Project Title

Smart ambulance with mqtt protocol

A project report submitted in partial fulfilment of requirements to

CENTER FOR DEVELOPMENT OF ADVANCED

COMPUTING

For the award of the degree of

POST GRADUATE DIPLOMA

In

EMBEDDED SYSTEM AND DESIGN

Under the esteemed guidance of

MRS. Sheran Evangelin

SUBMITTED BY

Yaswanth kuna	*230950330021*
Meghana nikambe	*230950330022*
Mrinal patil	*230950330023*
Smitha mule	*230950330024*

In PG-DIPLOMA IN EMBEDDED SYSTEM DESIGN

OFFERED BY C-DAC HYDERABAD

September 2023

ACKNOWLEDGMENT

TABLE OF CONTENTS

1. Abstract:

This project introduces the smart ambulance with mqtt protocol. This project mainly designed to enhance emergency medical services. Developed Health monitoring system for patient and Traffic management system for ambulance using RTOS. Here, we integrate ESP32 microcontroller with sensors like GPS module, MAX30102 heart rate sensor, temperature sensor, and 433MHz RF Tx & Rx module, delivering immediate health updates of patient and ambulance gps location. The system employs the Message Queuing Telemetry Transport (MQTT) protocol to facilitate efficient communication between the ambulance and the hospital. The use of MQTT ensures reliable and low-latency communication, allowing medical professionals to receive timely updates on the patient's condition. In Traffic management system, when the ambulance is near the traffic light, the traffic light changes from red to green which is intended to ambulance with the help of 433MHz Tx & Rx pair to avoid traffic. We incorporated these functionalities in this project for better enhancement of emergency medical services.

2. List of Components:

2.1. ESP32 -DevKitC v4 withESP32-WROOM-32:
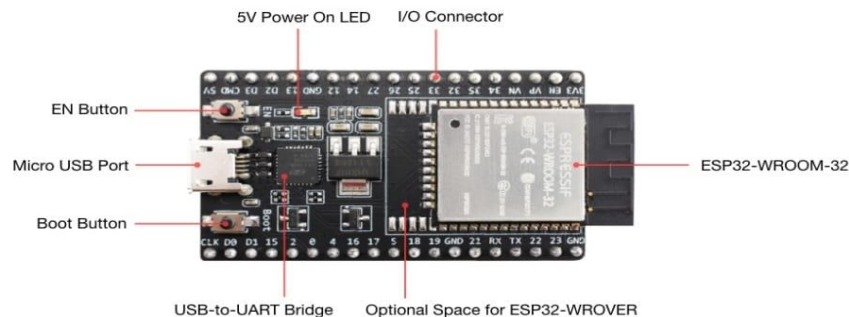
ESP32-DevKitC V4 with ESP32-WROOM-32 module soldered

Key Component	Description
ESP32-WROOM-32	A module with ESP32 at its core. For more information, see ESP32-WROOM-32 Datasheet.
EN	Reset button.
Boot	Download button. Holding down **Boot** and then pressing **EN** initiates Firmware Download mode for downloading firmware through the serial port.
USB-to-UART Bridge	Single USB-UART bridge chip provides transfer rates of up to 3 Mbps.
Micro USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the ESP32-WROOM-32 module.
5V Power On LED	Turns on when the USB or an external 5V power supply is connected to the board. For details see the schematics in Related Documents.
I/O	Most of the pins on the ESP module are broken out to the pin headers on the board. You can program ESP32 to enable multiple functions such as PWM, ADC, DAC, I2C, I2S, SPI, etc.

2.1.1.   Header Block⌕

The two tables below provide the **Name** and **Function** of I/O header pins on both sides of the board, as shown in ESP32-DevKitC V4 with ESP32-WROOM-32 module soldered.

2.1.2.   J2⌕

No.	Name	Type 1	Function
1	3V3	P	3.3 V power supply
2	EN	I	CHIP_PU, Reset
3	VP	I	GPIO36, ADC1_CH0, S_VP
4	VN	I	GPIO39, ADC1_CH3, S_VN
5	IO34	I	GPIO34, ADC1_CH6, VDET_1
6	IO35	I	GPIO35, ADC1_CH7, VDET_2
7	IO32	I/O	GPIO32, ADC1_CH4, TOUCH_CH9, XTAL_32K_P
8	IO33	I/O	GPIO33, ADC1_CH5, TOUCH_CH8, XTAL_32K_N
9	IO25	I/O	GPIO25, ADC1_CH8, DAC_1
10	IO26	I/O	GPIO26, ADC2_CH9, DAC_2
11	IO27	I/O	GPIO27, ADC2_CH7, TOUCH_CH7
12	IO14	I/O	GPIO14, ADC2_CH6, TOUCH_CH6, MTMS
13	IO12	I/O	GPIO12, ADC2_CH5, TOUCH_CH5, MTDI
14	GND	G	Ground
15	IO13	I/O	GPIO13, ADC2_CH4, TOUCH_CH4, MTCK
16	D2	I/O	GPIO9, D2 2
17	D3	I/O	GPIO10, D3 2
18	CMD	I/O	GPIO11, CMD 2
19	5V	P	5 V power supply

2.1.3.   J3

No.	Name	Type [1]	Function
1	GND	G	Ground
2	IO23	I/O	GPIO23
3	IO22	I/O	GPIO22
4	TX	I/O	GPIO1, U0TXD
5	RX	I/O	GPIO3, U0RXD
6	IO21	I/O	GPIO21
7	GND	G	Ground
8	IO19	I/O	GPIO19
9	IO18	I/O	GPIO18
10	IO5	I/O	GPIO5
11	IO17	I/O	GPIO17 [3]
12	IO16	I/O	GPIO16 [3]
13	IO4	I/O	GPIO4, ADC2_CH0, TOUCH_CH0
14	IO0	I/O	GPIO0, ADC2_CH1, TOUCH_CH1, Boot
15	IO2	I/O	GPIO2, ADC2_CH2, TOUCH_CH2
16	IO15	I/O	GPIO15, ADC2_CH3, TOUCH_CH3, MTDO
17	D1	I/O	GPIO8, D1 [2]
18	D0	I/O	GPIO7, D0 [2]
19	CLK	I/O	GPIO6, CLK [2]

1([1],[2])

P: Power supply; I: Input; O: Output.

2([1],[2],[3],[4],[5],[6])

The pins D0, D1, D2, D3, CMD and CLK are used internally for communication between ESP32 and SPI flash memory. They are grouped on both sides near the USB connector. Avoid using these pins, as it may disrupt access to the SPI flash memory/SPI RAM.

The pins GPIO16 and GPIO17 are available for use only on the boards with the modules ESP32-WROOM and ESP32-SOLO-1. The boards with ESP32-WROVER modules have the pins reserved for internal use.

2.1.4.   Pin Layout
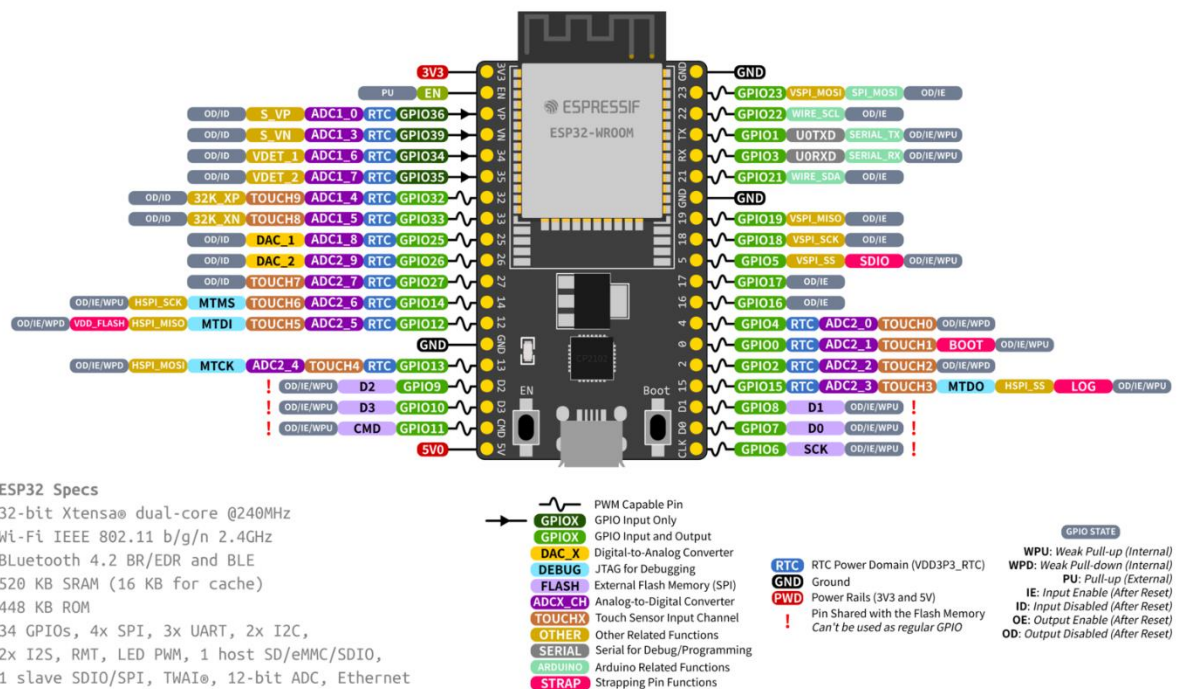
Features:
ESP32-WROOM-32E Module (2)
Wireless Connectivity
Dual-Core Processor
Sufficient Memory
Peripheral Support
USB-UART Bridge
Integrated Antenna

Specifications:
Dual-Core Processor: dual-core Tensilica LX6 microcontroller
Wireless Connectivity: 2.4 GHz Wi-Fi 802.11 b/g/n and Bluetooth 4.2
Rich Peripheral Set: GPIO pins, analog-to-digital converters (ADCs),

Secure Boot and Flash Encryption: The ESP32-WROOM-3 2E includes secure boot and flash encryption features to protect your firmware and data.

