PROJECT TITLE:-

Iot Based Multiple Hazards Detection System

Table Of Contents Includes:-

Introduction of our project

Introducing our cutting-edge project: the IoT-based Multiple Hazards Detection System! In today's rapidly evolving world, ensuring safety and security is paramount. Our innovative system harnesses the power of Internet of Things (IoT) technology to detect various hazards in real-time, providing proactive measures to mitigate risks and safeguard lives and property.

At the heart of our system lies an array of sophisticated sensors, each designed to detect specific environmental hazards. We employ the MQ135 sensor to monitor air quality, alerting users to the presence of harmful gases such as carbon monoxide, ammonia, benzene, and alcohol vapor. Additionally, the Passive Infrared (PIR) sensor and Infrared (IR) sensor are utilized for motion detection, enabling early detection of potential intrusions or unauthorized access.

To accurately pinpoint the location of hazards and track the movement of assets or individuals, we integrate a GPS module into our system. This ensures precise geolocation data, facilitating rapid response and evacuation procedures in emergency situations.

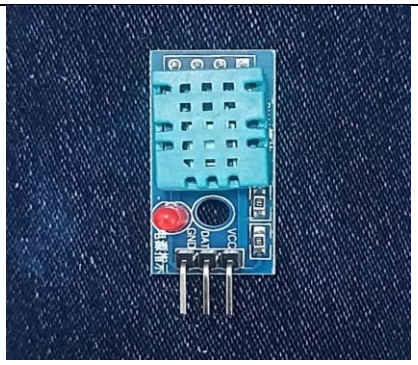
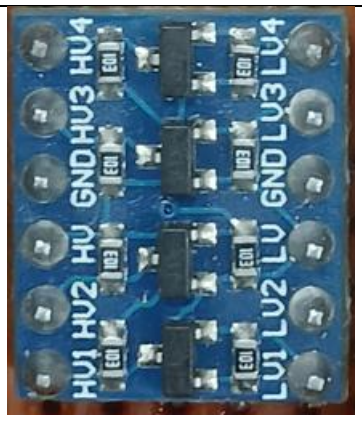
Furthermore, communication is key in ensuring timely alerts and coordination during critical events. Leveraging the Long Range (LoRa) module, our system enables long-distance, low-power communication, ensuring reliable connectivity even in remote or challenging environments.

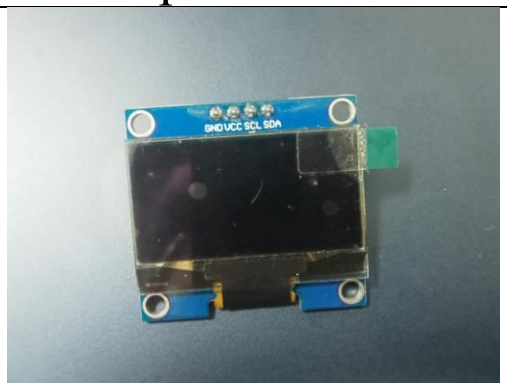
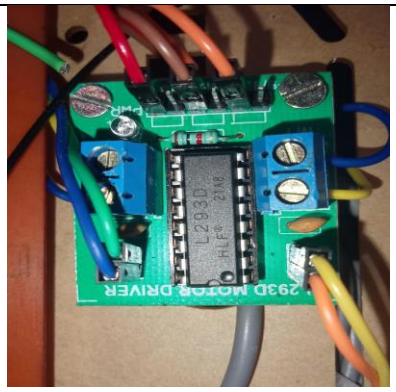
But our system doesn't stop there. We understand the importance of data visualization and analysis in making informed decisions. That's why we leverage the power of IoT platform ThingSpeak for comprehensive data visualization and analytics. ThingSpeak provides real-time insights into environmental conditions, trends, and anomalies, empowering users to take proactive measures to mitigate risks and optimize resource allocation.

Moreover, to capture crucial environmental parameters such as temperature and humidity, we integrate the DHT11 sensor into our system, providing comprehensive environmental monitoring capabilities.

In summary, our IoT-based Multiple Hazards Detection System represents a leap forward in safety and security technology. By combining advanced sensors, precise geolocation, long-range communication, and powerful data analytics, we empower organizations and communities to detect, analyze, and respond to hazards in real-time, ensuring a safer and more secure environment for all.

List Of Figures and Table:-

4	Fig 4	 DHT 11
5	Fig 5.	 Proximity Sensor
6	Fig 6.	 Logic Level Converter.
7	Fig 7	 Pir Sensor

8	Fig 8	 Gps Module
9	Fig 9	 O-LED
10	Fig 10	 LoRa Module.
11.	Fig 11	 L293D motor driv

12.	Fig 12	 Dc-DC buck converter.
13.	Fig 13	 Battery Operated Motors.

Software Used :-

In our iot based multiple hazards detection system we are using some libraries as well as Arduino IDE platform for programming/coding.

And the output or result will be also shown on Iot platform Thing Speak.

The list of libraries includes :-

- **#<DHT.h> (1.0.1) –** This library is used for DHT 11 sensor which is used for measuring temperature and humidity of surroundings.

- **#include<TinyGps++(0.0.1)-** TinyGPS, this library provides compact and easy-to-use methods for extracting position, date, time, altitude, speed, and course from consumer GPS devices.

- **#include<Thing Speak.h> (2.0.1)-** This library allows user to visualize the readings and analyze live data streams in cloud. The readings are stored in a channel. Each channel stores 8 fields.

- **#include<LoRa.h> (0.8.0)-** We are using this library in our project to communicate over the longer distance as LoRa (Long Range) covers the radius of 10km also the power consumption is low.

- **#include<Wifi.h>(1.1.0) –** This library is used for enabling connectivity of wifi with the Esp32.

- **#include“BluetoothSerial.h”(0.9.0)-** This library is used to enable the Bluetooth communication capabilities. Specifically , it allows the esp32 to act as a Bluetooth serial port profile device , enabling to communicate wirelessly with other Bluetooth devices such as smartphones, laptop, tablets etc.

- **#include<SPI.h>(1.1.2) -** The #include <SPI.h> library in Arduino IDE serves a fundamental role in facilitating Serial Peripheral Interface (SPI) communication between the ESP32 microcontroller and other

peripheral devices, such as sensors, displays, and other microcontrollers.

- **#include<u8g2lib.h>(0.8.2)-** The u8g2lib library, also known as the U8g2 library, serves the purpose of facilitating the programming of monochrome graphical displays, particularly OLED (Organic Light Emitting Diode) displays, in embedded systems and microcontroller projects. It provides an interface for controlling these displays efficiently, enabling developers to draw text, shapes, and images on the screen. This library is also compatible with ESP32.

COMPONENTS USED WITH PROTOCOLS, LIMITATIONS AND PIN CONNECTED TO IT:

- **ESP32 WROOM Generic DevKit**

FIG 1.

COMMUNICATION PROTOCOL:

Various (e.g., UART, SPI, I2C, Wi-Fi)

LIMITATIONS

- Limited RAM and flash memory.
- Power consumption can vary depending on usage.

- May require additional components for specific functionalities (e.g., external antenna for WiFi).

- **GAS SENSOR – (MQ-135)**

FIG 2.

COMMUNICATION PROTOCOL:

N/A

WORKING PRINCIPLE :

The sensing element used by MQ 135 gas sensor is a tin dioxide(SnO2) semiconductor. The property of tin dioxide is of being an excellent conductor at high temperatures when exposed to certain gases. The ceramic element of sensor consist with sno2 and when this element comes in contact with a specific gas it changes the conductivity of the sensor. This change in conductivity is then converted into an electrical signal that can be measured and interpreted. The mq 135 sensor is affected by how many gases are in the air around it . when there is more of a certain gas the sensor reacts more strongly ,and we get a bigger signal from it. this means the MQ 135 can tell us a lot about the air around us and whether there are any dangerous gases present.

Protocols connected to it:

The pin used in project for MQ-135 is its Analog pin which is connected to ESP32 general I/O32 .

I/O32 PROTOCOL: Analog to digital converter (ADC)

TECHNICAL LIMITATIONS:

- For MQ-135 Gas Sensor, it detects multiple gases at a time, not a targeted gas. This can make it less accurate, especially if there are lots of different gases in the air.
- The sensor's performance is influenced by temperature and humidity levels , again affecting its accuracy.

PINS CONNECTED WITH ESP32:

- VCC – 5V
- GND- GROUND
- DIGITAL OUTPUT- N/A
- ANALOG OUTPUT- GPI034

RANGE OF OPERATION:

The MQ-135 is capable of sensing ammonia (10ppm-300ppm), benzene (10ppm-1,000ppm), and alcohol (10ppm-300ppm) air concentration levels. The ideal sensing condition for the MQ135 is 20°C ±2°C at 65% ±5% humidity.

- **IR (FLAME) SENSOR- (LM393)**

FIG 3.

COMMUNICATION PROTOCOL:

N/A

Working principle:

IR sensor consists of two elements including infrared source and infrared detection. The infrared source or transmitter is an LED that emits the light in order to sense some object of the surroundings An IR sensor can measure the heat of an object as well as detects the motion. Usually, in the infrared spectrum, all the objects radiate some form of thermal radiation. These types of radiations are invisible to our eyes, but infrared sensor can detect these radiations.

Infrared detectors or infrared receivers detect the radiation from an IR transmitter. IR receivers come in the form of photodiodes and phototransistors. Infrared Photodiodes are different from normal photo diodes as they detect only infrared radiation. When used in an infrared transmitter – receiver combination, the wavelength of the receiver should match with that of the transmitter.

TECHNICAL LIMITATIONS:

- The IR sensor might not be able to detect things far away. It depends on how well the sensor works, how bright the light is around it, and what the thing it's looking at is made of.
- The response time of the IR sensor may introduce delays in detecting
- Extreme temperatures may affect the performance of the IR sensor, leading to variations in detection accuracy or reliability.

PINS CONNECTED WITH ESP32:

- VCC – 5V
- GND- GROUND
- OUTPUT- GPIO35

RANGE OF OPERATION:

The little knob on the sensor helps you adjust how far it can see. With this adjustment, it can detect things that are anywhere from 2 centimeters to 80 centimeters away.

- **<u>TEMPERATURE AND HUMIDITY SENSOR-(DHT11)</u>**

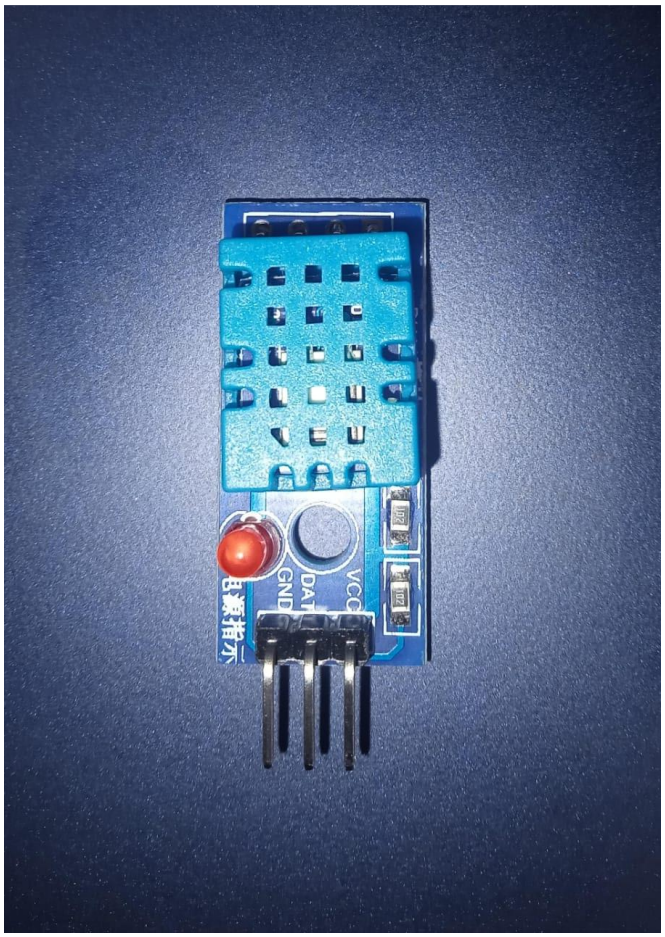


FIG 4.

COMMUNICATION PROTOCOL:

N/A

Working principle:

DHT11 sensor consists of a capacitive humidity sensing element and a thermistor for sensing temperature. The humidity sensor in the DHT11 has two electrodes with a dielectric material in between that holds moisture. When the humidity changes, the amount of electricity the sensor uses changes too. The DHT11 measures this change and turns it into a number that we can understand.

