Vehicular Collision Detection System

Problem Domain:

Vehicular and robotic systems (ie: autonomous cars, robotic food delivery) operate in dynamic environments where the risk of collisions is high. Vehicles often interact with other vehicles, pedestrians, and various traffic obstacles (cones, medians, etc). Robotic systems that are deployed in urban environments interact with humans, nature, cars, and more.

Goals/Objectives:

1. Detect a collision in real-time through Collision Sensor, IR Transmitter, and Thin-film Pressure Sensor and process the data through an ESP32.
    a. Detect incoming/possible collisions through sensors
2. After a collision trigger an alert via an LED or Buzzer
3. Alert users through a mobile application
    a. Send alerts quickly (as low latency as possible)

Initial Functionalities:

1. Read and process data from the Collision Sensor, IR Transmitter, and Thin-film Pressure Sensor.
    a. Determine a way to store and send data from the Collision Sensor, IR Transmitter, and Thin-film Pressure Sensor.
2. Create a basic algorithm that determines if a collision has occurred based on data from the Collision Sensor, IR Transmitter, and Thin-film Pressure Sensor that will trigger a buzzer and send an alert.
3. Implement data transmission and storage (possibly for mobile application), deciding between Wi-Fi, Bluetooth, or other protocols for efficient communication.
4. Log collision data for debugging, analysis, and improving detection accuracy.
5. Develop a method to classify collision severity (e.g., minor vs. major impact) and distinguish between object detection and direct impact.
6. Establish collision threshold tuning for collision detection, allowing adaptability to different objects.
7. Optimize for low-latency alerts, ensuring quick response times from sensor input to notification.

8. Define whether data processing occurs entirely on the ESP32 or involves external computing (e.g., mobile app or cloud-based processing).

Method of Communicating Data: MQTT Connection

Justification:

MQTT is a lightweight messaging protocol specifically designed for IoT. It guarantees reliable real-time communication with minimal bandwidth and setup. Realistically the user will not be in the vehicle when a crash occurs. Therefore, a bluetooth connection would not work since there is a maximum distance the user can be from the system before the connection is broken. Using MQTT guarantees that as long as the user has a data or WiFi connection then they should receive an alert.

Setting up MQTT on ESP32:
https://www.emqx.com/en/blog/esp32-connects-to-the-free-public-mqtt-broker

Data Outputs:
Thin-film Pressure Sensor: Converts pressure to Analog Voltage, range: 0-4096
Collision Detector: conditional value that reads true when not pressed and false when pressed.
IR Transmitter: Emits an infrared light (max distance 1.3m). Not sure how to use this without an IR Receiver as the transmitter only emits the infrared light but we cannot gain any information without a receiver.

Algorithm:
To keep the latency as low as possible we should run the algorithm on the ESP32, unless the algorithm is too complex. This will remove the need to send a JSON full of sensor data from the ESP32 over WiFi/MQTT to the application. We would only send the alert JSON over WiFi/MQTT.

SHFTR.IO CREDENTIALS

Username: emry.hankins001@umb.edu
Pass: CollisionDetection!@

Arduino Environment Setup

Properly setting up the Arduino IDE environment allows the user to program with the ESP32.

1. Download Arduino IDE
2. In File > Preferences (or Ctrl+Comma), enter the given link into "Additional boards manager URLs." This adds EPS32 support for the Arduino IDE.
    a. https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json
3. To select the EPS as your board, go to : Tools > Board > Boards Manager (or Ctrl+Shift+B). Search for "eps32" and install "esp32 by Espressif Systems"
4. Now under Tools > Board > eps32, look for "NodeMCU-32S" and select it. That is the board we were given.
5. When you connect the ESP32 to your computer with a USB cable, go to Tools > Port, and select the port the USB is connected to.
6. By this point, your Arduino environment is set up for basic ESP32 use. We still need to set it up to support the IR transmitter and WiFi.

Connecting to the WiFi

Adding support for WiFi allows the ESP32 to connect to the internet and other things connected to the same internet. There are two different ways of connecting to the internet : Blocking and Non-blocking.

With blocking, no code after the function call to connect to the internet will run while attempting to connect to the internet. Once it connects successfully the first time, it'll stop trying to connect and won't connect if it ever disconnects. This is useful for programs that don't run infinitely after launching.

If your program runs forever, blocking is a bad idea. So, you'd use non-blocking to connect to the internet. With non-blocking, code can still run while attempting to connect to the internet so the program isn't blocked while waiting to connect.