Built-In Antenna and External Antenna Option: The board comes with an integrated antenna for both Wi-Fi and Bluetooth
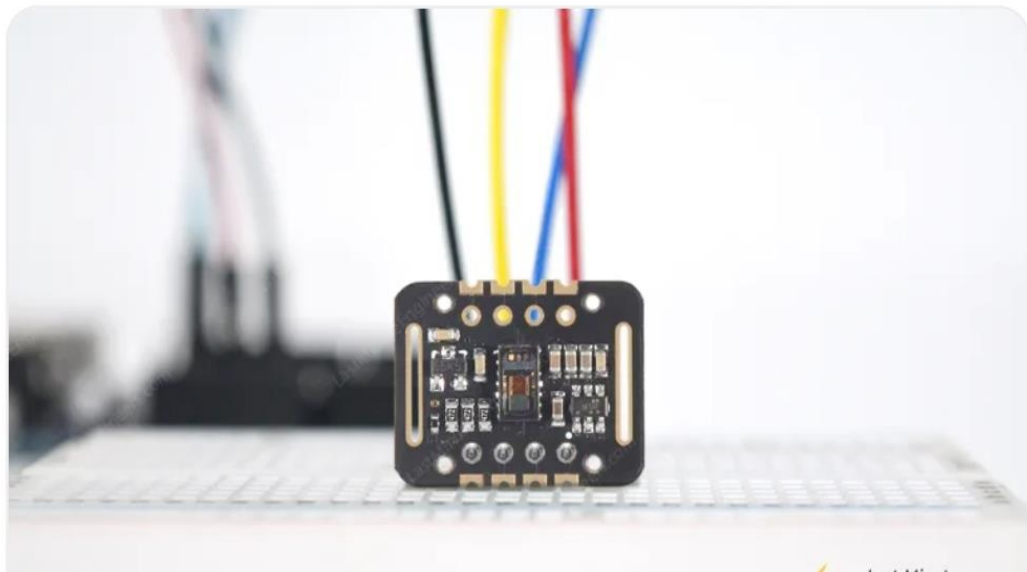
Onboard Sensors: onboard sensors like a temperature and humidity sensor (DHT22) and a digital microphone (SPM1423)

USB-to-UART Bridge: communication interfaces, including UART, SPI, I2C, and more

Referred this for ESP32:
https://docs.espressif.com/projects/esp-idf/en/stable/esp32/hw-reference/esp32/get-started-devkitc.html

2.2. MAX30102 heart rate sensor

Benefits and Features
● Heart-Rate Monitor and Pulse Oximeter Sensor in LED Reflective Solution
● Tiny 5.6mm x 3.3mm x 1.55mm 14-Pin Optical Module
• Integrated Cover Glass for Optimal, Robust
Performance
● Ultra-Low Power Operation for Mobile Devices
• Programmable Sample Rate and LED Current for
Power Savings
• Low-Power Heart-Rate Monitor (< 1mW)
• Ultra-Low Shutdown Current (0.7μA)
● Fast Data Output Capability
• High Sample Rates
● Robust Motion Artifact Resilience
• High SNR
● -40°C to +85°C Operating Temperature Range

References:
Datasheet – max30102.pdf
https://lastminuteengineers.com/max30102-pulse-oximeter-heart-rate-sensor-arduino-tutorial/

2.3. LM75 temperature sensor module

The LM75A or B is a high-speed I2C interface temperature sensor that converts temperature directly to digital signals from -55°C to +125°C and achieves an accuracy of 0.125°C. The MCU can read directly from the I2C bus.

The data in the internal registers and the four data registers can be operated by I2C to set different operating modes.

The LM75A has three optional logical address pins, so that 8 devices can be connected simultaneously on the same bus without An address conflict has occurred.

The device can be used as a stand-alone temperature controller at power-up;
Separate Open-Drain OS Output Operates as Interrupt or Comparator/Thermostat Input
Register Readback Capability
Power-Up Defaults Permit Stand-Alone Operation as a Thermostat
Low Operating Supply Current 250µA (typ), 1mA (max)
4µA (typ) Shutdown Mode Minimizes Power Consumption.
This temperature sensor has been specifically designed to be used with the Raspberry Pi, but it can also be used with the Arduino and esp32. The module features Maxim's LM75 Digital Temperature Sensor!

Features:-
Range in -25°C~100°C
Small Profile
Works with both 3.3V and 5V
Supported on Raspberry Pi and Arduino
LM75 IC
I2C port

2.4. NEO-6M GPS Module

The NEO-6M GPS module has five major parts on the board, the first major part is the **NEO-6M GPS chip** in the heart of the PCB. Next, we have a **rechargeable battery and a serial EEPROM module**. An EEPROM together with a battery helps retain the clock data, latest position data (GNSS orbit data), and module configuration but it's not meant for permanent data storage. Without the battery, the GPS always cold-starts so the initial GPS lock takes more time. The battery is automatically charged when power is applied and maintains data for up to two weeks without power. Next, we have our **LDO**, because of the onboard LDO, the module can be powered from a 5V supply. Finally, we have our **UFL connector** where we need to connect an external antenna for the GPS to properly work.

Features:
5Hz position update rate
Operating temperature range: -40 TO 85°CUART TTL socket
EEPROM to save configuration settings
Rechargeable battery for Backup
The cold start time of 38 s and Hot start time of 1 s
Supply voltage: 3.3 V
Configurable from 4800 Baud to 115200 Baud rates. (default 9600)
SuperSense Indoor GPS: -162 dBm tracking sensitivity
Support SBAS (WAAS, EGNOS, MSAS, GAGAN)
Separated 18X18mm GPS antenna

 NEO-6M GPS MODULE PINOUT The NEO-6M GPS module has four pins: GND, TxD, RxD, and VCC. The TxD and RxD pins are used to communicate with the microcontroller.
GND is the ground pin of the GPS Module and it should be connected to the ground pin of the ESP32. TXD is the transmit pin of the GPS module that needs to connect to the RX pin of the ESP32. RXD is the receive pin of the GPS module that needs to connect to the TX pin of the ESP32. VCC is the power pin of the GPS module and needs to connect to the 3.3V pin of the ESP32.

What will GPS Module do

This is a complete GPS module that is based on the NEO-6M. This unit uses the latest technology to give the best possible positioning information and includes a larger built-in 25 x 25mm active GPS antenna with a UART TTL socket. GPS module that allows them to know their location relative to a network of orbiting satellites

2.5. 433 MHz RF Module

Why 433MHz-

There are several reasons why one might choose to use a 433 MHz RF module over other frequency bands or wireless communication technologies:

1. Availability and Regulations: The 433 MHz band is widely available for unlicensed use in many countries, making it easy to use without needing to obtain specific licenses. Regulations regarding power output and usage vary by region but generally allow for relatively low-power devices without complex licensing requirements.

2. Long Range: The lower frequency of 433 MHz allows for better penetration through obstacles like walls and buildings compared to higher frequency bands. This makes 433 MHz modules suitable for applications requiring longer range communication

Specifications of 433MHz RF Transmitter Receiver Wireless Module:-
Range in open space(Standard Conditions) : 100 Meters
RX Receiver Frequency : 433 MHz
RX Typical Sensitivity : 105 Dbm
RX Supply Current : 3.5 mA
RX IF Frequency : 1MHz
RX Operating Voltage : 5V
TX Frequency Range : 433.92 MHz
TX Supply Voltage : 3V ~ 6V
TX Out Put Power : 4 ~ 12 Dbm
 Features of 433MHz RF Transmitter Receiver Wireless Module:-
Low Power Consumption
Easy For RF based Application
Complete Radio Transmitter
Transmit Range Up To 50m
CMOS / TTL Input
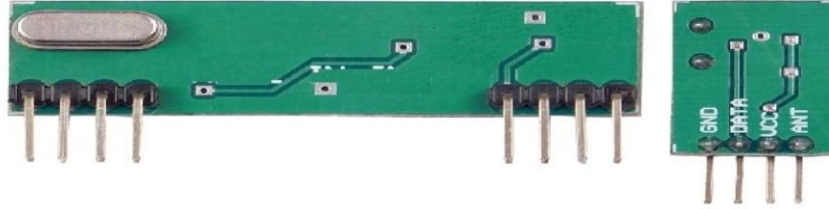No Adjustable Components
Very Stable Operating Frequency
Low Current Consumption (Typ 11mA)
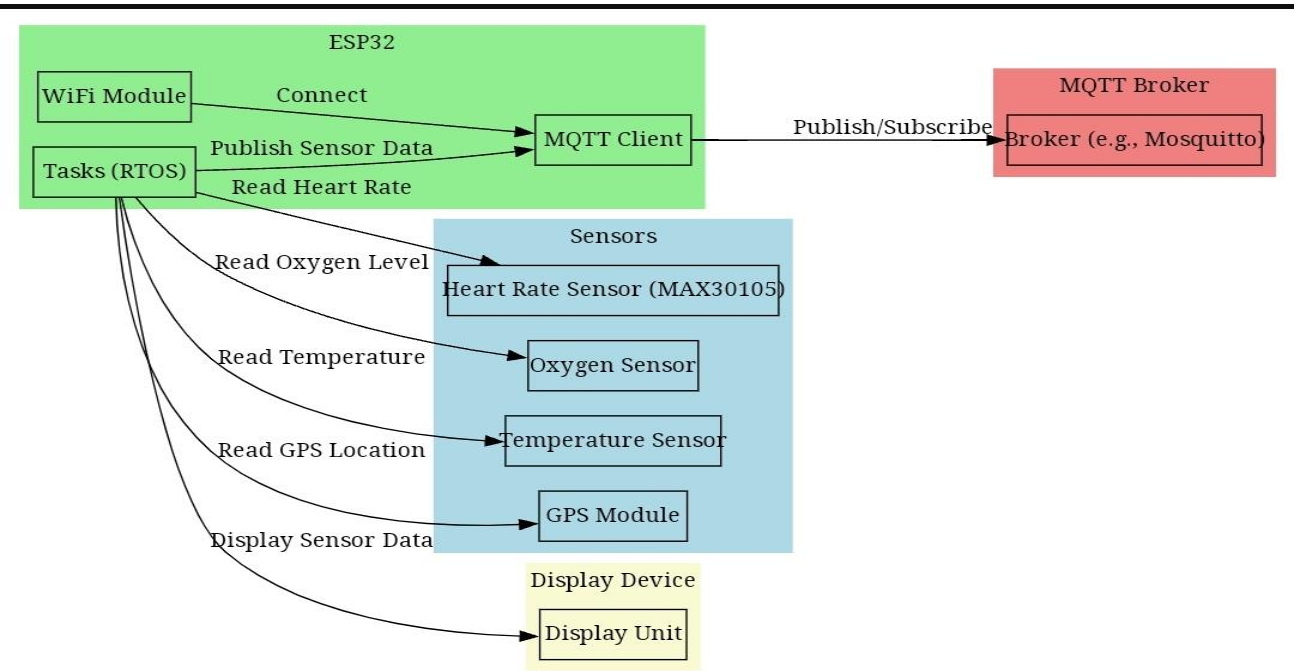Wide Operating Voltage
ASK Modulation

433MHz RF Module

What will Tx Rx module do

In the traffic management system, traffic light changes from red to green when ambulance approaches it , using 433MHz Tx & Rx modules. The Traffic Management System triggers traffic light changes, aiding ambulance navigation to avoid congestion. These advancements aim to improve emergency medical services effectively. In Traffic management system, when the ambulance is near the traffic light, the traffic light changes from red to green to avoid traffic.