For measuring temperature, the DHT11 uses a special kind of material that changes its electricity use when it gets hotter or colder. It's usually made of special types of ceramic or plastic. Even a tiny change in temperature can make a big change in the electricity it uses.

The DHT11 can measure temperatures from 0 to 50 degrees Celsius and humidity levels from 20 to 80%. It's accurate to within 2 degrees for temperature and 5% for humidity. It gives us a new reading every second, and it works with voltages between 3 and 5 volts. It doesn't use much power, only about 2.5 milliamps.

TECHNICAL LIMITATIONS:

- Limited temperature range (0°C to 50°C) and humidity range (20% to 80% RH).
- Limited Accuracy:
- Slower response time.

PINS CONNECTED WITH ESP32:

- VCC – 5V
- GND- GROUND
- OUTPUT- GPIO32.

RANGE OF OPERATION:

1. Temperature:
    - Operating Range: 0°C to 50°C (32°F to 122°F)
    - Accuracy: ±2°C
2. Humidity:
    - Operating Range: 20% to 80% relative humidity
    - Accuracy: ±5%

- **PROXIMITY SENSOR-PNP-NO**

FIG 5.

COMMUNICATION PROTOCOL:

N/A

Working principle:

Inductive sensors are designed specifically for the detection of conductive and any metal objects and also objects that can be detected from close proximity. Inductive sensor is used to detect multiple types of metal. These sensors are non-contacting and performed based on induction law. When a metallic object comes close to the sensor, the oscillator coil starts to move. When the object approaches within the detection range, the sensor uses an electromagnetic field that is sent out for detection of the object, then the sensor searches for changes in the returning signal. After detecting the object, the returning signal becomes shorter than when there is no object. We are using unshielded Proximity Sensor. In the unshielded sensors, the

Electromagnetic field generated by the coil is not restricted and has longer and larger sensing ranges.

TECHNICAL LIMITATIONS:

- Limited detection range.
- May require adjustment for optimal sensitivity.
- Sensitivity may vary based on environmental conditions.

PINS CONNECTED WITH ESP32 THROUGH LLC:

- VCC (BROWN) – 12V
- GND (BLUE) - GROUND
- OUTPUT(BLACK) – GPIO25

RANGE OF OPERATION:

Typically, inductive proximity sensors with PNP output have a detection range between 1mm to 15mm. This means they can detect metal objects within this range.

- **LOGIC LEVEL CONVERTER (LLC)**

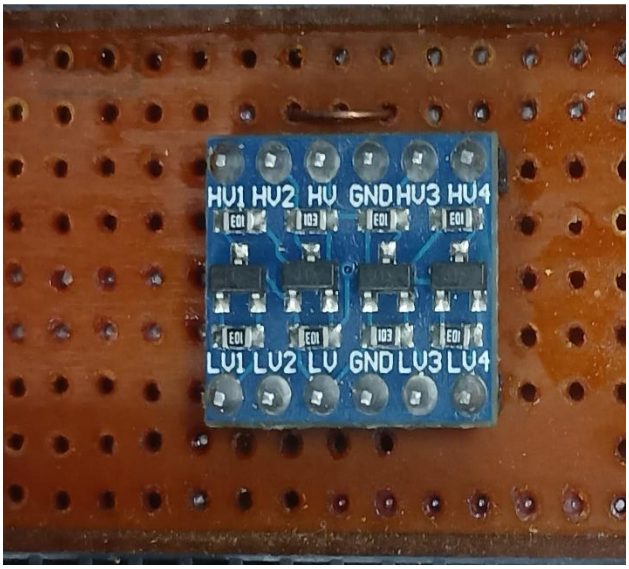


FIG 6.

COMMUNICATION PROTOCOL:

N/A

Working principle:

A logic level converter is a necessary circuit while interfacing devices with different voltage requirements. Logic level converter steps up or steps down the voltage as per the requirement of a device. Here we are using to step down the Voltage. A logic level converter allows signals to move both ways, from input to output and back, while keeping the voltage levels safe. Inside, it uses components like MOSFETs or transistors to change the voltage levels, ensuring signals can travel between devices with different voltage needs without causing harm. Some converters even have isolation features to shield against voltage problems. Overall, these converters ensure signals flow smoothly between devices, making sure everything works well together in a system or network.

TECHNICAL LIMITATIONS:

- Limited to converting voltage levels between different logic standards (e.g., 3.3V to 5V).
- May introduce signal delay or distortion in some cases.
- Limited number of channels for voltage conversion (typically 4 channels in common logic level converters).

PINS CONNECTED WITH ESP32:

- HV – 12V
- GND – GROUND
- HV1 – O/P OF PROXIMITY SENSOR
- LV – 3.3 OF ESP32
- GND – GROUND
- LV1 – GPIO25

RANGE OF OPERATION:

The logic level converter circuit convert signals as low as 1.8 V to as high as 5V and vice versa,

- **PIR SENSOR- (HW416B)**

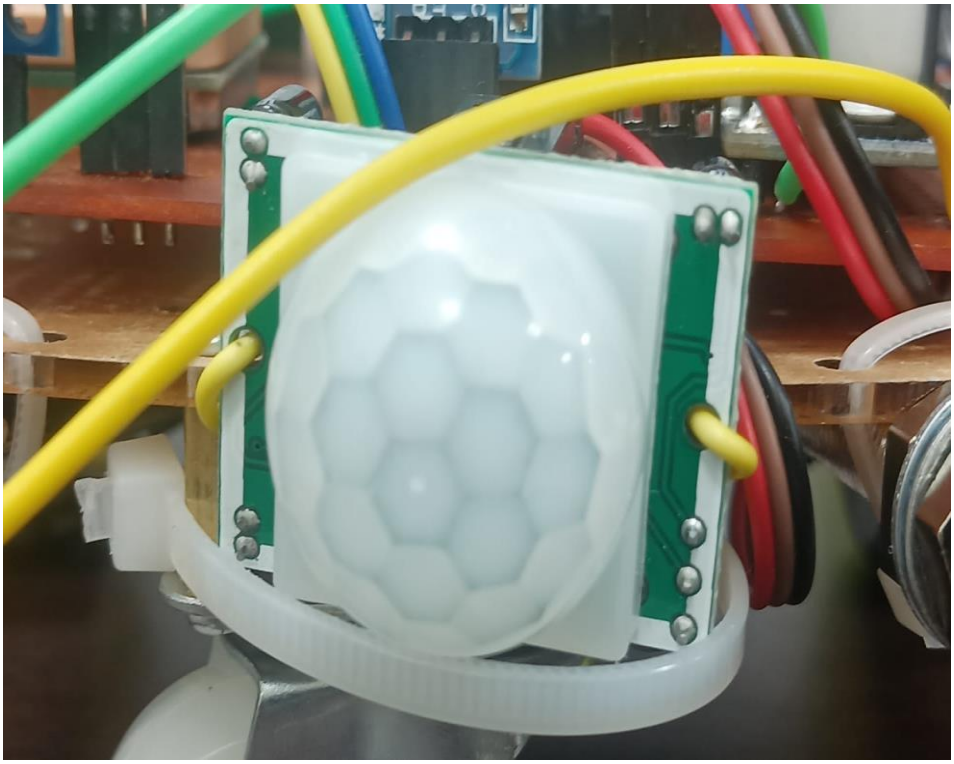


FIG 7.

COMMUNICATION PROTOCOL:

N/A

Working principle:

The passive infrared sensor does not radiate energy to space. It receives the infrared radiation from the human body to make an alarm. Any object with temperature is constantly radiating infrared rays to the outside world. The surface temperature of the human body is between 36° C - 27 ° C and most of its radiant energy is concentrated in the wavelength range of 8 um-12 um.It has two main parts:a sensor and a lens. The sensor senses the heat from any object or person that passes

by, and it's made of a special material called pyroelectric material. This material can turn heat into electricity. The pyroelectric sensor sees the moving human body for a while and then does not see it, so the infrared radiation of human body constantly changes the temperature of the pyroelectric material. So that it outputs a corresponding signal, which is the alarm signal.

TECHNICAL LIMITATIONS:

- They have lower sensitivity and less coverage compare to microwave sensors.
- It does not operate greater than 35 degree C.
- It works effectively in LOS (Line of Sight) and will have problems in the corner regions.
- It is insensitive to very slow motion of the objects

.PINS CONNECTED WITH ESP32:

- VCC – 3.3V
- GND- GROUND
- OUTPUT- GPIO33

RANGE OF OPERATION:

- Indoor passive infrared: Detection distances range from 25 cm to 20 m.

- **GPS SENSOR – (NE0-6M).**

FIG 8.

COMMUNICATION PROTOCOL:

UART (Serial)

WORKING PRINCIPLE:

The NEO-6M module is like a GPS receiver for your device. It catches signals from different satellites up in space, including GPS, GLONASS, Galileo, and BeiDou. These satellites send out signals with timing details and where they are in space. Next, the module figures out where it is by looking at these signals. It uses a mix of math tricks to work out the distance between itself and each satellite. By checking how long it takes for signals to travel, it can find its spot on Earth. Once it knows where it is, it puts the info into a neat format called NMEA sentences. These sentences include details like latitude, longitude, how high up it

is, how fast it's moving, and the time. Then, your device can use this data for all sorts of things, like maps, directions, or tracking where something is. And if you need to tweak how the module works, you can send special commands to it using the serial connection. These commands let you change settings or control how it behaves.

TECHNICAL LIMITATIONS:

- Requires a clear view of the sky for optimal satellite reception.
- Limited accuracy in urban canyons or areas with obstructed sky view.
- May have longer acquisition times in cold start situations.

PINS CONNECTED WITH ESP32:

- VCC – 3.3V
- GND- GROUND
- TX- GPIO03 (RX)
- RX- GPIO01 (TX)

RANGE OF OPERATION:

the NEO-6M can typically receive signals from satellites up to approximately 20,000 kilometers (about 12,427 miles) away.

- **OLED DISPLAY MODULE – 0.96 inches 128×64.**

FIG 9.

COMMUNICATION PROTOCOL:

I2C (INTER INTEGRATED CIRCUIT)

WORKING PRINCIPLE:

An OLED display module, with its 128x64 resolution, works like a tiny screen that uses organic light-emitting diodes (OLEDs) to create images and text. These OLEDs are made of organic compounds that emit light when an electric current passes through them.

When you send information to the OLED display, such as text or graphics, the module activates specific OLEDs to light up and create the desired shapes and characters. Each pixel on the screen has its own OLED, and by controlling the intensity of the electric current flowing through each OLED, the module can produce different colors and brightness levels.

The OLED display module doesn't need a backlight like traditional LCD displays because each pixel emits its own light. This allows for high contrast

ratios, vibrant colors, and wide viewing angles. OLED displays are energy-efficient because they only consume power when emitting light, making them ideal for portable devices and battery-powered applications.

TECHNICAL LIMITATIONS:

- Limited display size and resolution.
- Limited color capabilities.
- Higher power consumption compared to other display types.

PINS CONNECTED WITH ESP32:

- VCC – 3.3V
- GND- GROUND
- SERIAL CLOCK (SCL)- GPIO36
- SERIAL DATA (SDA)- GPIO33

RANGE OF OPERATION:

OLED display modules like the 128x64 variant can operate within a wide range of temperatures, typically from -40°C to 85°C (-40°F to 185°F).

- **LORA MODULE – (SX1278)**

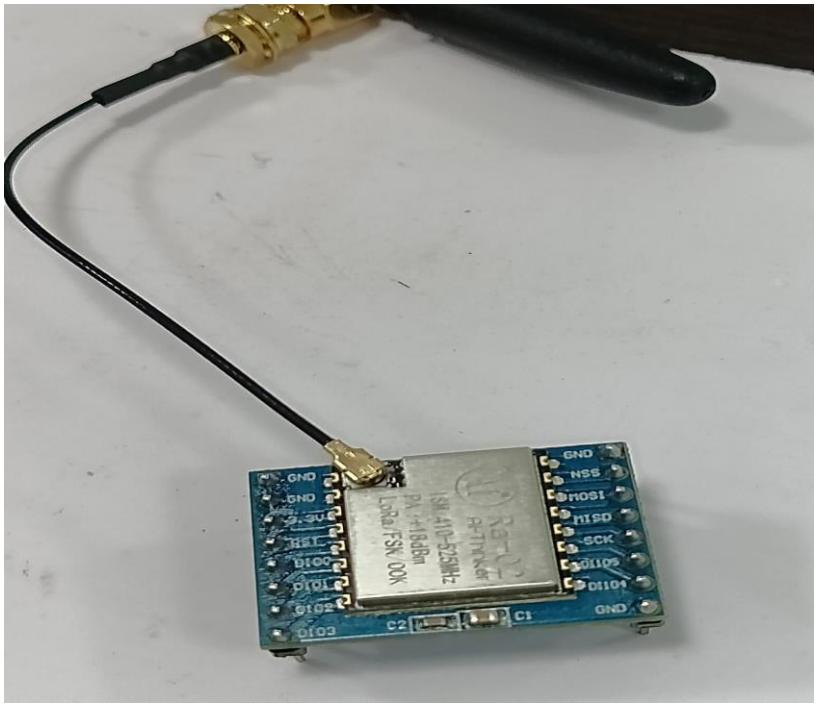


FIG 10.

