


is_connected = False
```

```
last_seen = {
    "blue": time.time(),
    "red": time.time(),
    "black": time.time()
}

last_ros_seen = {
    "blue": time.time(),
    "red": time.time(),
    "black": time.time()
}


car_status = {
    "blue": True,
    "red": True,
    "black": True
}

# === Background thread to check car timeouts ===
def monitor_car_status():
     while True:
        now = time.time()
        for color in ["blue", "red", "black"]:
            if now - last_seen[color] > 2:  # 2 seconds timeout
               car_status[color] = False
            else:
               car_status[color] = True
```

```
            if now - last_ros_seen[color] > 2:
                if color == "blue":
                    config.isBlueCarAutonomous = True
                elif color == "red":
                    config.isRedCarAutonomous = True
                elif color == "black":
                    config.isBlackCarAutonomous = True

        config.isBlueCar_live = car_status["blue"]
        config.isRedCar_live = car_status["red"]
        config.isBlackCar_live = car_status["black"]

        time.sleep(0.5)  # check every 500ms

# === Callback when connected ===
def on_connect(client, userdata, flags, reason_code, properties=None):
    global is_connected
    if reason_code == 0:
        print("✅ Connected to MQTT Broker!")
        is_connected = True
        client.subscribe(MQTT_TOPIC_SUB_BLUE)
        client.subscribe(MQTT_TOPIC_SUB_RED)
        client.subscribe(MQTT_TOPIC_SUB_BLACK)
        client.subscribe(MQTT_TOPIC_SUB_BLUE_ROS)
        client.subscribe(MQTT_TOPIC_SUB_RED_ROS)
        client.subscribe(MQTT_TOPIC_SUB_BLACK_ROS)
        client.subscribe(enable_topic)
```

```python
    else:
        print(f"❌ Failed to connect, return code {reason_code}")
        is_connected = False

# === Callback when a message is received ===
def on_message(client, userdata, msg):
    topic = msg.topic

    if topic == enable_topic:
        config.isMQTTEnabled = True

    if topic in [MQTT_TOPIC_SUB_BLUE_ROS, MQTT_TOPIC_SUB_RED_ROS,
MQTT_TOPIC_SUB_BLACK_ROS]:
        ros_takeOver(msg)
        return

    payload = msg.payload.decode('utf-8')

    try:
        value = int(payload)  # parse directly as int
    except ValueError:
        if not topic == enable_topic:
            print(f"⚠️ Invalid payload: '{payload}'")
        return

    if topic == MQTT_TOPIC_SUB_BLUE:
        if not config.isBlueCarAutonomous:
            return
```

```python
        config.blueCar_data = value
        last_seen["blue"] = time.time()

    elif topic == MQTT_TOPIC_SUB_RED:
        if not config.isRedCarAutonomous:
            return
        config.redCar_data = value
        last_seen["red"] = time.time()

    elif topic == MQTT_TOPIC_SUB_BLACK:
        if not config.isBlackCarAutonomous:
            return
        config.blackCar_data = value
        last_seen["black"] = time.time()


# === Start the monitoring thread ===
threading.Thread(target=monitor_car_status, daemon=True).start()

def ros_takeOver(msg):
    payload = msg.payload
    topic = msg.topic

    try:
        value = int(payload)
    except ValueError:
        print(f"⚠️ Invalid payload: '{payload}'")
        return
```

```python
if topic == MQTT_TOPIC_SUB_BLUE_ROS:
    config.blueCar_data = value
    config.isBlueCarAutonomous = False
    config.isBlackCarAutonomous = True
    config.isRedCarAutonomous = True
    last_ros_seen["blue"] = time.time()

elif topic == MQTT_TOPIC_SUB_RED_ROS:
    config.redCar_data = value
    config.isBlueCarAutonomous = True
    config.isBlackCarAutonomous = True
    config.isRedCarAutonomous = False
    last_ros_seen["red"] = time.time()

elif topic == MQTT_TOPIC_SUB_BLACK_ROS:
    config.blackCar_data = value
    config.isBlueCarAutonomous = True
    config.isBlackCarAutonomous = False
    config.isRedCarAutonomous = True
    last_ros_seen["black"] = time.time()
```

Mqtt_client.py

```python
import paho.mqtt.client as mqtt
```

```python
import warnings
from paho.mqtt.client import CallbackAPIVersion
print(CallbackAPIVersion)
from config import MQTT_BROKER, MQTT_PORT, MQTT_CLIENT_ID
from laptop_pub import publish_message, on_publish
from laptop_sub import on_connect, on_message
from checkWifi import validate_network

warnings.filterwarnings("ignore", category=DeprecationWarning)

# === Validate Network Connection ===
validate_network()

# === Setup MQTT Client ===
client = mqtt.Client(
    client_id=MQTT_CLIENT_ID,
    protocol=mqtt.MQTTv311,
)
client.on_connect = on_connect
client.on_message = on_message
client.on_publish = on_publish

client.connect(MQTT_BROKER, MQTT_PORT, 60)


# === Loop and Publish ===
client.loop_start()
```

```python
try:
    publish_message(client)

except KeyboardInterrupt:
    print("🛑 Exiting...")
    print("\nStopping MQTT client due to KeyboardInterrupt")
    client.loop_stop()
    client.disconnect()

except Exception as e:
    print("🛑 Exiting...")
    print("\nStopping MQTT client")
    print(e)

finally:
    client.loop_stop()
    client.disconnect()
```