3. Block Diagram:

ESP32

WiFi Module

Connect

Tasks (RTOS)

Publish Sensor Data

MQTT Client

Read Heart Rate

Read Oxygen Level

Read Temperature

Read GPS Location

Display Sensor Data

Publish/Subscribe

MQTT Broker

Broker (e.g., Mosquitto)

Sensors

Heart Rate Sensor (MAX30105)

Oxygen Sensor

Temperature Sensor

GPS Module

Display Device

Display Unit

4. Interfacing and connections:

In this project we had integrated a 433MHz RF transmitter and receiver, LM75 temperature sensor, MAX30102 pulse oximeter for measuring oxygen and heart rate, Neo 6M GPS module, and ESP32 microcontroller. This setup enables real-time monitoring of vital signs and location tracking, making it ideal for applications such as remote health monitoring or outdoor activities. We used 433 MHz RF Tx & Rx for traffic management.

1) LM75 temperature sensor module
   LM75 has 5 pins. VCC, GND, SDA, SCL, OS.
   VCC- VCC of ESP32
   SDA – Pin 21 of Esp32
   SCL- Pin 22 of ESP32
   OS- The open-drain overtemperature output (OS) sinks current when the programmable temperature limit is exceeded. The OS output operates in either of two modes, comparator or interrupt.

2) GPS Module:
   Connect the VCC pin on the GPS to the 3.3V pin on the ESP32.
   Connect the GND pin on the GPS to the GND pin on the ESP32.
   Connect the Tx pin on the GPS to the Rx pin on the ESP32.
   Connect the Rx pin on the GPS to the Tx pin on the ESP32.
   (Works with UART Protocol)

3) MAX 30102 pulse oximetry and heart-rate monitor module

MAX30102 Module	ESP32
VCC	3.3V
SCL	GPIO22
SDA	GPIO21
GND	GND

4) 433 MHz RF Module
   Transmitter Section Wiring
   The wiring for the transmitter is simple. It has only three connections. Connect the VCC pin to 5V pin and GND to ground on the ESP. The Data-In pin should be connected to ESP32 digital pin 4. Leave antenna as it is. (1st ESP)

   Receiver Section Wiring
   The wiring for the receiver is just as easy as the transmitter. Once again there are only three connections to make. Connect the VCC pin to 5V pin and GND to ground on the ESP. Any one of the middle two Data-Out pins should be connected to digital pin 5.
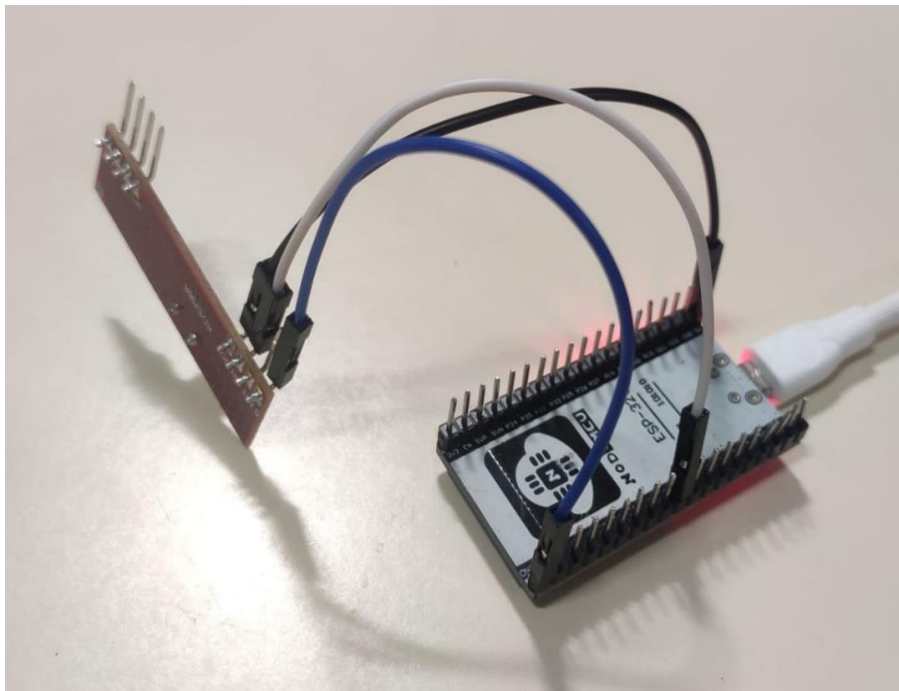   (2nd ESP)

4.1. Connections of 433 MHz RF Tx & Rx with esp 32:

The Tx is placed in ambulance connected with esp32 and Rx is placed in Traffic light connected with esp32. When the Ambulance is approaching Rx in ESP32
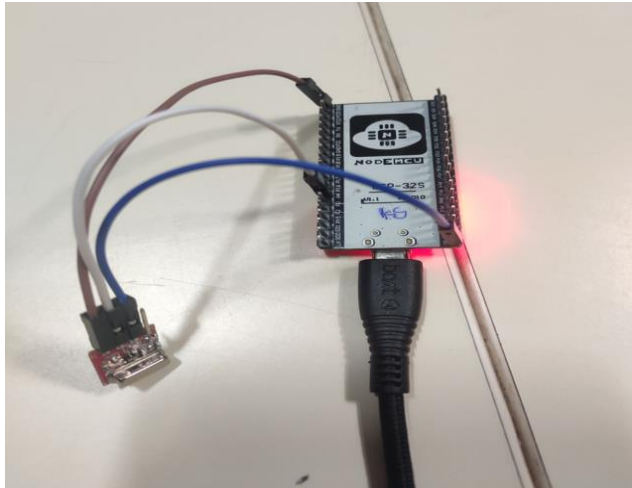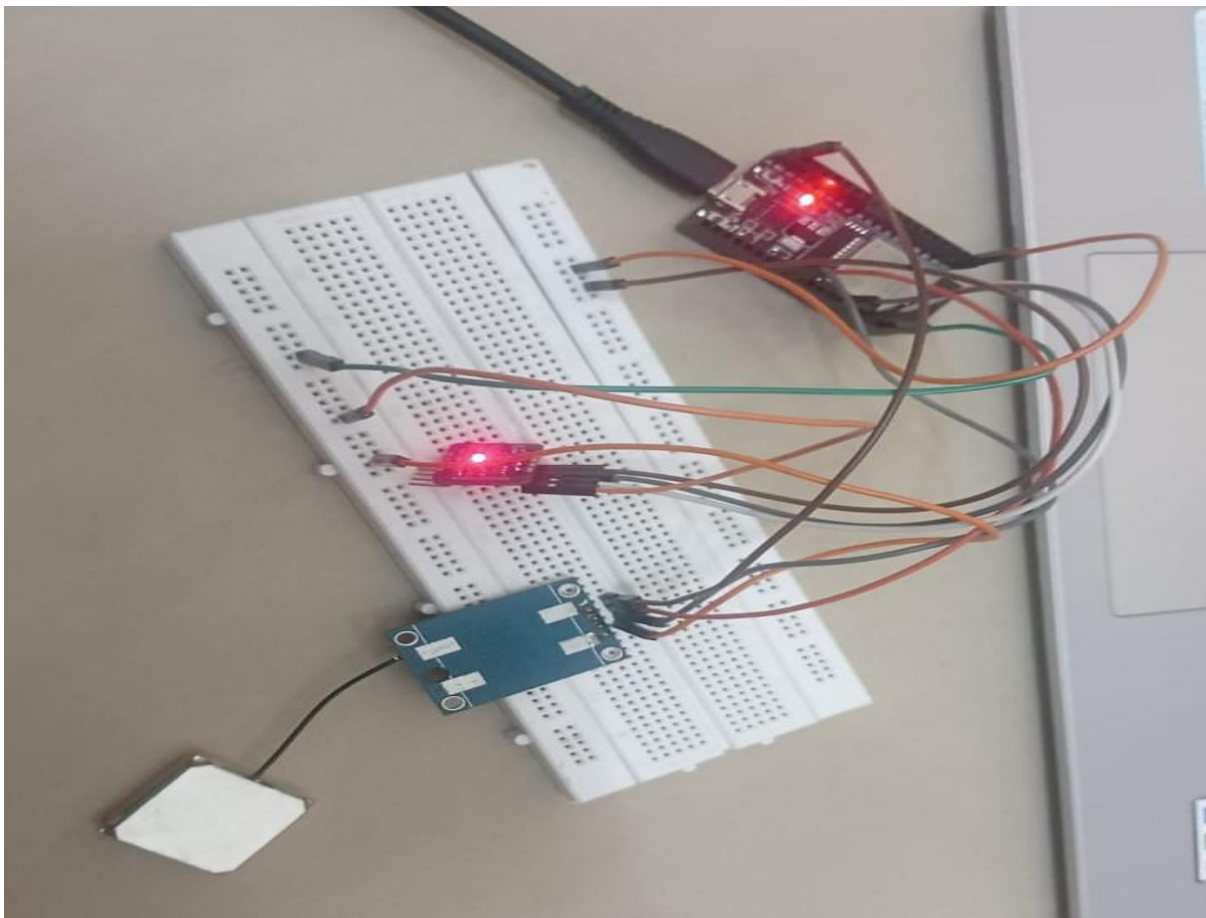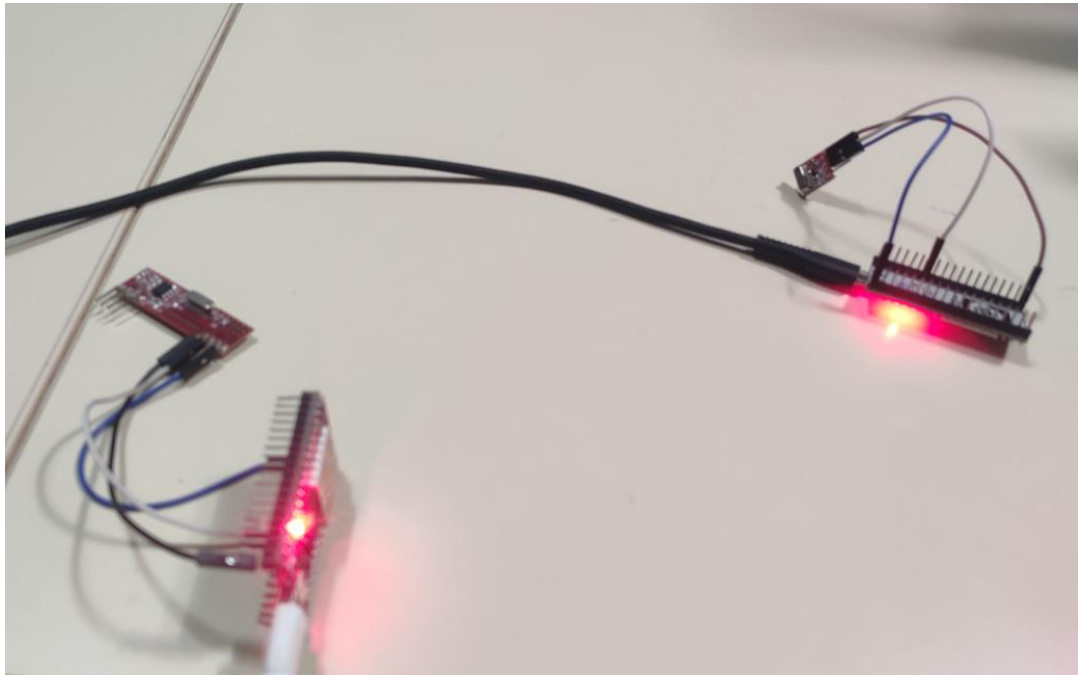