COMMUNICATION PROTOCOL:

SPI

WORKING PRINCIPLE:

The term LoRa stands for Long Range. It is a long-range, low power wireless platform that has become the de-facto technology for Internet of Things (IoT) networks worldwide. LoRa is a spread spectrum modulation technique derived from chirp spread spectrum (CSS) technology. The basic principle is that information is encoded using chirp (a gradual increase or decrease in the frequency of the carrier wave over time). Before sending a message, the LoRa transmitter will send out a chirp signal to check that the band is free to send the message. Once the LoRa receiver has picked up the preamble chirp from the

transmitter, the end of the preamble is signalled by the reverse chirp, which tells the LoRa transmitter that is it clear to begin transmission.

TECHNICAL LIMITATIONS:

- Low transmission rate
- Slow data transfer rate
- LoRa's data transmission payload is very small and only has one byte limit.

PINS CONNECTED WITH ESP32:

- 3.3V – 3.3V
- GND- GROUND
- NSS- GPIO05 (SPI)
- MOSI-GPIO23(SPI)
- MISO- GPIO19(SPI)
- SCK- GPIO18(SPI)
- RST- GPIO14
- DI01- GPIO04

RANGE OF OPERATION:

It adopts advanced LoRa spread spectrum technology, with a communication distance of 10,000 meters.

- **L293D MOTOR DRIVER**

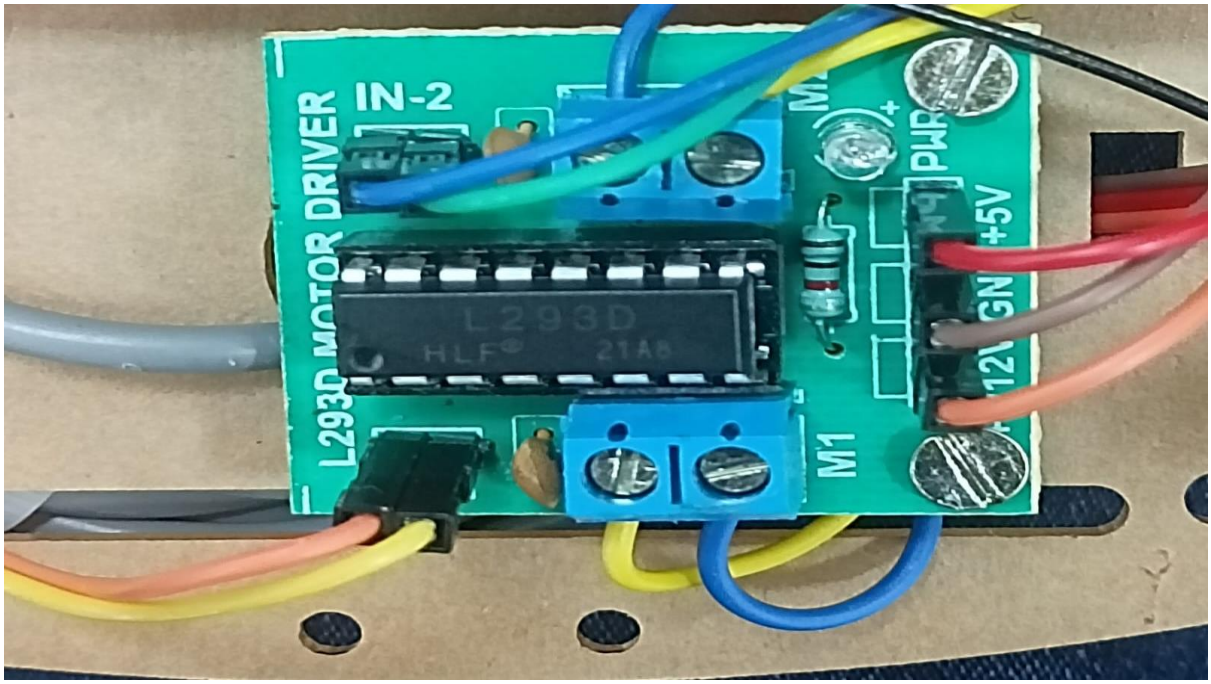


FIG 11.

COMMUNICATION PROTOCOL:

N/A

WORKING PRINCIPLE:

The L293D motor driver module is used to control the movement of DC motors. Its primary function is to manage the direction and speed of one or more DC motors by receiving control signals from a microcontroller or other control devices.

The module contains H-bridge circuits, which are specialized circuits capable of controlling the direction of motor rotation by changing the polarity of the voltage applied to the motor terminals. This allows the motor to move forward, reverse, or stop, depending on the signals received by the motor driver. In operation, the L293D module receives input signals from the microcontroller, indicating the desired direction and speed of the motor. Based on these signals, the module switches the appropriate transistors within the H-bridge circuits to control the flow of current through the motor windings. One key feature of the

L293D module is its ability to handle relatively high currents and voltages, making it suitable for driving a wide range of DC motors. Additionally, the module includes built-in diodes to protect against voltage spikes generated by the motor's inductive load, ensuring safe and reliable operation.

TECHNICAL LIMITATIONS:

- Limited current handling capacity.
- May generate heat under high loads.
- Requires external components for motor control.

PINS CONNECTED WITH ESP32:

- VO(9V/12V) - +12V
- VO(5V) - +5V
- GND – GROUND
- MA+ – MOTOR A(+)
- MA- – MOTOR A(-)
- MB+ – MOTOR B(+)
- MB- – MOTOR B (-)
- IN1A– GPIO0
- IN1A–GPIO04
- IN2B–GPIO26
- IN2B–GPIO27

RANGE OF OPERATION:

i. Input Voltage Range: 4.5V to 36V
ii. Output Voltage Range: Close to input voltage
iii. Current Handling Capacity: Up to 600mA continuous per channel, and up to 1.2A peak per channel

- **LM2596 DC-DC BUCK CONVERTER**

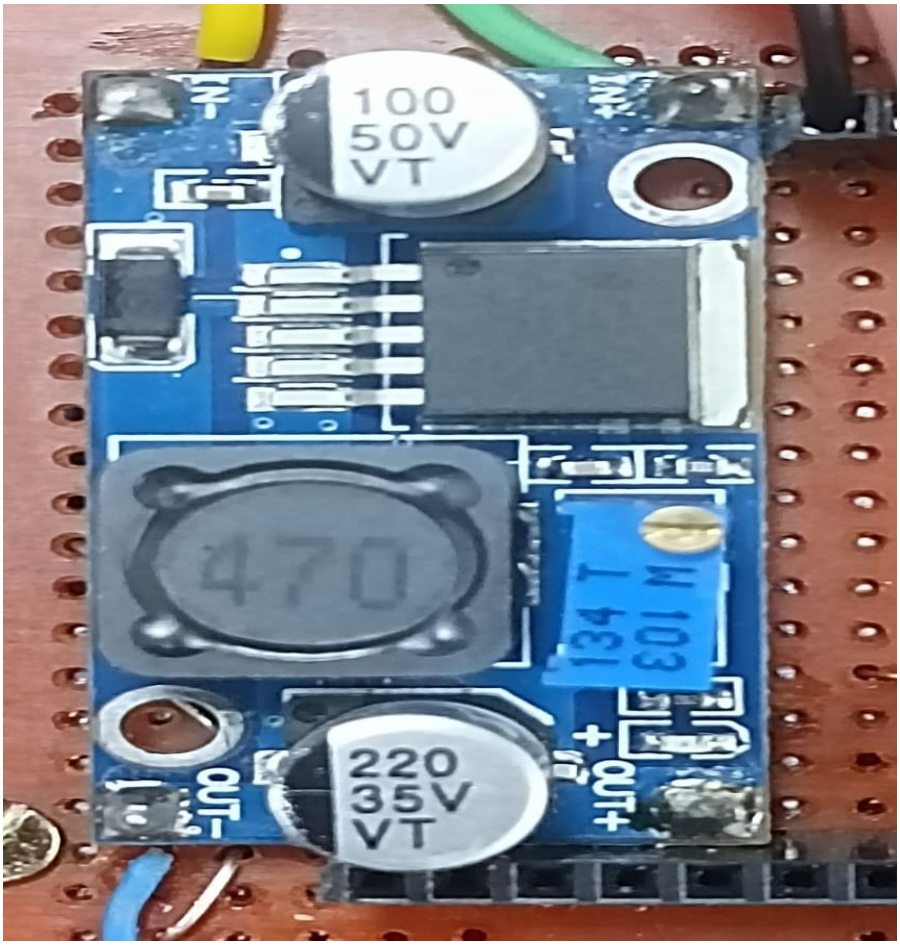


FIG 12.

COMMUNICATION PROTOCOL:

N/A

WORKING PRINCIPLE:

LM2596 is a voltage regulator mainly used to step down the voltage or to drive load under 3A. It is also known as a DC-to-DC power converter or buck converter which is used to step down the voltage from its input supply to the output load. The current goes up during this voltage step-down process. When an input voltage is applied to the LM2596 converter, it uses an internal switching mechanism to

regulate the voltage. First, the input voltage passes through an inductor, which stores energy. Then, the LM2596 rapidly switches an internal transistor on and off, creating a series of pulses. These pulses are smoothed out by a capacitor, resulting in a steady output voltage. One of the key advantages of the LM2596 converter is its ability to handle a wide range of input voltages while providing a stable output voltage. It's highly efficient, meaning it wastes minimal energy during the conversion process.

TECHNICAL LIMITATIONS:

- Limited input and output voltage range.
- May generate heat under high loads or inefficient operation.
- Requires proper heat sinking and current handling consideration

PINS CONNECTED WITH ESP32:

- IN+ – 12V
- IN- – GROUND
- OUT+ – ESP 5V PIN
- OUT- – GROUND

RANGE OF OPERATION:

i. The LM2596 typically accepts input voltages ranging from around 4.5 volts to 40 volts.
ii. The LM2596 can regulate output voltages from as low as 1.23 volts up to around 37 volts depending on model.
iii. the LM2596 can handle continuous output currents in the range of 1.5 amps to 3 amps.

- **BATTERY OPERATED MOTORS**

FIG 13.

COMMUNICATION PROTOCOL:

N/A

WORKING PRINCIPLE:

The L293D motor driver module is used to control the movement of DC motors. Its primary function is to manage the direction and speed of one or more DC motors by receiving control signals from a microcontroller or other control devices.

The module contains H-bridge circuits, which are specialized circuits capable of controlling the direction of motor rotation by changing the polarity of the voltage applied to the motor terminals. This allows the motor to move forward, reverse, or stop, depending on the signals received by the motor driver. In operation, the L293D module receives input signals from the microcontroller,

indicating the desired direction and speed of the motor. Based on these signals, the module switches the appropriate transistors within the H-bridge circuits to control the flow of current through the motor windings. One key feature of the L293D module is its ability to handle relatively high currents and voltages, making it suitable for driving a wide range of DC motors. Additionally, the module includes built-in diodes to protect against voltage spikes generated by the motor's inductive load, ensuring safe and reliable operation.

TECHNICAL LIMITATIONS:

- Requires proper power management to avoid over-discharge of batteries.
- Operating time depends on motor power consumption.

RANGE OF OPERATION:

i. Input Voltage Range: 4.5V to 36V
ii. Output Voltage Range: Close to input voltage
iii. Current Handling Capacity: Up to 600mA continuous per channel, and up to 1.2A peak per channel

Details of each and every component:-

Sr/no	Name of sensors	Input		output	
		voltage	current	Voltage	Current
1	Temperature & humidity sensor	3.3V to 5V	1 to 2.5 mA	2V	<250mA
2	Gas sensor	3.3V to 5V	150mA	5 to 10V	4 to 20mA
3	Pir sensor	3.3V to 5V	50 to 60mA	3V	1 to 3W
4	Esp 32	3 to 3.6V	250mA	3.3V	250mA
5	0-Led display	3.3V to 5V	20 to 30mA	1.8 to 3.3V	20mA
6	Motor driver module	3.6V to 4.5V	1 to 600mA	4.5V	2.4A
7	Flam sensor	3.3V to 5V	16mA	3.3V	2 to 6mA
8	Buck converter	4V to 40V	333mA	1.25V to 37V	1A to 3A
9	Inductive proximity sensor	10 to 30V	250mA	5 to 10V	200mA
10	Dc motor	3 to 12V	40 to 180mA	12V	18W

Methodology And Problems Faced :

METHODOLOGY

The following are the Steps we performed in our IOT based multiple hazard detection system :

- We first figured out how many components and which sensors will we require to make this project, and how the body of our project will look like.
- Moving forward we explored the detail features in our project and possibility of the features we can add.
- After finalizing the above parameters, we broke down each component with esp32 and interfaced them individually with their individual codes to make sure the connectivity and compatibility of our components.
- We started interfacing of all individual components together and also merging the codes one by one on breadboard before soldering them onto Pcb (Printed Circuit Board).
- After ensuring the working and compatibility of components with each other, we immediately initiated the soldering on pcb.
- We roughly mapped out the circuit with a marker on pcb to solder the components in the right place with right connection.
- After soldering, the final step was to make our project representable on the chassis board and build its shape into a vehicle.

Problems faced

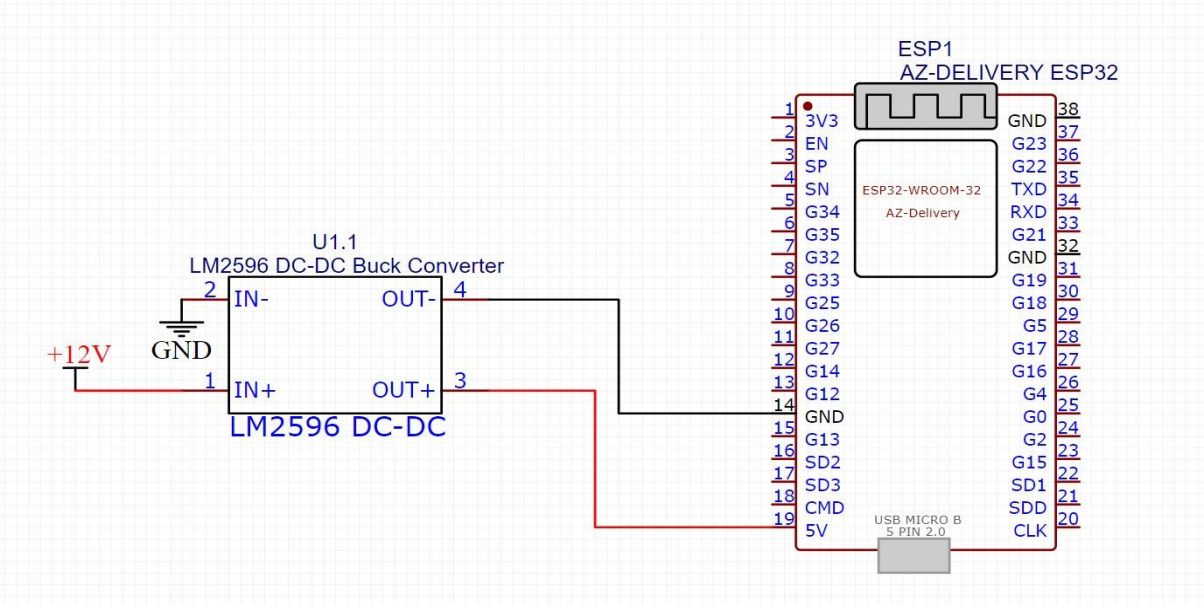
- LORA MODULE : we have two lora modules of same model number, one on transmitter side and the other on receiver. To exchange data between the two modules we faced problems such as not being able to receive the data properly sended by the transmitter side. We overcome this problems by realizing that the baud rates of both modules was not same but even after changing them it was still not receiving so we changed the pins of lora with esp32 entirely and it started receiving the accurate data.
- OLED DISPLAY : The problems we faced with Oled interfacing was in code, we were not able to find a suitable library which is compatible with our Oled but finally by using the U8glib library our Oled was working properly. The reason for using U8g2lib library was because it supports a wide range of display controllers and communication interfaces. It provides a united interface for controlling OLED displays, making it easier to work with different display modules without needing to write custom code for each one.

- MOTOR DRIVER : All connections and connectivity of motor driver with esp32 and motors was proper but we were still facing problems even after trying many things so we decided to change our motor driver with new one with same model and the problems were solved eventually.

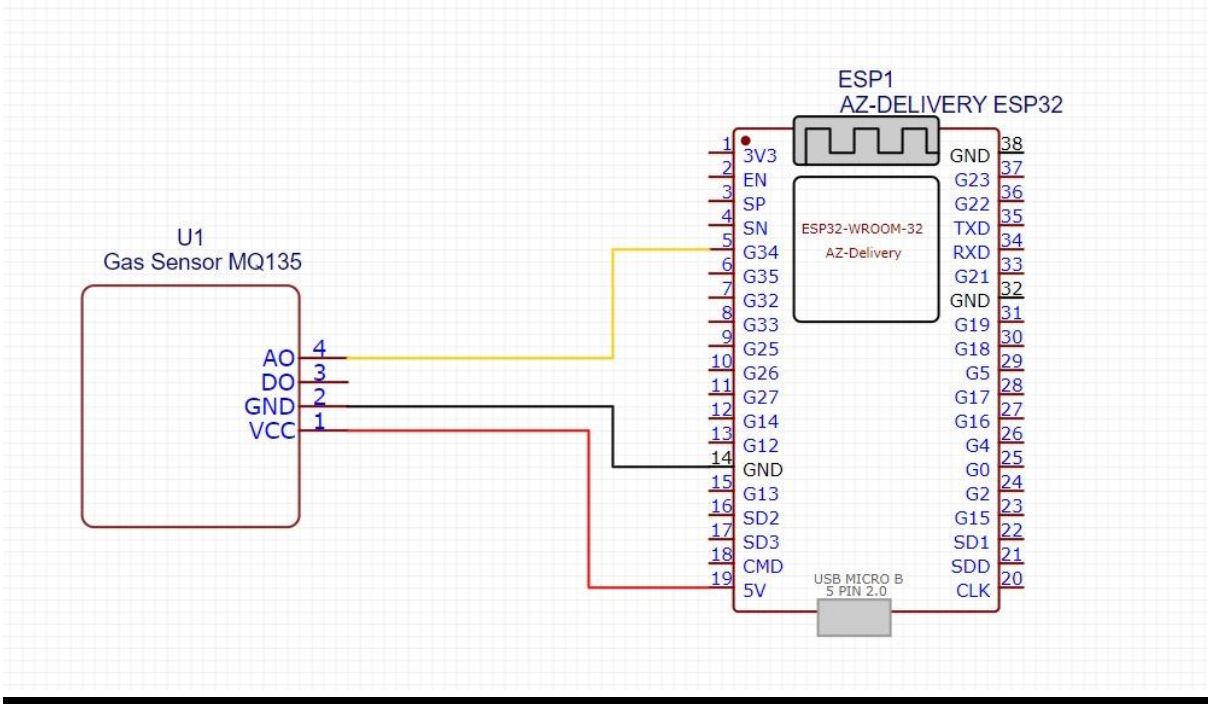
- GPS SENSOR : In gps sensor our antenna was not catching enough satellites for it to get locked in and send the location because of not enough open space so after changing the location to test it again ,the gps also worked properly.

Individual Components Schematics And Codes:-

ESP32 interfacing with DC-DC buck converter:-



ESP32 interfacing with MQ135 Gas sensor :-

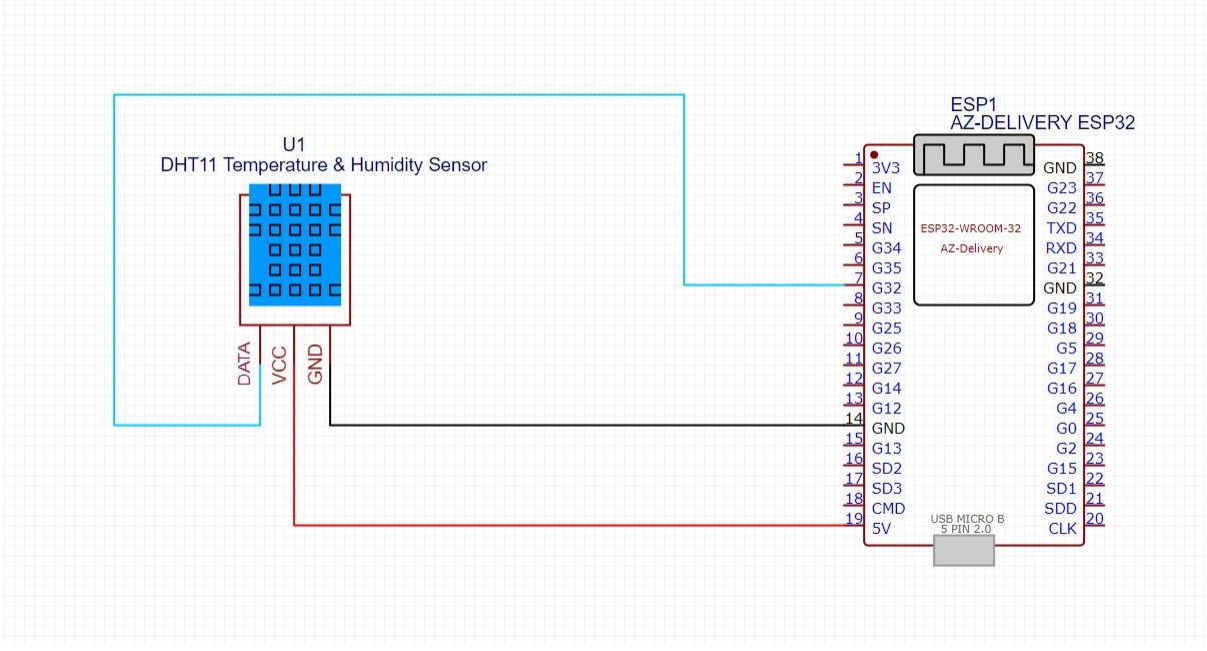


Individual Code For MQ135 Sensor.

```
const int MQ_PIN = 34;
void setup() {
  Serial.begin(9600);
  pinMode(MQ_PIN, INPUT
}
void loop() {
```

```
  int sensorValue = analogRead(MQ_PIN);
  float voltage = sensorValue * (3.3 / 4095.0);
  float ppm = map(voltage, 0.1, 3.0, 0, 50
  // Print the gas concentration
  Serial.print("Gas Concentration (PPM): ");
  Serial.println(ppm);
  delay(1000); // Wait for some time before reading again
}
```

ESP32 interfacing with DHT 11 sensor.