433 MHz RF Tx

433 MHz RF Rx

433 MHz Tx & Rx communication

5. MQTT Protocol:

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe messaging protocol often used in IoT (Internet of Things) and mobile applications. It operates on the client-server or broker-based model.

- **Publish-Subscribe Model:** Devices (clients) communicate through a central broker. Publishers send messages to specific topics, and subscribers receive messages from topics they are interested in.
- **QoS (Quality of Service) Levels:** MQTT supports three QoS levels for message delivery: 0 (at most once), 1 (at least once), and 2 (exactly once), allowing flexibility based on the application's reliability requirements.
- **Retained Messages:** Brokers can retain the last message sent on a specific topic. When a new subscriber joins, it immediately receives the most recent message for each topic.
- **Lightweight:** Designed to be efficient, MQTT has a minimal overhead, making it suitable for resource-constrained devices and low-bandwidth networks.
- **Persistent Sessions:** Clients can establish persistent sessions with the broker, allowing them to receive messages that were sent while they were offline.
- **Security:** While MQTT itself does not prescribe security mechanisms, it can be implemented over secure protocols like TLS/SSL, ensuring data confidentiality and integrity.

Overall, MQTT's simplicity and efficiency make it a popular choice for real-time communication in scenarios where lightweight and low-latency messaging is crucial.
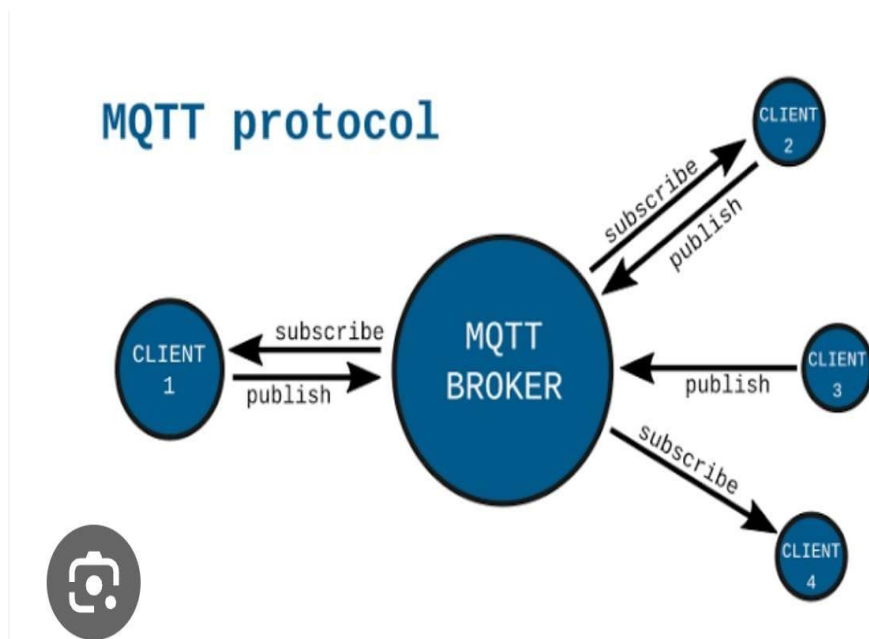
Why MQTT Protocol:

1. **Low Overhead and Bandwidth Efficiency:** MQTT is designed to be lightweight, with a minimal protocol overhead. This is crucial for IoT devices, which often have limited processing power, memory, and bandwidth. The protocol's efficiency allows for effective communication in resource-constrained environments.
2. **Publish-Subscribe Model:** The publish-subscribe model of MQTT is well-suited for IoT applications where multiple devices need to communicate with each other. This model allows for scalable and flexible communication, as devices can publish messages to specific topics, and other devices interested in those topics can subscribe to receive the messages.
3. **Asynchronous Communication:** MQTT enables asynchronous communication, allowing devices to send and receive messages independently. This is beneficial for IoT scenarios where devices may not always be online simultaneously, and they need to exchange information without waiting for real-time responses.
4. **Quality of Service (QoS) Levels:** MQTT supports different levels of message delivery assurance (QoS levels), allowing IoT applications to choose the appropriate level of reliability for their communication. This flexibility is important when dealing with unreliable networks or varying latency requirements.
5. **Retained Messages:** The concept of retained messages in MQTT ensures that devices receive the most recent information when they subscribe to a topic. This is particularly useful in IoT scenarios where devices might join or leave the network intermittently.
6. **Scalability:** MQTT can scale to accommodate a large number of devices. The central broker facilitates communication between devices without each device needing to be aware of the others, making it a scalable solution for IoT deployments.
7. **Security Integration:** While MQTT itself doesn't dictate security mechanisms, it can be implemented over secure transport layers (e.g., TLS/SSL), providing encryption and authentication. This is crucial for maintaining the confidentiality and integrity of data exchanged between IoT devices.
8. **Open Standard:** MQTT is an open standard, fostering interoperability among various IoT devices and platforms. This openness contributes to its widespread adoption in the IoT ecosystem.

In summary, MQTT's lightweight nature, publish-subscribe model, asynchronous communication, and support for different QoS levels make it a well-suited protocol for the communication needs of IoT devices.

6. Flowchart:
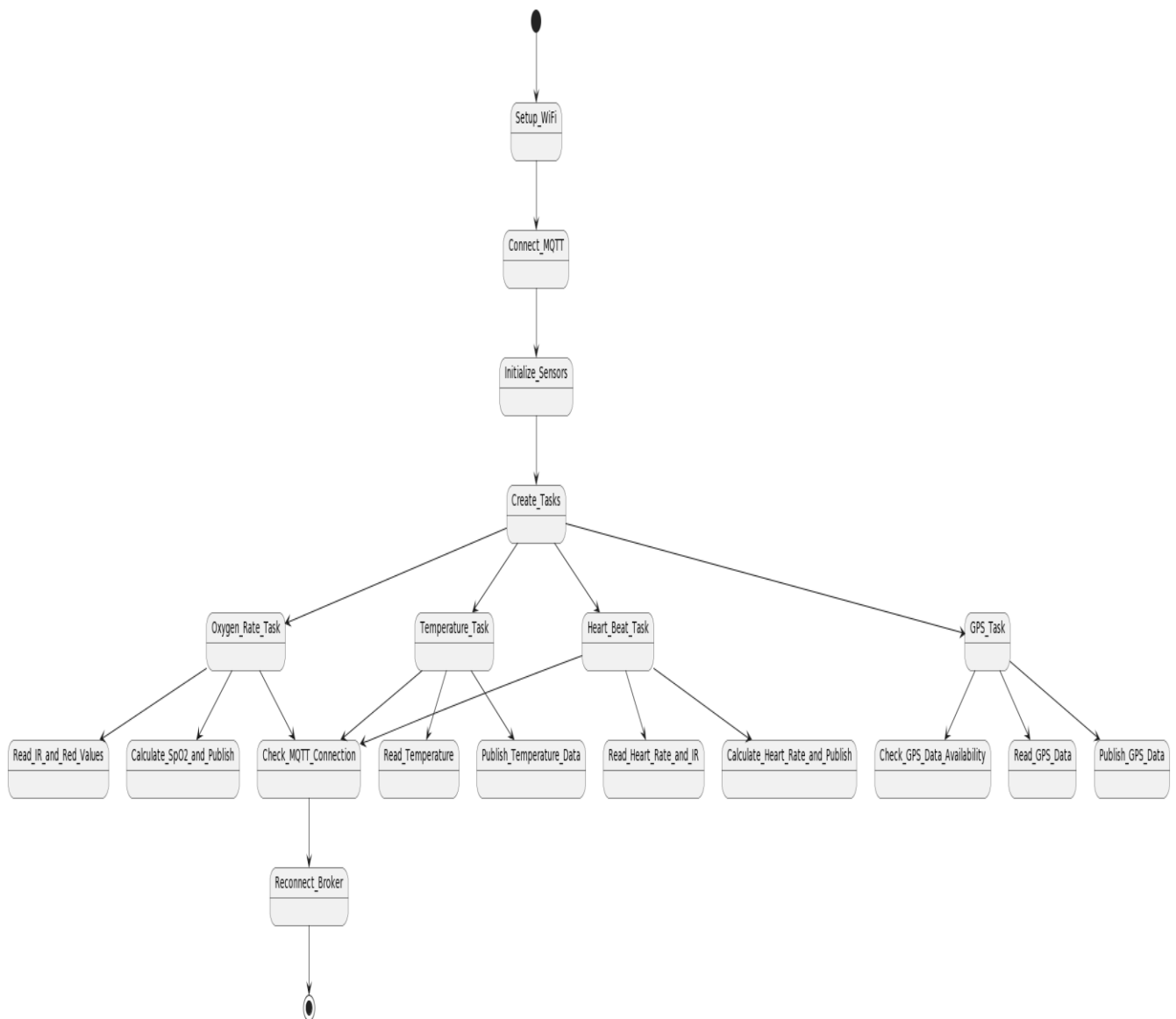
The flowchart outlines the sequence of operations for a smart ambulance project using MQTT. It begins with setting up the WiFi connection and connecting to the MQTT broker. Then, it initializes sensors such as the heart rate sensor, oxygen rate sensor, temperature sensor, and GPS module. Tasks are created for each sensor to monitor their respective data. These tasks periodically check the MQTT connection status and reconnect if necessary. The heart rate task reads and calculates heart rate data, the oxygen rate task reads and calculates SpO2 data, the temperature task reads temperature data, and the GPS task reads GPS data. Finally, all sensor data is published via MQTT for further processing or monitoring.