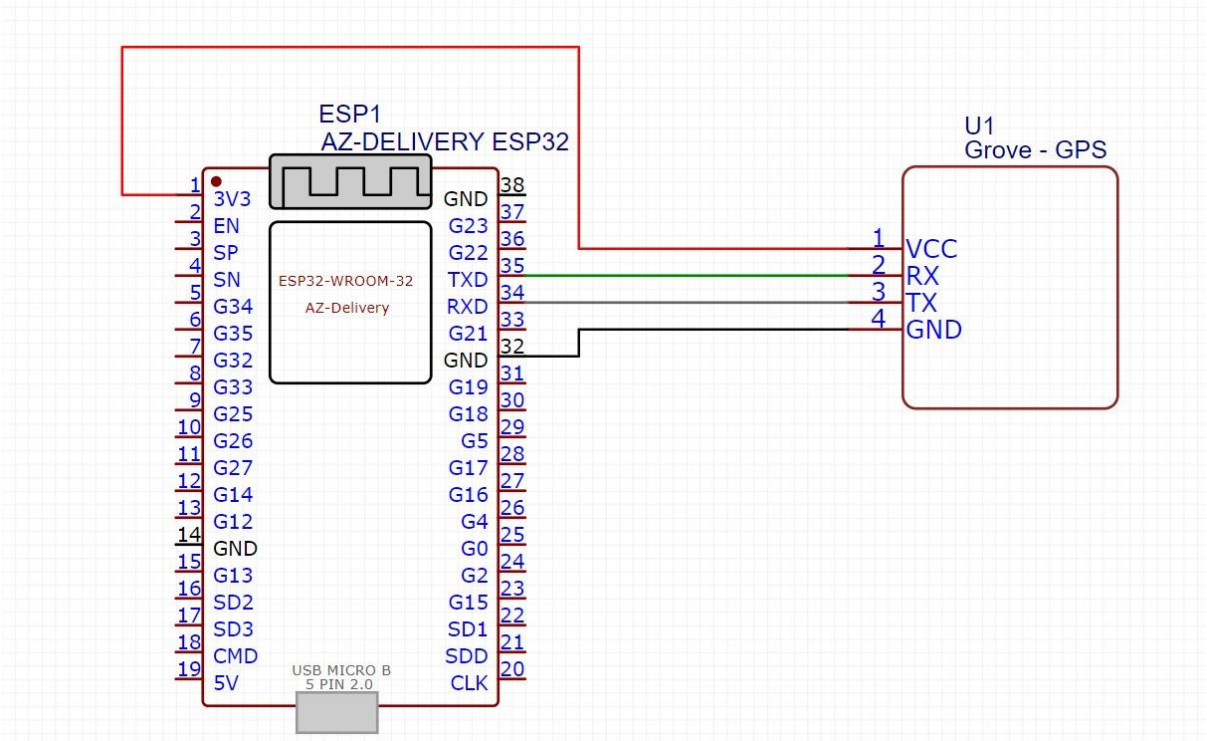


Individual Code For DHT11 Sensor.

```
#include <DHT.h>
#define DHTPIN 4      // Digital pin connected to the DHT sensor
#define DHTTYPE DHT11 // DHT 11
DHT dht(DHTPIN, DHTTYPE);
void setup() {
  Serial.begin(9600);
  Serial.println("DHT11 Sensor Test");
  dht.begin();
}
void loop() {
  delay(2000); // Delay between sensor readings
  float humidity = dht.readHumidity();
```

```
  float temperature = dht.readTemperature();
  // Check if any reads failed and exit early (to try again).
  if (isnan(humidity) || isnan(temperature)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }
  Serial.print("Humidity: ");
  Serial.print(humidity);
  Serial.print(" %\t");
  Serial.print("Temperature: ");
  Serial.print(temperature);
  Serial.println(" °C");
}
```

ESP32 interfacing with GPS module.



Individual Code For GPS MODULE.

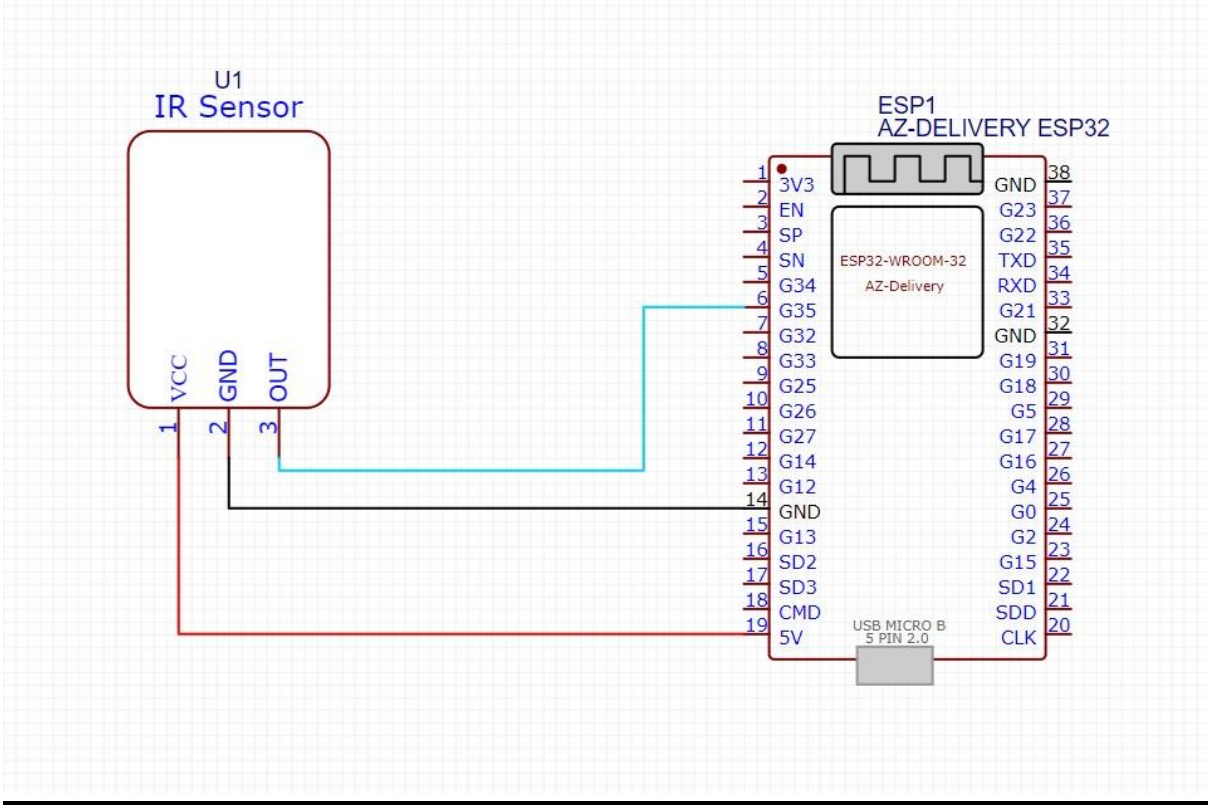
```
#include <TinyGPSPlus.h>
#include <SoftwareSerial.h>
/*
   This sample sketch demonstrates the normal use of a TinyGPSPlus (TinyGPSPlus) object.
   It requires the use of SoftwareSerial, and assumes that you have a
   4800-baud serial GPS device hooked up on pins 4(rx) and 3(tx).
*/
static const int RXPin = 5, TXPin = 4;
static const uint32_t GPSBaud = 4800;
```

```
// The TinyGPSPlus object
TinyGPSPlus gps;
// The serial connection to the GPS device
SoftwareSerial ss(RXPin, TXPin);
void setup() {
  Serial.begin(115200);
  ss.begin(GPSBaud);
  Serial.println();
}

void loop() {
  // This sketch displays information every time a new sentence is correctly encoded.
  while (ss.available() > 0)
    if (gps.encode(ss.read()))
      displayInfo();

  if (millis() > 5000 && gps.charsProcessed() < 10) {
    Serial.println(F("No GPS detected: check wiring."));
    while (true)
      ;
  }
}
void displayInfo() {
  Serial.print(F("Location: "));
  if (gps.location.isValid()) {
    Serial.print(gps.location.lat(), 6);
    Serial.print(F(","));
```

```
  Serial.print(gps.location.lng(), 6);
 } else {
  Serial.print(F("INVALID"));
 }
 Serial.print(F("  Date/Time: "));
 if (gps.date.isValid()) {
  Serial.print(gps.date.month());
  Serial.print(F("/"));
  Serial.print(gps.date.day());
  Serial.print(F("/"));
  Serial.print(gps.date.year());
 } else {
  Serial.print(F("INVALID"));
 }
 Serial.print(F(" "));
 if (gps.time.isValid()) {
  if (gps.time.hour() < 10) Serial.print(F("0"));
  Serial.print(gps.time.hour());
  Serial.print(F(":"));
  if (gps.time.minute() < 10) Serial.print(F("0"));
  Serial.print(gps.time.minute());
  Serial.print(F(":"));
  if (gps.time.second() < 10) Serial.print(F("0"));
  Serial.print(gps.time.second());
  Serial.print(F("."));
  if (gps.time.centisecond() < 10) Serial.print(F("0"));
  Serial.print(gps.time.centisecond());
```

```
  } else {
    Serial.print(F("INVALID"));
  }
  Serial.println();
}
```

ESP32 interfacing with IR sensor.

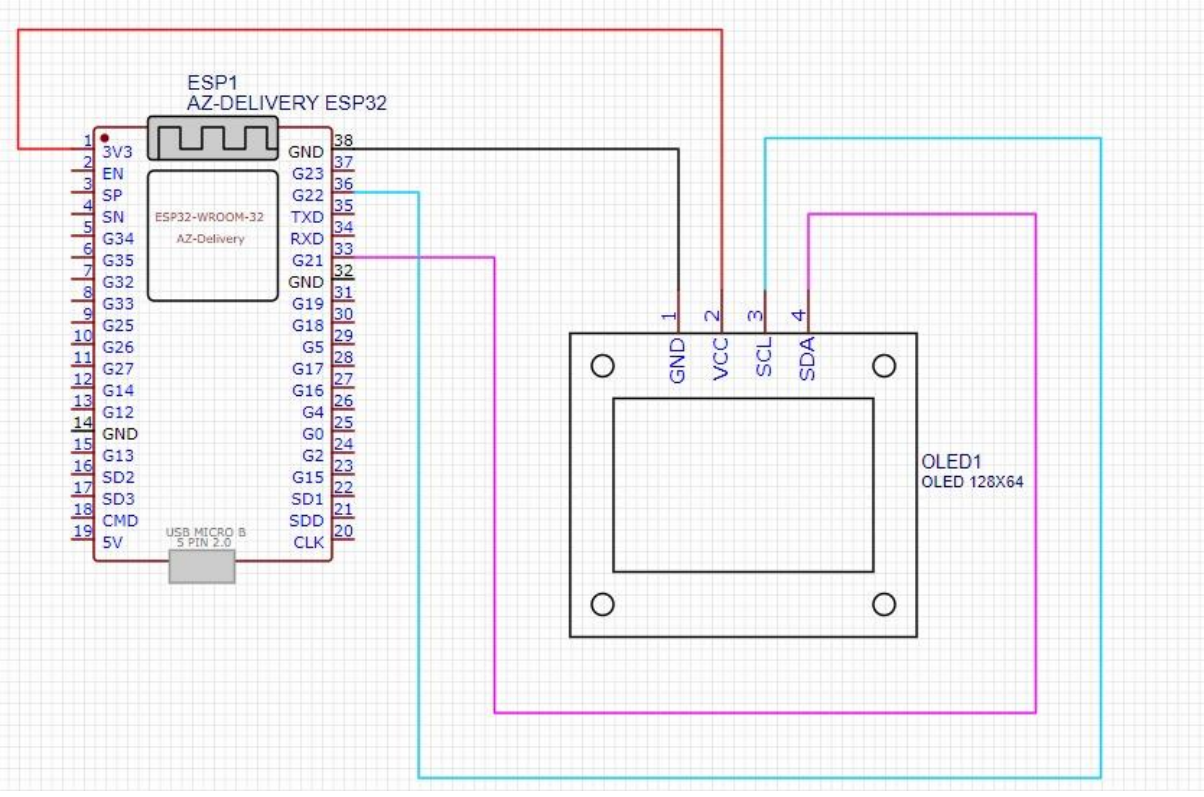


Individual Code Of IR sensor.

```
// Define pin number for IR sensor
#define IR_SENSOR_PIN 4
void setup() {
  Serial.begin(9600); // Initialize serial communication
  pinMode(IR_SENSOR_PIN, INPUT); // Set IR sensor pin as input
}
void loop() {
  // Read the state of the IR sensor
  int irSensorValue = digitalRead(IR_SENSOR_PIN);
```

```
  // Print the sensor value to the serial monitor
  Serial.print("IR Sensor Value: ");
  Serial.println(irSensorValue);
  // Wait for a short duration before reading again
  delay(100);
}
```

ESP32 interfacing with OLED.

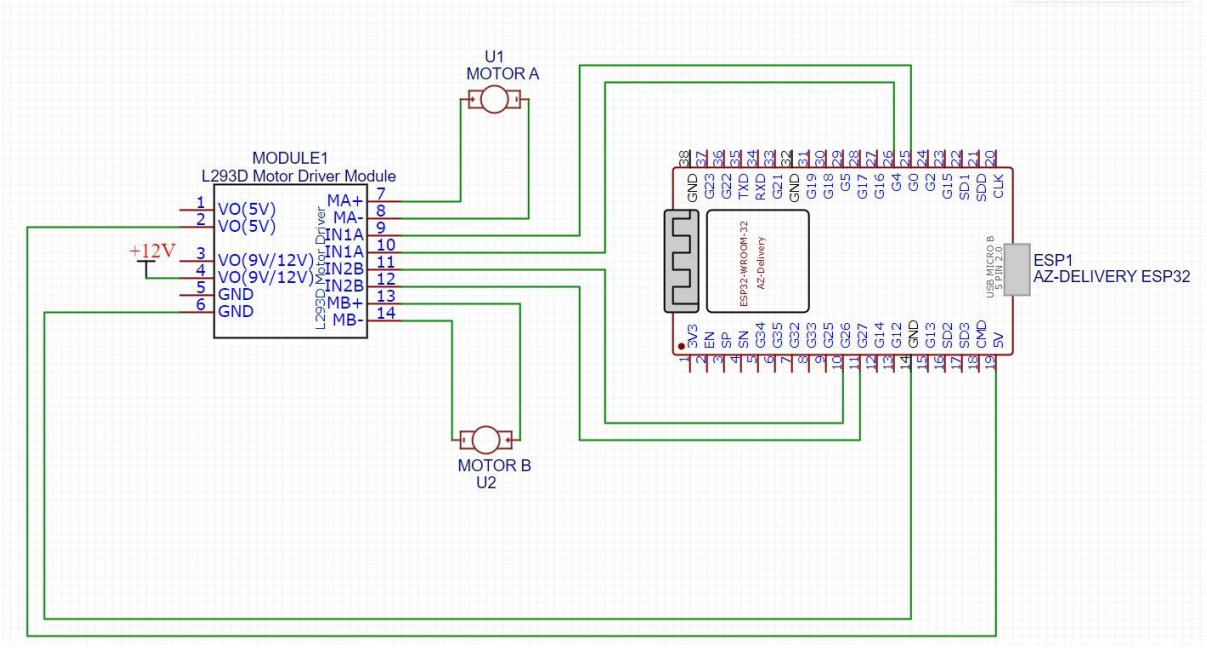


Individual Code Of O-LED.

```
#include <U8g2lib.h>

// Define the I2C pins

#define SDA_PIN 21

#define SCL_PIN 22

// Create a U8g2 object for the SSD1306 128x64 OLED display

U8G2_SH1106_128X64_NONAME_F_HW_I2C u8g2(U8G2_R0, /* reset=*/
U8X8_PIN_NONE, SCL_PIN, SDA_PIN);

void setup() {

  Serial.begin(9600);
```

```
  // initialize the OLED display
  u8g2.begin();
}
void loop() {
  // Clear the display buffer
  u8g2.clearBuffer();
  // Set font and position
  u8g2.setFont(u8g2_font_ncenB08_tr);
  u8g2.setCursor(0, 10);
  // Print text
  u8g2.print("iot based hazards detection system!");
  // Send the buffer to the display
  u8g2.sendBuffer();
  // Pause for 2 seconds
  delay(2000);
}
```

ESP32 interfacing with L293D motor driver.



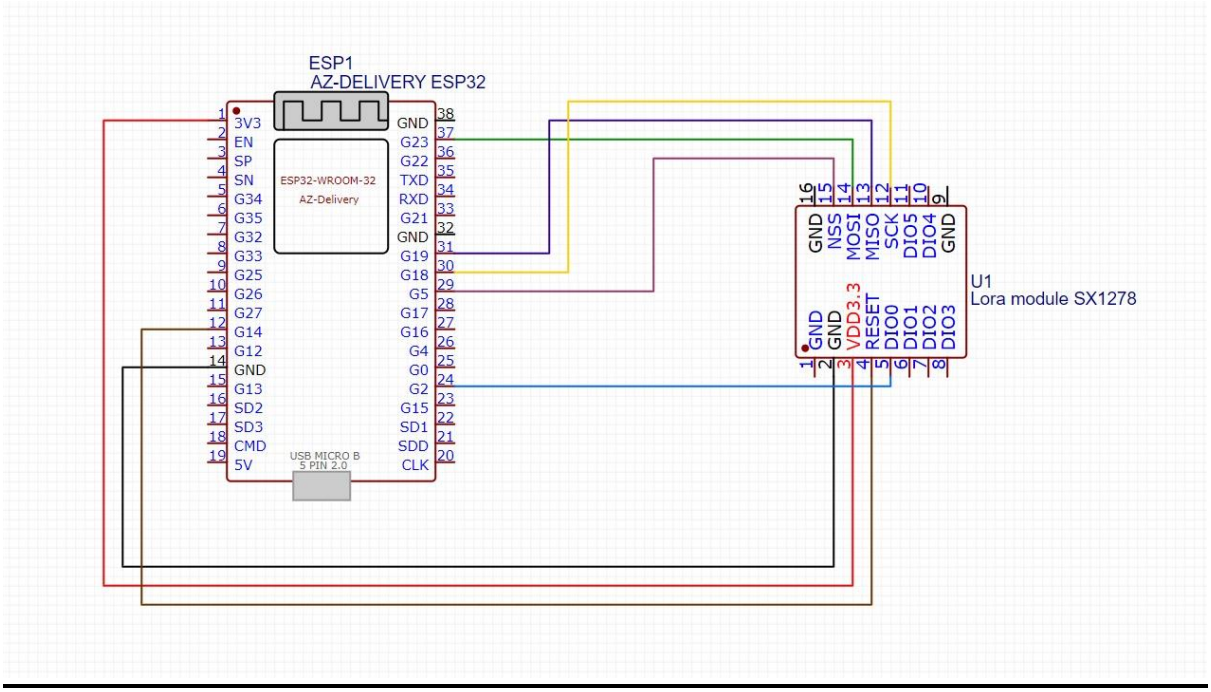
Individual Code Of L293D.

```
// Define motor control pins
const int motorPin1 = 26; // Connected to L293D pin 3
const int motorPin2 = 27; // Connected to L293D pin 4

void setup() {
  // Initialize motor control pins as outputs
  pinMode(motorPin1, OUTPUT);
  pinMode(motorPin2, OUTPUT);

  // Set initial motor state (motor stopped)
```

```
  digitalWrite(motorPin1, LOW);
  digitalWrite(motorPin2, LOW);
}

void loop() {
  // Run the motor forward for 2 seconds
  motorForward();
  delay(2000);

  // Stop the motor for 1 second
  motorStop();
  delay(1000);

  // Run the motor in reverse for 2 seconds
  motorReverse();
  delay(2000);

  // Stop the motor for 1 second
  motorStop();
  delay(1000);
}

// Function to drive the motor forward
void motorForward() {
  digitalWrite(motorPin1, HIGH);
  digitalWrite(motorPin2, LOW);
}
```

```
// Function to stop the motor
void motorStop() {
  digitalWrite(motorPin1, LOW);
  digitalWrite(motorPin2, LOW);
}

// Function to drive the motor in reverse
void motorReverse() {
  digitalWrite(motorPin1, LOW);
  digitalWrite(motorPin2, HIGH);
}
```

ESP32 interfacing with LoRa Module SX1278.



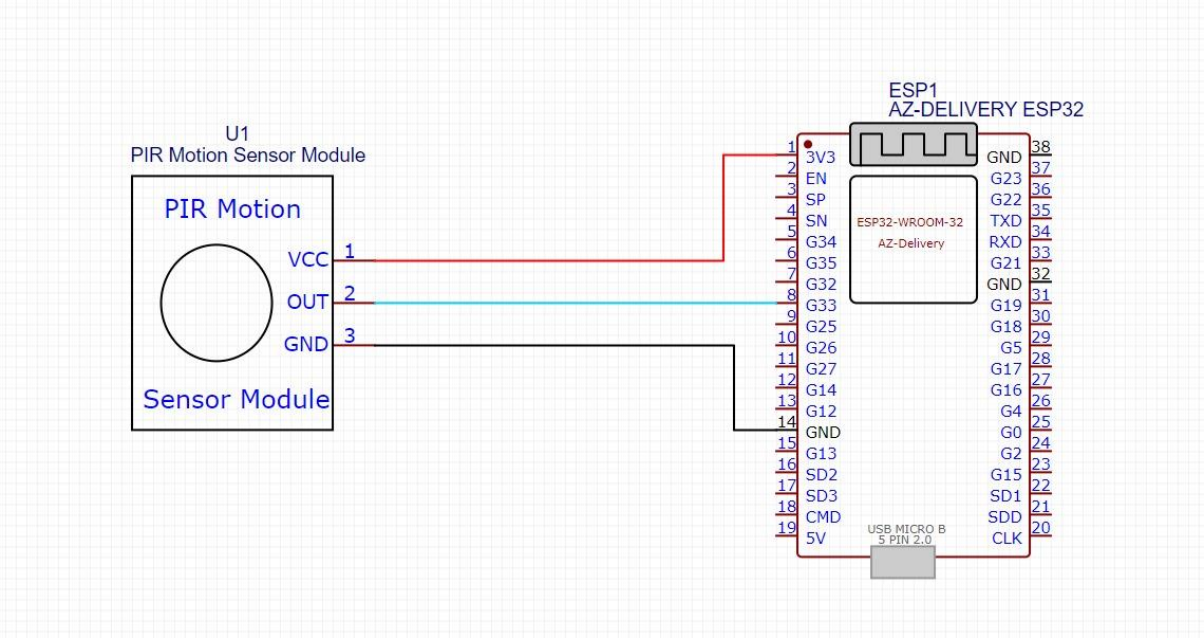
TX code of LoRa.

```
#include <LoRa.h>
#include <SPI.h>

#define ss 5
#define rst 14
#define dio0 2
```

```
int counter = 0;

void setup() {
  Serial.begin(115200);
  while (!Serial)
    ;
  Serial.println("LoRa Sender");

  LoRa.setPins(ss, rst, dio0);  //setup LoRa transceiver module

  while (!LoRa.begin(433E6))  //433E6 - Asia, 866E6 - Europe, 915E6
- North America
  {
    Serial.println(".");
    delay(500);
  }
  LoRa.setSyncWord(0xA5);
  Serial.println("LoRa Initializing OK!");
}

void loop() {
  Serial.print("Sending packet: ");
  Serial.println(counter);

  LoRa.beginPacket();  //Send LoRa packet to receiver
```

```
  LoRa.print("hello ");
  LoRa.print(counter);
  LoRa.endPacket();

  counter++;

  delay(10000);
}
```

ESP32 interfacing with PIR sensor.

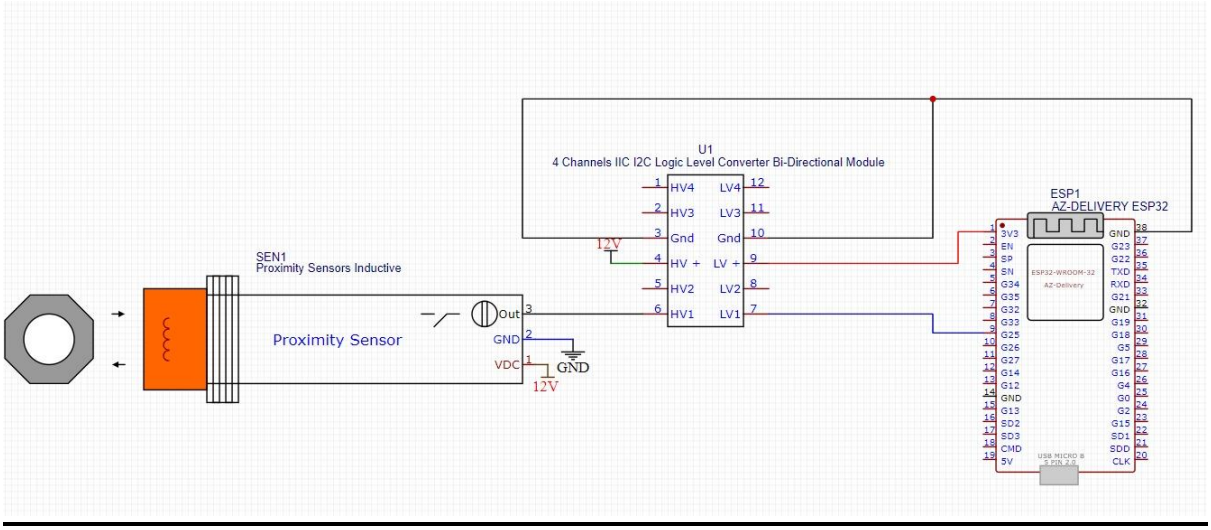


Individual Code Of PIR sensor.

```
// Define pin number for PIR sensor
#define PIR_SENSOR_PIN 2
void setup() {
  Serial.begin(9600); // Initialize serial communication
  pinMode(PIR_SENSOR_PIN, INPUT); // Set PIR sensor pin as input
}
void loop() {
  // Read the state of the PIR sensor
  int pirSensorValue = digitalRead(PIR_SENSOR_PIN);
```

```
  // Print the sensor value to the serial monitor
  Serial.print("PIR Sensor Value: ");
  Serial.println(pirSensorValue);
  // Wait for a short duration before reading again
  delay(100);
}
```

ESP32 interfacing with Inductive Proximity Sensor.