7. Application Code:

The code is designed for an ESP32-based IoT device that incorporates various sensors to monitor health-related parameters, including heart rate, oxygen saturation, temperature, and GPS location. The device utilizes the MQTT protocol for communication, allowing it to send real-time data to a designated MQTT broker.

Flow of the Code:

1. Import Libraries:

   - WiFi.h, PubSubClient.h: For connecting to the Wi-Fi network and implementing MQTT communication.

   - Wire.h: Required for I2C communication.

   - MAX30105.h, heartRate.h: Libraries for interfacing with the MAX30105 pulse oximeter and heart-rate sensor.

   - TinyGPS++.h: Library for parsing GPS data.

2. Define Constants:

   - Define Wi-Fi credentials, MQTT server details, and other constants like pins, baud rate, and buffer sizes.

3. Initialize Wi-Fi:

   - Connect to the specified Wi-Fi network.

4. MQTT Connection:

   - Connect to the MQTT broker with the provided credentials.

   - Publish initial messages upon successful connection.

5. Sensor Initialization:

   - Initialize the MAX30105 sensor for heart rate and oxygen saturation measurements.

   - Set up additional components such as the buzzer and brightness control.

6. Task Creation:

   - Create tasks for monitoring heart rate, oxygen saturation, temperature, and GPS location.

- These tasks run concurrently on a separate core (core 1) to ensure real-time data collection without blocking the main loop.

7. Heart Rate Task:

  - Continuously monitor heart rate using the MAX30105 sensor.

  - Calculate beats per minute (BPM) and average BPM over a specified period.

  - Publish heart rate and average heart rate to the MQTT broker.

  - Trigger a buzzer alarm if the heart rate drops below a predefined threshold.

8. Oxygen Saturation Task:

  - Measure oxygen saturation based on the red and infrared signals from the MAX30105 sensor.

  - Calculate SpO2 using a predefined calibration curve.

  - Publish SpO2 level to the MQTT broker.

  - Activate a visual and audible alarm if SpO2 falls below a specified threshold.

9. Temperature Task:

  - Read temperature data from a sensor (assumed to be LM75).

  - Publish temperature readings to the MQTT broker.

10. GPS Task:

  - Read GPS data from a NEO-6M GPS module using TinyGPS++.

  - Publish latitude, longitude, altitude, speed, and date/time to the MQTT broker.

11. Main Loop:

  - The main loop is intentionally left empty as most of the functionality is implemented in tasks running on a separate core.

This ESP32-based IoT device integrates health monitoring capabilities using the MAX30105 sensor for heart rate and oxygen saturation, a temperature sensor, and a GPS module. The MQTT protocol facilitates seamless communication with a designated MQTT broker, allowing real-time data transmission. The code organizes tasks on separate cores to ensure concurrent and non-blocking execution of health monitoring processes. Additionally, visual and audible alarms are incorporated to notify users of critical health parameter thresholds. The modular design and use of industry-standard protocols make it adaptable for various IoT health monitoring applications.

Application Code:

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include "MAX30105.h"
#include "heartRate.h"
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <TinyGPS++.h>


// Update these with values suitable for your network and MQTT broker
const char* ssid = "House64_Wifi";
const char* password = "House_6401";
const char* mqtt_server = "192.168.0.117";
const char* mqtt_user = "yesh";
const char* mqtt_password = "yesh@123";

#define bright 23
const int buzzer = 2;

#define MSG_BUFFER_SIZE (50)
#define GPS_BAUDRATE 9600   // The default baudrate of NEO-6M is 9600
char msg[MSG_BUFFER_SIZE];  //string to publish the data
char bpm[MSG_BUFFER_SIZE];
char avg_bpm[MSG_BUFFER_SIZE];
char oxy_spo2[MSG_BUFFER_SIZE];
char temp[MSG_BUFFER_SIZE];
char lat[MSG_BUFFER_SIZE];
char lon[MSG_BUFFER_SIZE];
```

```cpp
const byte RATE_SIZE = 4;  //Increase this for more averaging. 4 is good.

byte rates[RATE_SIZE];    //Array of heart rates

byte rateSpot = 0;

long lastBeat = 0;    //Time at which the last beat occurred

float beatsPerMinute;  //heart rate

int beatAvg;        //average of four heartrates

// const int buzzer = 2;

#define core 1


WiFiClient espClient;

PubSubClient client(espClient);

MAX30105 particleSensor;

TinyGPSPlus gps;  // the TinyGPS++ object



//   We start by connecting to a WiFi network

void setup_wifi() {
  delay(10);

  Serial.println();

  Serial.print("Connecting to ");

  Serial.println(ssid);

  WiFi.mode(WIFI_STA);

  WiFi.begin(ssid, password);


  while (WiFi.status() != WL_CONNECTED) {
    delay(500);

    Serial.print(".");
  }


  Serial.println("");

  Serial.println("WiFi connected");
```

```cpp
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}




//  to subscribe to mqtt broker
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
  // if ((char)payload[0] == '1') {
  //  digitalWrite(BUILTIN_LED, LOW);  // Turn the LED on (Note that LOW is the voltage level
  //  // but actually the LED is on; this is because
  //  // it is active low on the ESP-01)
  // } else {
  //  digitalWrite(BUILTIN_LED, HIGH);  // Turn the LED off by making the voltage HIGH
  // }
}




void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
   Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("CL_ID_1", mqtt_user, mqtt_password)) {
      // to establishing connection between esp32 and mqtt broker
```

```
    Serial.println("connected");

    // Once connected, publish an announcement...

    client.publish("irValue", "hello world");

    snprintf(bpm, MSG_BUFFER_SIZE, "%f", beatsPerMinute);

    client.publish("heart rate", bpm);

    // ... and resubscribe

    client.subscribe("inTopic");

  } else {

    Serial.print("failed, rc=");

    Serial.print(client.state());

    Serial.println(" try again in 5 seconds");

    // Wait 5 seconds before retrying

    delay(5000);

  }

 }

}

float readTemperature() {

 Wire.beginTransmission(LM75_ADDRESS);  // Start communication with LM75 at specified address

 Wire.write(0x00);  // Select LM75 register for temperature reading

 Wire.endTransmission();  // End the transmission


 delay(100);  // Wait for the LM75 to complete the temperature conversion


 Wire.requestFrom(LM75_ADDRESS, 2);  // Request 2 bytes of data from LM75

 while (Wire.available() < 2);  // Wait until data is available


  int16_t rawTemperature = Wire.read() << 8 | Wire.read();  // Combine two bytes into a 16-bit raw
temperature value

 float temperature = rawTemperature / 256.0;  // Convert raw temperature to Celsius

  // The LM75 has a resolution of 0.5°C per bit, and dividing by 256 gives the temperature in degrees
Celsius.
```

```
  return temperature;  // Return the temperature value

}




TaskHandle_t heartBeatTaskHandle, oxygenRateTaskHandle, temperatureTaskHandle,
gpsTaskHandle;


void heartBeatTask(void* data) {
 while (1) {
  if (!client.connected()) {  //mqtt client not currently connected
   reconnect();          //reconnect broker
  }
  client.loop();




   long irValue = particleSensor.getIR();
   uint32_t redValue = particleSensor.getRed();


  if (checkForBeat(irValue) == true)  // Perform analysis on the IR signal to detect a beat // compare
the current IR value with a threshold // If the signal crosses the threshold, consider it a beat
  {
    //We sensed a beat!
    long delta = millis() - lastBeat;  //delta - difference between two heart beats. current time - last
time of heart beat
    lastBeat = millis();          //number of milliseconds since the Arduino started running the current
program.


    beatsPerMinute = 60 / (delta / 1000.0);  //(delta/1000.0) convert to seconds
    //This expression calculates the number of beats per minute.
    //The formula for BPM is typically expressed as "beats per minute = 60 / (time in seconds
between beats)." Therefore, dividing 60 by the time in seconds between beats gives the BPM.
```

```arduino
  if (beatsPerMinute < 255 && beatsPerMinute > 20) {

    rates[rateSpot++] = (byte)beatsPerMinute;  //Store this reading in the array

    rateSpot %= RATE_SIZE;              //Wrap variable

    //after storing again make ratespot to zero 4%4 is 0


    //Take average of readings

    beatAvg = 0;

    for (byte x = 0; x < RATE_SIZE; x++)

      beatAvg += rates[x];

    beatAvg /= RATE_SIZE;

  }

}


if(beatAvg < 50){

  digitalWrite(buzzer,HIGH);

  // Buzzer_ALARM();

}

else{

digitalWrite(buzzer,LOW);

}



snprintf(bpm, MSG_BUFFER_SIZE, "Beatsperminute #%lf", beatsPerMinute);

Serial.print("Publish message: ");

Serial.println(bpm);

client.publish("heart rate", bpm);


snprintf(avg_bpm, MSG_BUFFER_SIZE, "average_beat #%lf", beatAvg);

Serial.print("Publish message: ");

Serial.println(avg_bpm);
```

```
      client.publish("average heart rate", avg_bpm);



      snprintf(msg, MSG_BUFFER_SIZE, "irvalue #%ld", irValue);

      Serial.print("Publish message: ");

      Serial.println(msg);

      client.publish("irValue", msg);


      vTaskDelay(pdMS_TO_TICKS(500));  // Delay for 500 milliseconds
  }
  vTaskDelete(NULL);
}


//oxygen task
void oxygenRateTask(void *data){
  while(1){
    if (!client.connected()) {  //mqtt client not currently connected
      reconnect();          //reconnect broker
    }
    client.loop();


    long irValue = particleSensor.getIR();

    uint32_t redValue = particleSensor.getRed();


    float ratio = static_cast<float>(redValue) / static_cast<float>(irValue);

    float spo2 = -45.060 * pow(ratio, 2) + 30.354 * ratio + 94.845;

    //The quadratic equation is a calibration curve that relates the ratio of red to infrared signals to the
SpO2 level. The coefficients (-45.060, 30.354, and 94.845) in the equation are specific to the
calibration of the sensor and may vary based on the sensor model or manufacturer.


    snprintf (oxy_spo2, MSG_BUFFER_SIZE, "oxygen percent #%lf %", spo2);

    Serial.print("Publish message: ");
```

```
    Serial.println(oxy_spo2);

    client.publish("oxygen percentage",oxy_spo2);


    if(spo2 < 60){

      for(int x=0;x<255;x++){

        analogWrite(23,x);

        delay(1);

      }

      for(int x=255;x>0;x--){

        analogWrite(23,x);

        delay(1);

      }

    }




    vTaskDelay(pdMS_TO_TICKS(500));  // Delay for 500 milliseconds

  }

  vTaskDelete(NULL);

}


void temperatureTask(void *data){

  while(1){

    if (!client.connected()) {  //mqtt client not currently connected

      reconnect();           //reconnect broker

    }

    client.loop();


    //temperature readings

    // need to use lm75 sensor

  //  float temperatureF = particleSensor.readTemperatureF();
```

```cpp
  float temperature = readTemperature();  // Call the function to read
temperature

    snprintf (temp, MSG_BUFFER_SIZE, "temperature #%lf %", temperature);

    Serial.print("Publish message: ");

    Serial.println(temp);

    client.publish("temperature ",temp);


    vTaskDelay(pdMS_TO_TICKS(500));  // Delay for 500 milliseconds
  }
  vTaskDelete(NULL);
}


void gpsTask(void* data) {
  while (1) {
    if (Serial2.available() > 0) {
      if (gps.encode(Serial2.read())) {
        if (gps.location.isValid()) {
          Serial.print("- latitude: ");
          Serial.println(gps.location.lat());


          Serial.print("- longitude: ");
          Serial.println(gps.location.lng());


          Serial.print("- altitude: ");
          if (gps.altitude.isValid())
            Serial.println(gps.altitude.meters());
          else
            Serial.println("INVALID");
        } else {
          Serial.println("- location: INVALID");
        }
```

```cpp
    Serial.print("- speed: ");
    if (gps.speed.isValid()) {
      Serial.print(gps.speed.kmph());
      Serial.println(" km/h");
    } else {
      Serial.println("INVALID");
    }


    Serial.print(F("- GPS date&time: "));
    if (gps.date.isValid() && gps.time.isValid()) {
      Serial.print(gps.date.year());
      Serial.print(F("-"));
      Serial.print(gps.date.month());
      Serial.print(F("-"));
      Serial.print(gps.date.day());
      Serial.print(F(" "));
      Serial.print(gps.time.hour());
      Serial.print(F(":"));
      Serial.print(gps.time.minute());
      Serial.print(F(":"));
      Serial.println(gps.time.second());
    } else {
      Serial.println(F("INVALID"));
    }


    Serial.println();
  }
}


if (millis() > 5000 && gps.charsProcessed() < 10)
```

```cpp
    Serial.println(F("No GPS data received: check wiring"));


     snprintf (lat, MSG_BUFFER_SIZE, "latitude #%lf ", gps.location.lat());
    Serial.print("Publish message: ");
    Serial.println(lat);
    client.publish("latitude ",lat);


     snprintf (lon, MSG_BUFFER_SIZE, "longitude #%lf ", gps.location.lng());
    Serial.print("Publish message: ");
    Serial.println(lon);
    client.publish("longitude ",lon);


    vTaskDelay(pdMS_TO_TICKS(500));  // Delay for 500 milliseconds


  }
  vTaskDelete(NULL);
}



void setup() {
  pinMode(buzzer, OUTPUT);
  pinMode(bright,OUTPUT);
  Serial.begin(115200);
  setup_wifi();                  // connecting mc to wifi
  client.setServer(mqtt_server, 1883);  // set the broker details
  client.setCallback(callback);      //It sets the callback function to handle incoming messages.
Whenever a message is received from the MQTT broker, this function (callback) will be invoked
automatically to process the message.


  Serial.println("Initializing...");        // Initialize max30102 sensor
  if (!particleSensor.begin(Wire, I2C_SPEED_FAST)) //Use default I2C port, 400kHz speed
```

```
  {

    Serial.println("MAX30102 was not found. Please check wiring/power. ");

    while (1);

  }
```

//If the sensor initialization fails, the code enters an infinite loop (while(1)) to halt the program execution. This is a way to stop further execution, as the sensor is not functioning as expected.

```
  Serial.println("Place your index finger on the sensor with steady pressure.");

  particleSensor.setup();


  //Task Creations

  //part1

  xTaskCreatePinnedToCore(heartBeatTask, "heartBeatTask", 2048, NULL, 6, &heartBeatTaskHandle, core);

  xTaskCreatePinnedToCore(oxygenRateTask, "oxygenRateTask", 2048, NULL, 6, &oxygenRateTaskHandle, core);

  xTaskCreatePinnedToCore(temperatureTask, "temperatureTask", 2048, NULL, 6, &temperatureTaskHandle, core);

  xTaskCreatePinnedToCore(gpsTask, "gpsTask", 2048, NULL, 6, &gpsTaskHandle, core);