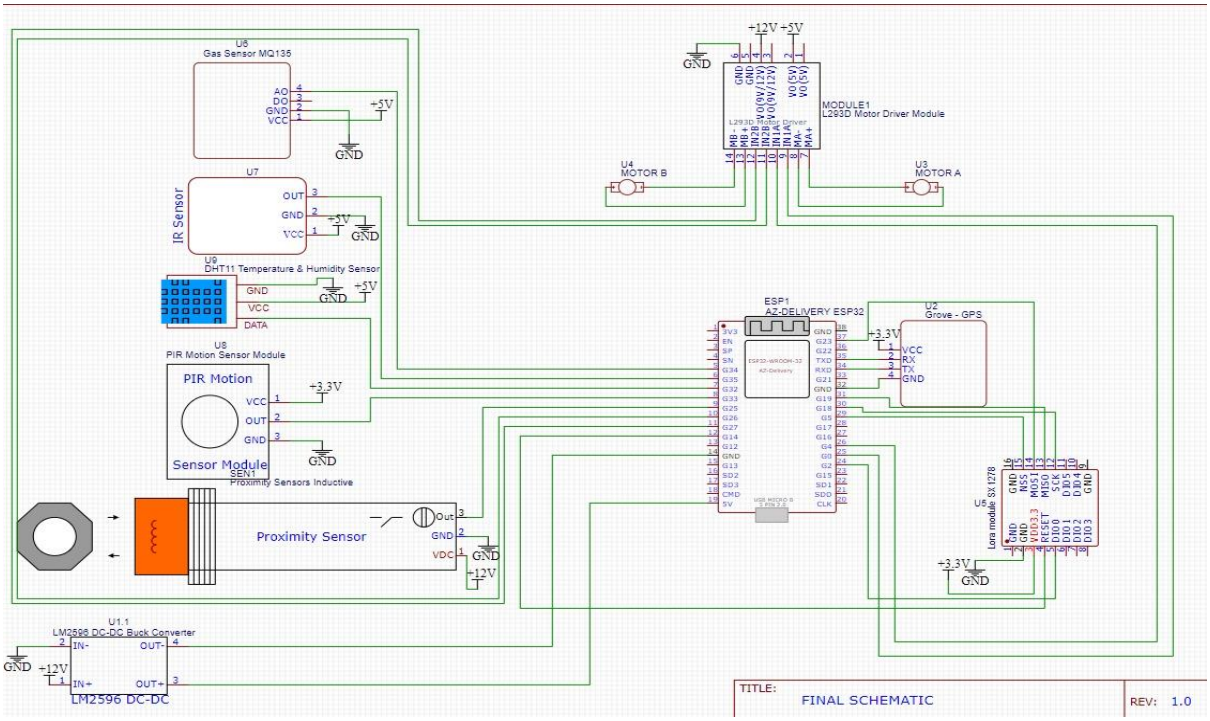


Individual Code Of Proximity Sensor.

```
const int sensorPin = 25; // GPIO pin connected to LV1 of the logic level converter

void setup() {
  Serial.begin(9600);
  pinMode(sensorPin, INPUT);
}

void loop() {
  // Read the sensor value
  int sensorValue = digitalRead(sensorPin);
  // Check if a metal object is detected or not
  if (sensorValue == HIGH) {
```

```
    Serial.println("Metal Detected (1)");
  } else {
    Serial.println("No Metal Detected (0)");
  }
  delay(1000); // Delay for readability, adjust as needed
}
```

Final Sender Schematic And Final Sender Code:-




```
#include <DHT.h>
#include <LoRa.h>
#include <SPI.h>
#include <WiFi.h>
#include <ThingSpeak.h>
#include <BluetoothSerial.h>
// WiFi credentials
const char *ssid = "ESP_IoT_Lab";
const char *password = "ESP32_IoT";
// ThingSpeak channel settings
```

```
const unsigned long channelID = 2517267;
const char *apiKey = "2ODFRK0BDA672KCS";
// Define pin numbers for sensors
#define DHTPIN 32          // Digital pin connected to the DHT sensor
#define PIR_SENSOR_PIN 33  // Pin number for PIR sensor
#define IR_SENSOR_PIN 35   // Pin number for IR sensor
#define MQ_PIN 34          // Pin number for MQ135 sensor
#define SENSOR_PIN 25      // Pin number for proximity sensor
#define motor1Input1A 4    // Motor control pins
#define motor1Input1B 0
#define motor2Input2A 26
#define motor2Input2B 27

WiFiClient client;
BluetoothSerial serialBT;
// Define sensor types
#define DHTTYPE DHT11  // DHT 11

DHT dht(DHTPIN, DHTTYPE);
#define ss 5
#define rst 14
#define dio0 2

void setup() {
  Serial.begin(115200);
  while (!Serial)
    ;
```

```
  Serial.println("Sensors Test");
  // Initialize sensors
  dht.begin();
  pinMode(PIR_SENSOR_PIN, INPUT);
  pinMode(IR_SENSOR_PIN, INPUT);
  pinMode(MQ_PIN, INPUT);
  pinMode(SENSOR_PIN, INPUT);
  // Set motor control pins as outputs
  pinMode(motor1Input1A, OUTPUT);
  pinMode(motor1Input1B, OUTPUT);
  pinMode(motor2Input2A, OUTPUT);
  pinMode(motor2Input2B, OUTPUT);
  // Activate motor 1
  digitalWrite(motor1Input1A, HIGH);
  digitalWrite(motor1Input1B, LOW);
  // Activate motor 2
  digitalWrite(motor2Input2A, HIGH);
  digitalWrite(motor2Input2B, LOW);


  LoRa.setPins(ss, rst, dio0);  // Setup LoRa transceiver module
  while (!LoRa.begin(433E6)) {  // 433E6 - Asia, 866E6 - Europe, 915E6 -
North America
    Serial.println(".");
    delay(500);
  }
  LoRa.setSyncWord(0xA5);
  Serial.println("LoRa Initializing OK!");
```

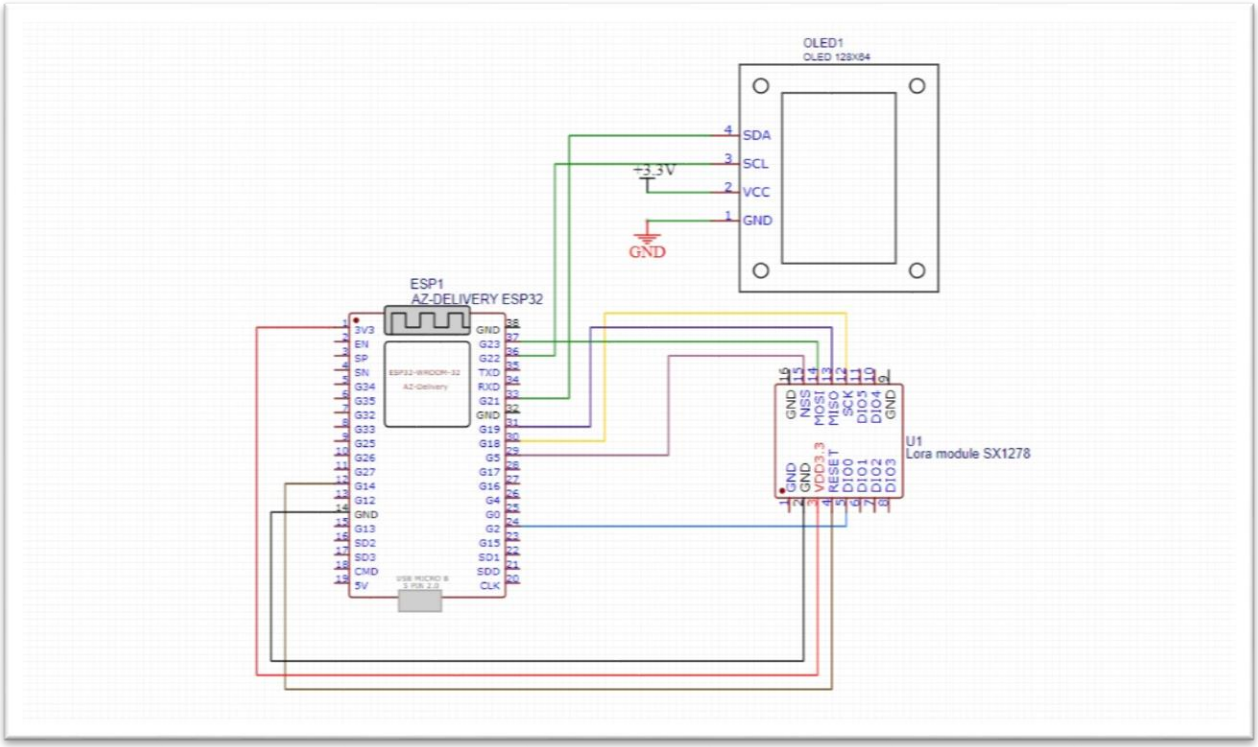
```
  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Initialize ThingSpeak
  ThingSpeak.begin(client);

  // Start Bluetooth serial communication
  serialBT.begin("Tarzan");
}
void loop() {
  // DHT Sensor reading
  float humidity = dht.readHumidity();
  float temperature = dht.readTemperature();
  String message = "Temperature: " + String(temperature) + " °C, Humidity: " +
String(humidity) + " %\t";
  // PIR Sensor reading
  int pirSensorValue = digitalRead(PIR_SENSOR_PIN);
  message += "PIR Sensor Value: " + String(pirSensorValue) + ", ";
  // IR Sensor reading
  int irSensorValue = digitalRead(IR_SENSOR_PIN);
  message += "IR Sensor Value: " + String(irSensorValue) + ", ";
  // MQ135 Sensor reading
  int mqsensorValue = analogRead(MQ_PIN);
```

```
float voltage = mqsensorValue * (3.3 / 4095.0);
float ppm = map(voltage, 0.1, 3.0, 0, 500);
message += "Gas Concentration (PPM): " + String(ppm) + ", ";
// Metal Sensor reading
int metalsensorValue = digitalRead(SENSOR_PIN);
message += "Metal Sensor Value: " + String(metalsensorValue);

Serial.println("Sending packet: " + message);
// Send sensor data to ThingSpeak
ThingSpeak.writeField(channelID, 1, temperature, apiKey);
ThingSpeak.writeField(channelID, 2, humidity, apiKey);
ThingSpeak.writeField(channelID, 3, pirSensorValue, apiKey);
ThingSpeak.writeField(channelID, 4, irSensorValue, apiKey);
ThingSpeak.writeField(channelID, 5, ppm, apiKey);
ThingSpeak.writeField(channelID, 6, metalsensorValue, apiKey);
// LoRa communication
LoRa.beginPacket();  // Send LoRa packet to receiver
LoRa.print(message);
LoRa.endPacket();
// Bluetooth motor control
if (serialBT.available()) {
  char cmd = serialBT.read();
  switch (cmd) {
    case 'F':  // Forward
      digitalWrite(motor1Input1A, HIGH);
      digitalWrite(motor1Input1B, LOW);
      digitalWrite(motor2Input2A, HIGH);
```

```
  digitalWrite(motor2Input2B, LOW);
  break;
case 'B':  // Backward
  digitalWrite(motor1Input1A, LOW);
  digitalWrite(motor1Input1B, HIGH);
  digitalWrite(motor2Input2A, LOW);
  digitalWrite(motor2Input2B, HIGH);
  break;
case 'L':  // Left
  digitalWrite(motor1Input1A, LOW);
  digitalWrite(motor1Input1B, HIGH);
  digitalWrite(motor2Input2A, HIGH);
  digitalWrite(motor2Input2B, LOW);
  break;
case 'R':  // Right
  digitalWrite(motor1Input1A, HIGH);
  digitalWrite(motor1Input1B, LOW);
  digitalWrite(motor2Input2A, LOW);
  digitalWrite(motor2Input2B, HIGH);
  break;
case 'S':  // Stop
  digitalWrite(motor1Input1A, LOW);
  digitalWrite(motor1Input1B, LOW);
  digitalWrite(motor2Input2A, LOW);
  digitalWrite(motor2Input2B, LOW);
  break;
default:
```

```
      break;
    }
  }
  delay(15000);  // Send data every 15 seconds (ThingSpeak limit)
}
```

Final Receiver Schematic & Final Receiver Code:



```
#include <LoRa.h>
#include <U8g2lib.h>
#include <SPI.h>

#define ss 5
#define rst 14
#define dio0 2

#define SDA_PIN 21
#define SCL_PIN 22
```

```
U8G2_SH1106_128X64_NONAME_F_HW_I2C u8g2(U8G2_R0, /* reset=*/
U8X8_PIN_NONE, SCL_PIN, SDA_PIN);


void setup() {
  Serial.begin(115200);
  while (!Serial);
  Serial.println("LoRa Receiver");


  LoRa.setPins(ss, rst, dio0);    // Setup LoRa transceiver module


  while (!LoRa.begin(433E6)) {    // 433E6 - Asia, 866E6 - Europe, 915E6 -
North America
    Serial.println(".");
    delay(500);
  }
  LoRa.setSyncWord(0xA5);
  Serial.println("LoRa Initializing OK!");


  // Initialize the OLED display
  u8g2.begin();
}


void loop() {
  int packetSize = LoRa.parsePacket();    // Try to parse packet
  if (packetSize) {


    Serial.print("Received packet: '");
```

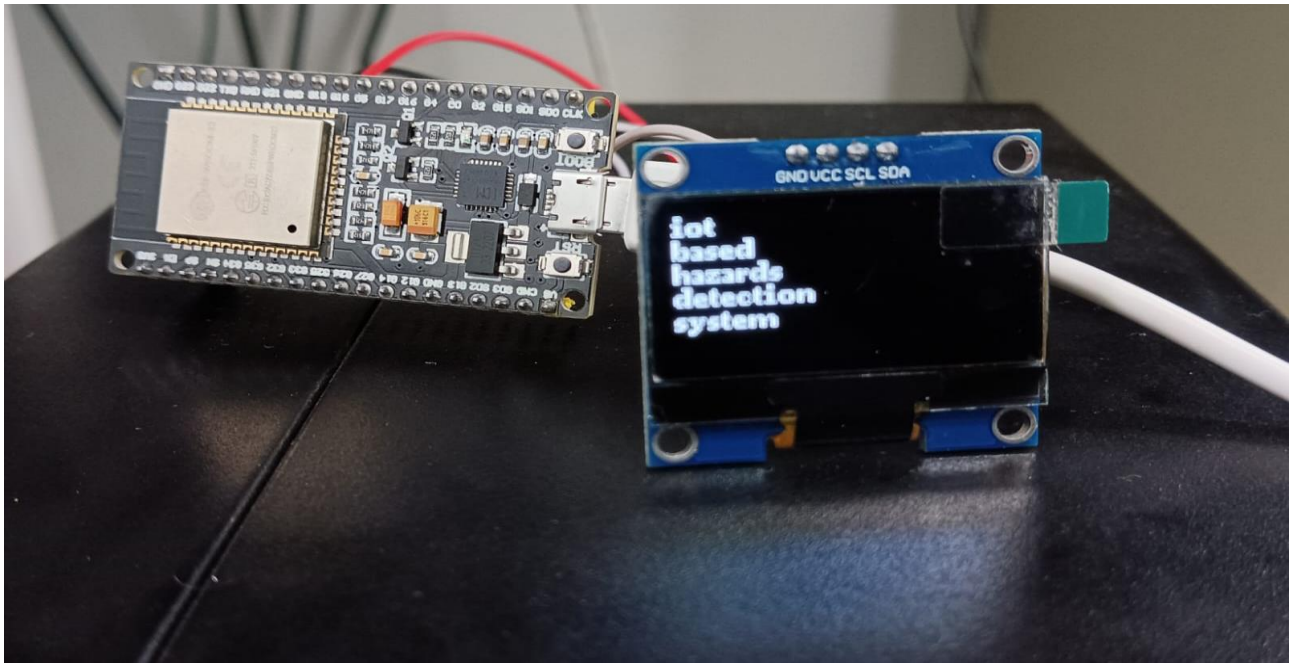
```
while (LoRa.available()) {              // Read packet
  String LoRaData = LoRa.readString();
  Serial.print(LoRaData);

  // Clear the display buffer
  u8g2.clearBuffer();

  // Set font and position
  u8g2.setFont(u8g2_font_ncenB08_tr);
  u8g2.setCursor(0, 10);

  // Print the received LoRa data
  u8g2.print(LoRaData);

       u8g2.setCursor(0, 20);
       u8g2.print(humidity);
       u8g2.setCursor(0, 30);
       u8g2.print(Pir sensor value);
       u8g2.setCursor(0, 40);
       u8g2.print(Gas concentration);
       u8g2.setCursor(0, 50);
       u8g2.print(Metal detection):
       u8g2.setCursor(0, 60);
       u8g2.print(Lat-21.123570, Lng-79.049217);

  // Send the buffer to the display
```

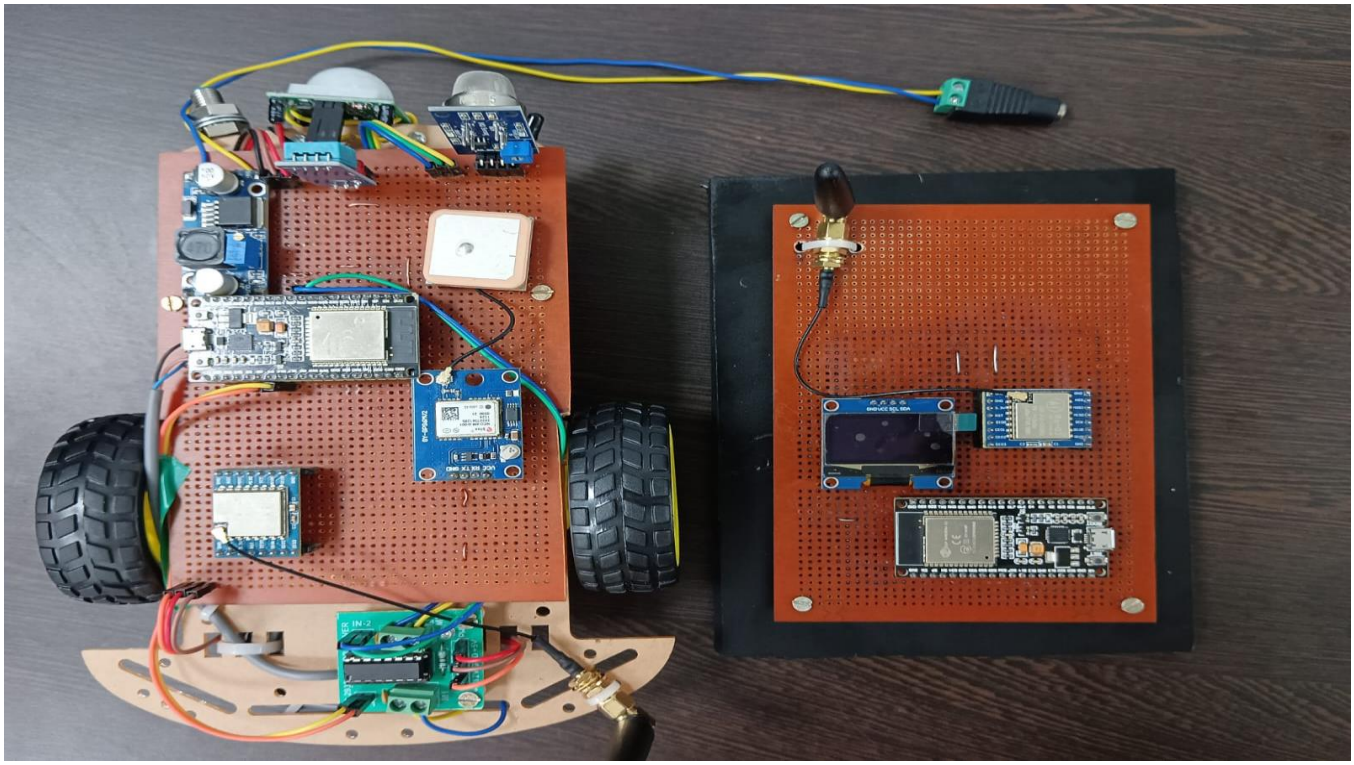
```
      u8g2.sendBuffer();
    }
    Serial.print("' with RSSI ");         // Print RSSI of packet
    Serial.println(LoRa.packetRssi());
  }
}
```

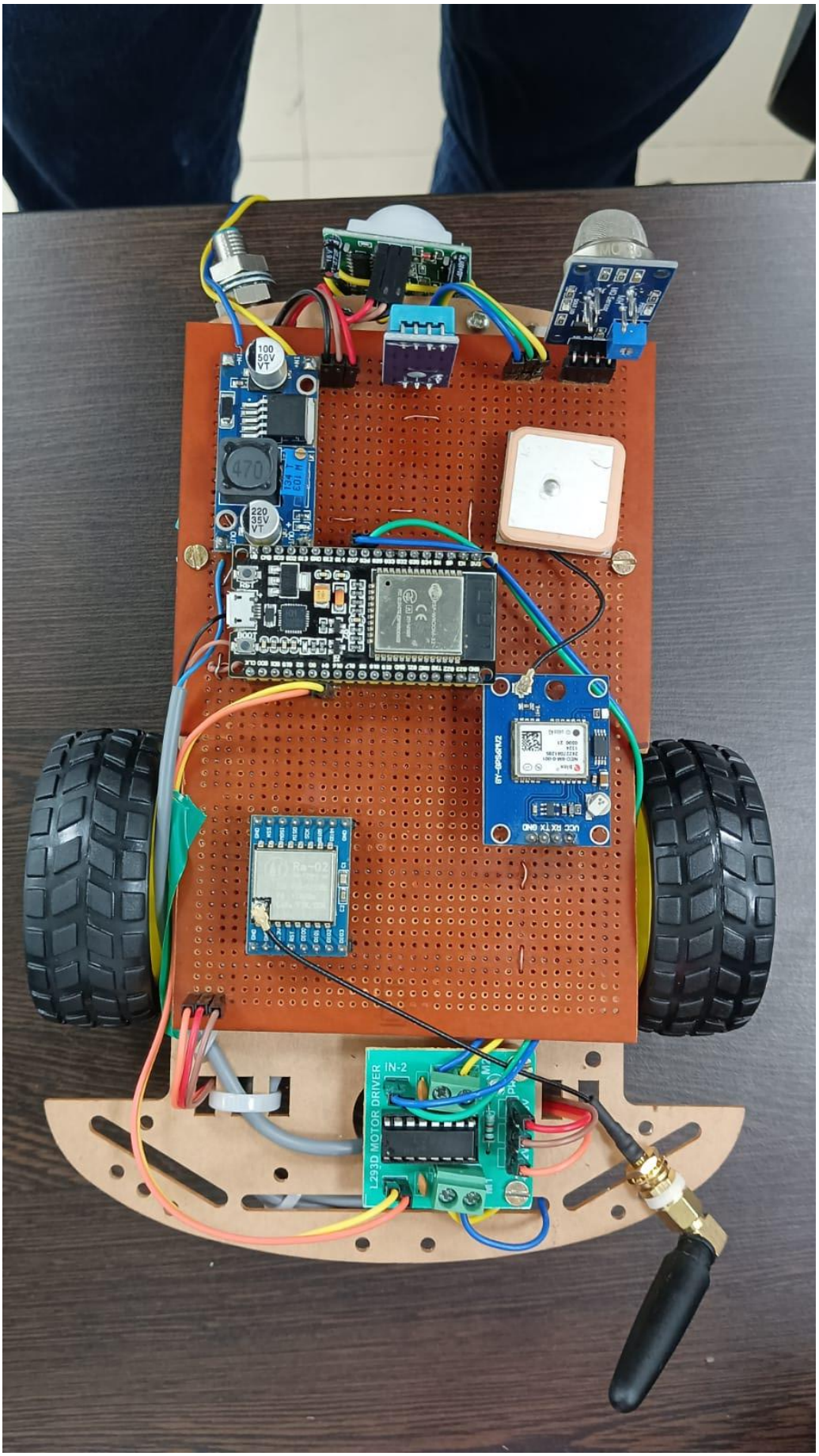
Future Aspects And Modifications:

1. Detecting air quality
2. Radiation levels
3. Seismic activity.
4. Chain wheels for travelling in different terrains.
5. If we use high defined temperature sensor then we can use it in the harsh temperature regions too, as DHT11 has 0 c to 50 c.
6. Even biological contaminants.
7. We can also run this project on Li-ion batteries.
8. Making the waterproof body for the vehicle.
9. We can also add a camera on it.
10. In case of danger we can self destructive mode on it.

Photos Of Phases:






100
50V
VT
470
220
35V
VT
L293D MOTOR DRIVER
IN-2
GY-GPS6MV2
VCC RX TX GND

List Of References:

- Chat gpt
- Github
- https://www.alldatasheet.com/view.jsp?Searchword=LM324&gad_source=1&gclid=CjwKCAjw26KxBhBDEiwAu6KXt2ziqawrDis5gOUGfAWnMg5sNwVEwx-L_jWsUW0VOzdMXfmaUk1aYRoC7WEQAvD_BwE
- https://www.researchgate.net/publication/340693942_Design_of_IoT_Based_Multiple_Hazards_Detection_and_Alarming_System
- Easyeda
- Circuit Digest