}


void loop() {                          }
```
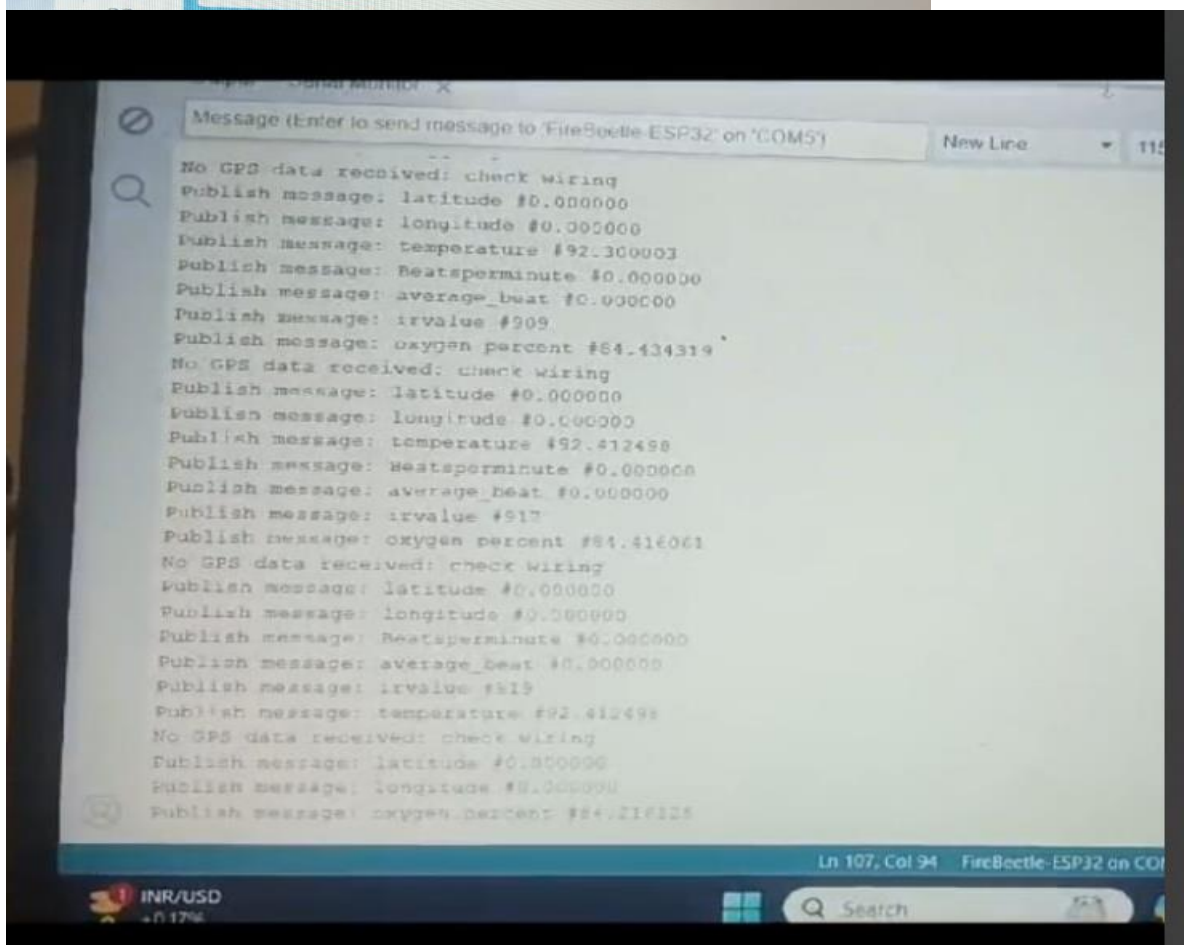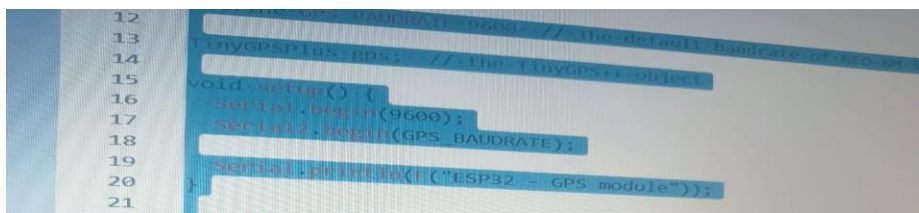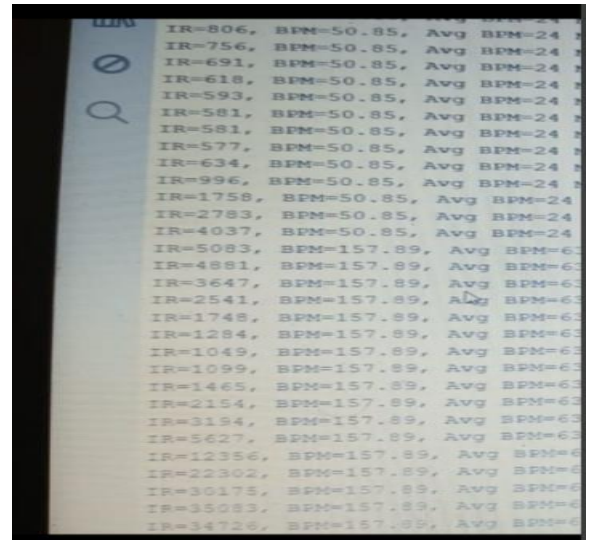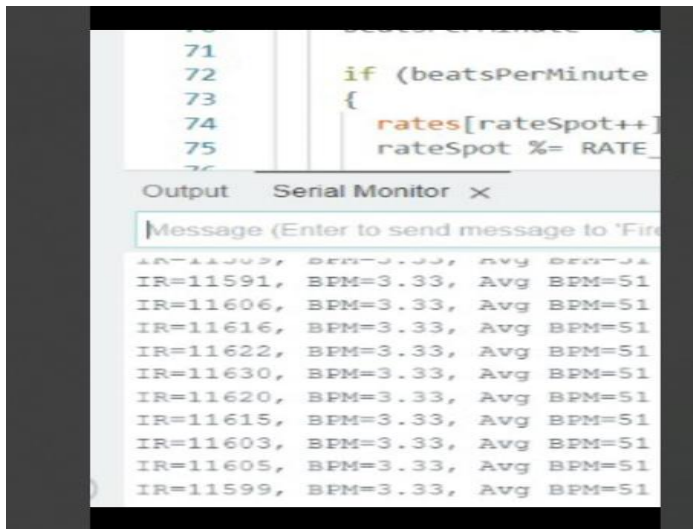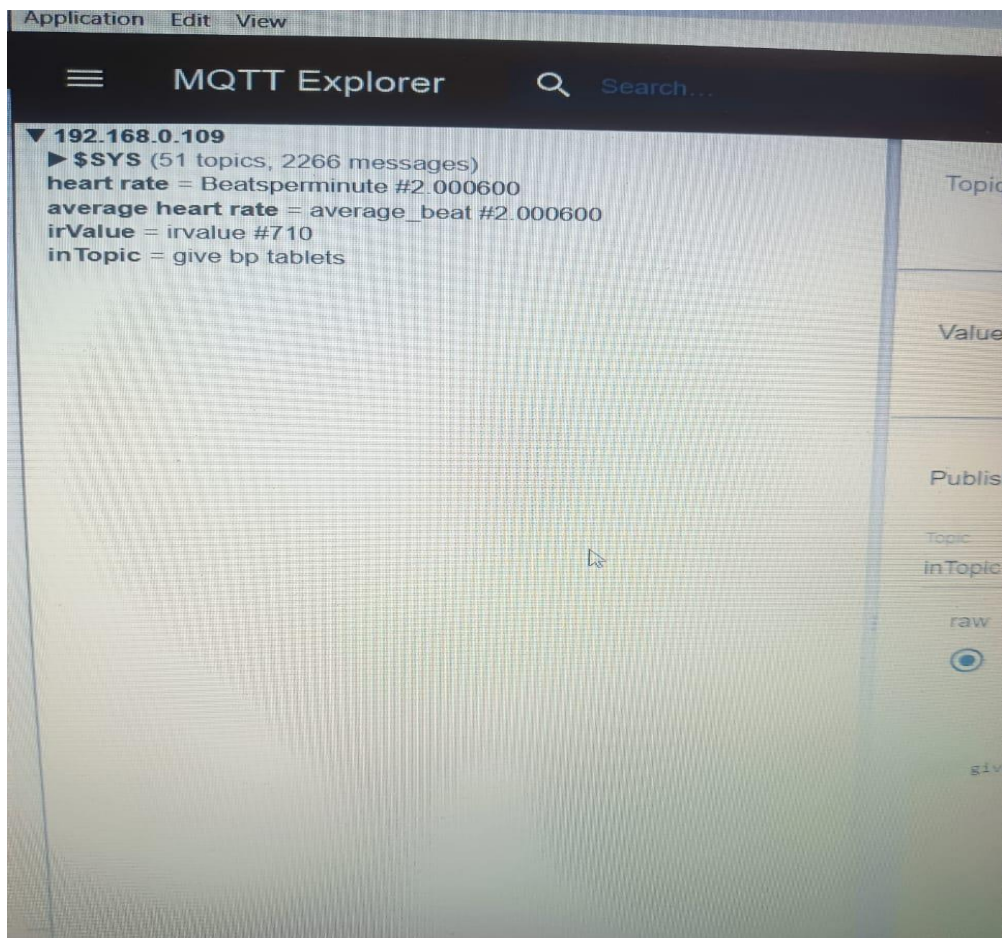
8. Results and Outputs:

8.1.1.   Heart Rate Measurement & Oxygen Saturation

MQTT Explorer output:

9. Application Code of 433MHz RF Tx & Rx

9.1 Radio Frequency Transmitter Code

```cpp
#include <RH_ASK.h>
#include <SPI.h>
#include <RadioHead.h>

// Define the transmitter pin
#define TRANSMITTER_PIN 4

// Create an instance of the RF driver
RH_ASK rf_driver(2000, TRANSMITTER_PIN);

void setup() {
  // Initialize Serial Monitor
  Serial.begin(9600);

  // Initialize RF driver
  if (!rf_driver.init()) {
    Serial.println("RF driver init failed");
  }
}
void loop() {
  const char *message = "Ambulance Arriving";
  uint8_t buf[strlen(message)];

  // Copy the message into the buffer
  strcpy((char *)buf, message);

  // Send the message
  rf_driver.send((uint8_t *)buf, strlen(message));
  rf_driver.waitPacketSent();

  // Print the message sent
  Serial.println("Message sent: ");
  // Serial.println(message);

  // Wait before sending the next message
  delay(1000);
}
```

9.2 Radio Frequency Receiver Code

```cpp
#include <RH_ASK.h>
#include <SPI.h>
#include <WiFi.h>
#include <PubSubClient.h>

// Define the receiver pin
#define RECEIVER_PIN 2

const char* ssid = "yaswanth";
const char* password = "123456789";
const char* mqtt_server = "192.168.238.15";

WiFiClient espClient;
PubSubClient client(espClient);

#define MSG_BUFFER_SIZE (50)
char msg[MSG_BUFFER_SIZE];

int value = 0;

#define grn 13
#define yel 3
#define red 4

#define grn2 7
#define yel2 8
#define red2 0

#define grn3 25
#define red3 5
#define yel3 26

#define grn4 12
#define red4 18
#define yel4 9

int grn_delay = 5000;
int yel_delay = 2000;
int red_delay = 5000;


void green_light() {
  digitalWrite(grn, HIGH);
  digitalWrite(yel, LOW);
  digitalWrite(red, LOW);
}
```

```cpp
void yellow_light() {
  digitalWrite(grn, LOW);
  digitalWrite(yel, HIGH);
  digitalWrite(red, LOW);
}

void red_light() {
  digitalWrite(grn, LOW);
  digitalWrite(yel, LOW);
  digitalWrite(red, HIGH);
}

void green_light2() {
  digitalWrite(grn2, HIGH);
  digitalWrite(yel2, LOW);
  digitalWrite(red2, LOW);
}

void yellow_light2() {
  digitalWrite(grn2, LOW);
  digitalWrite(yel2, HIGH);
  digitalWrite(red2, LOW);
}

void red_light2() {
  digitalWrite(grn2, LOW);
  digitalWrite(yel2, LOW);
  digitalWrite(red2, HIGH);
}

void green_light3() {
  digitalWrite(grn3, HIGH);
  digitalWrite(yel3, LOW);
  digitalWrite(red3, LOW);
}

void yellow_light3() {
  digitalWrite(grn3, LOW);
  digitalWrite(yel3, HIGH);
  digitalWrite(red3, LOW);
}

void red_light3() {
  digitalWrite(grn3, LOW);
  digitalWrite(yel3, LOW);
  digitalWrite(red3, HIGH);
}
```

```arduino
void green_light4() {
  digitalWrite(grn4, HIGH);
  digitalWrite(yel4, LOW);
  digitalWrite(red4, LOW);
}

void yellow_light4() {
  digitalWrite(grn4, LOW);
  digitalWrite(yel4, HIGH);
  digitalWrite(red4, LOW);
}

void red_light4() {
  digitalWrite(grn4, LOW);
  digitalWrite(yel4, LOW);
  digitalWrite(red4, HIGH);
}

// Create an instance of the RF driver
RH_ASK rf_driver(2000, RECEIVER_PIN);


void setup_wifi() {

  delay(10);
  // We start by connecting to a WiFi network
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.mode(WIFI_STA);
  //setting to wifi_sta mode
  WiFi.begin(ssid, password);
  //connection of microcontroller to wifi

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
```

```arduino
  Serial.print(topic);
  Serial.print("] ");
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
  if((char)payload[0]== 'R' && (char)payload[1]=='e' && (char)payload[2] ==
'd'){
      Serial.print("Red on lane 2");
      Serial.print("Red on Lane 3");
      Serial.print("Red on lane 4");
      // digitalWrite(red2,HIGH);
      // digitalWrite(red3,HIGH);
      // digitalWrite(red4,HIGH);
      red_light2();
      red_light3();
      red_light4();
  }
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Create a random client ID
    if (client.connect("CL_ID_3", "yesh", "yesh@123")) {
      Serial.println("connected");
      // Once connected, publish an announcement...
      client.publish("outTopic", "Green");

      // ... and resubscribe
      client.subscribe("inTopic");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void setup() {
  // Initialize Serial Monitor
  Serial.begin(9600);
  setup_wifi();   // connecting mc to wifi
  client.setServer(mqtt_server, 1883);  //set the mqtt server connections
```

```
  client.setCallback(callback);    //It sets the callback function to handle
incoming messages. Whenever a message is received from the MQTT broker, this
function (callback) will be invoked automatically to process the message.

  // Initialize RF driver
  if (!rf_driver.init()) {
    Serial.println("RF driver init failed");
  }

}

void loop() {
  if (!client.connected()) {       //if mqtt client not connected to mqtt
broker
      reconnect();          //reconnect it with broker
  }
  client.loop();    // client to perform background tasks
  //mataining mqtt connection


  // uint8_t buf[RH_ASK_MAX_MESSAGE_LEN];
  uint8_t buf[19];
  uint8_t buflen = sizeof(buf);

  // Check if a message is available
  if (rf_driver.recv(buf, &buflen)) {
    // Message received
    buf[buflen] = '\0';  // Null-terminate the string
    Serial.print("Message received: ");
    Serial.println((char*)buf);
    green_light();
    ++value;
    snprintf(msg, MSG_BUFFER_SIZE, "Green Alert #%ld", value);
    Serial.print("Publish message: ");
    Serial.println(msg);
    client.publish("outTopic", msg);

    delay(grn_delay);

  } else {
    green_light();
    delay(grn_delay);
    yellow_light();
    delay(yel_delay);
    red_light();
    delay(red_delay);
  }
}
// this code for receiver
```
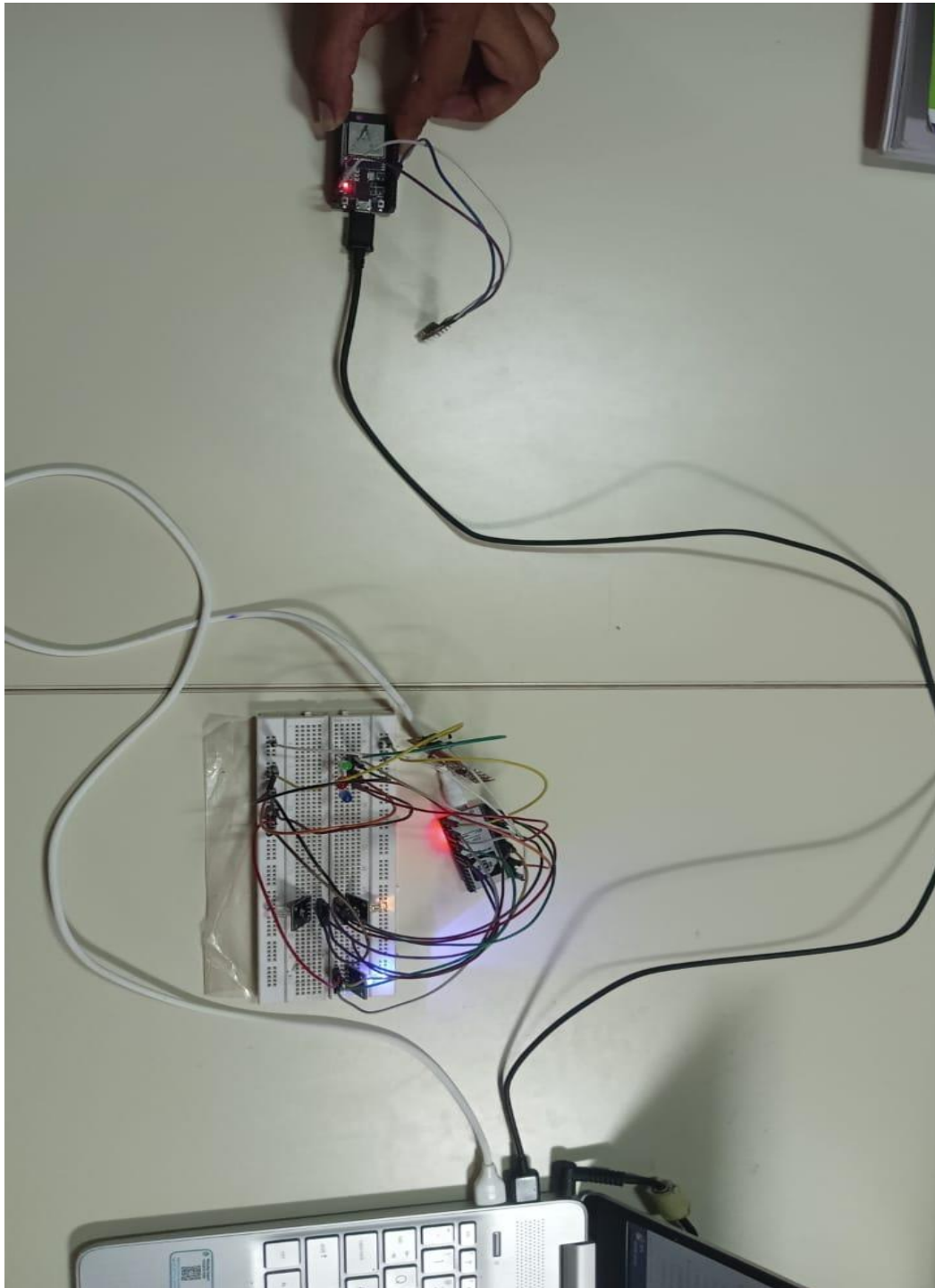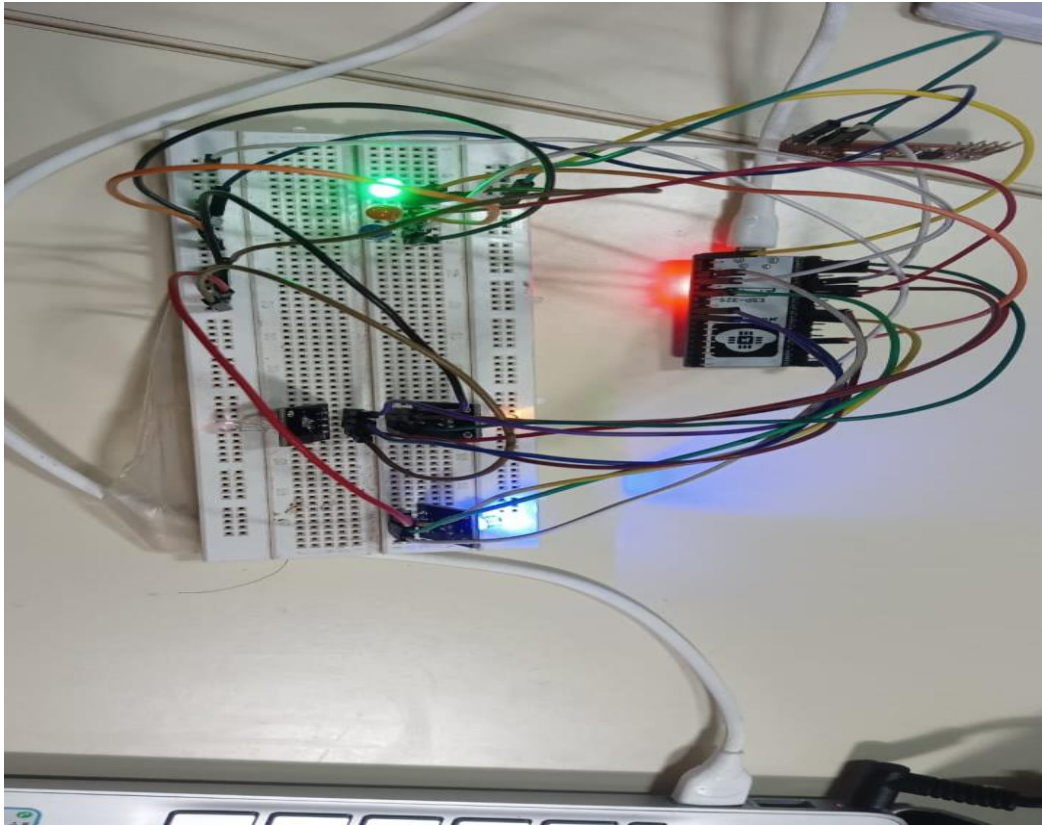
9.3 OUTPUT OF Transmitter & Receiver

Communication between RF Tx in Ambulance & Rx in Traffic Light.

9.4 Working

We place the Transmitter terminal in Ambulance. We place the Receiver Terminal in Traffic Light.

When the Ambulance reaches near to the traffic light

1. The alert message is sent by transmitter to Receiver.
2. From receiver the "Green Alert" message is sent to broker through mqtt protocol
3. The Traffic Management authority will receive this alert message then it will send "Red" message to broker then after it is received in callback function
4. After receiving of "Red" message. We make the other Traffic Lights to red. (By DigitalWrite Function).
5. So, Ambulance can move freely. Finally, we can avoid some sort of traffic by going through this approach.