Blocking:

1. Include the library with:

```
#include <WiFiMulti.h>
```

2. Define the WiFi's SSID and password with the following:

```
#define WIFI_SSID "WiFi name"
#define WIFI_PASSWORD "WiFi password"
```

3. Initialize a WiFi Multi instance outside of any functions with:

```
WiFiMulti wifimulti;
```

4. Under `void setup()` , add an access point to your WiFi using:

```
wifimulti.addAP(WIFI_SSID, WIFI_PASSWORD);
```

5. Under the access point function call, start a while loop that repeats until the WiFi is connected.

```
while (wifimulti.run() != WL_CONNECTED) {
     delay(100);
}
```

- `WL_CONNECTED` is a status type in the WiFiMulti library that confirms the WiFi is connected
- `wifimulti.run()` returns `WL_CONNECTED` if it successfully connected to the WiFi and `WL_CONNECT_FAILED` if it failed.
- This loop blocks any following code from running, making this method the blocking method

6. To confirm that the Wifi is properly connected, run the following code in the `void loop()` function:

```
digitalWrite(LED_BUILTIN, WiFi.status() == WL_CONNECTED);
```

- This code snippet keeps the built-in LED in the ESP on while the WiFi is connected
- Make sure to initialize the pin in the setup so the `LED_BUILTIN` part works

```
pinMode(LED_BUILTIN, OUTPUT);
```

Non-blocking:

1. Include the library with:

```
#include <WiFi.h>
```

    - Note that the library is different than the one used in blocking
2. Define the WiFi SSID and password in the same way as blocking.

```
#define WIFI_SSID "WiFi name"
#define WIFI_PASSWORD "WiFi password"
```

3. Setup the access point to the WiFi using the following under `void setup()` :

```
WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
```

    - `WiFi.begin()` is a function that runs and keeps running. It detects if the WiFi is not connected and tries to connect to it. It sits idle, waiting for the WiFi to disconnect so it can reconnect.
        - You can stop here or keep following instructions to set up a boolean that's `true` when the WiFi is connected and `false` otherwise
4. Under `void loop()` , check if the WiFi is connected. If it is, great. Set the boolean to `true` and move on. Here is the condition:

```
WiFi.status() == WL_CONNECTED
```

    - Here is where you would turn the LED on if you want a physical way to see that the WiFi is connected
    - Since we don't want this to run every time the loop resets, we can check for `WL_CONNECTED` and if the boolean we made is false. This makes sure that the conditional runs only once since when it succeeds, the boolean is `true` and the condition will now fail.
5. If the WiFi is not connected, set the boolean to `false`.
    - Here, the LED would be either off or blinking depending on what you want it to look like.

To do: WiFi library documentation

- Only need to document used functions in library

ESP32 and Other Parts

For this project, we are given:

- An ESP32
    - Microcontroller
- A breadboard
    - Allows easy connections between ESP32 and sensors
- A collision sensor
    - Detects collision with a limit switch
- A thin-film sensor
    - Detects collision by returning difference in pressure
- An Ultrasonic Sensor
    - Detects distance from nearest obstacle

ESP32 (Espressif 32)

- The type of EPS32 we are using is the “ESP32 DEVKIT V1”
- The pins we will be using are:
    - 3V3
        - This will provide power to the sensors
        - Provides 600mA of current
    - GND
        - This is the “ground” pin
    - D25
        - This is one of the input/output pins
        - Information taken from the sensors will be read here
        - If there was any information to be sent, it will be sent through these pins
    - D35
        - This is one of the input only pins
        - Information can be taken from here, but it can’t be sent

Collision Sensor

- The collision sensor has 3 pins:
  - S
    - This pin is connected to the input/output parts of the EPS
    - This is where information will be sent to the ESP
  - V
    - This pin is where the power enters the sensor, so it has to be connected to either 3.3V or VIN on the ESP
  - G
    - This pin is supposed to be connected to ground
- This sensor returns 1 if the sensor is not pressed and 0 if the sensor is pressed
- Code to get values on pin:

```
// This reads the value in the sensor
// COLLISION_PIN is the pin the sensor is connected to
int collision = digitalRead(COLLISION_PIN);

// This prints the value onto the serial monitor
Serial.println(collision ? "Collision: Unpressed" : "Collision:
Detected");
```

- The resistance needed for this sensor depends on how much voltage you're using:
  - If the voltage is 3.3V, you can just use the following code to use the built-in resistance:

```
pinMode(COLLISION_PIN, INPUT_PULLUP);
```

  - If the voltage is 5V (or pin VIN), you can use 10kΩ

Thin-film Sensor

- The thin-film sensor has 3 pins like the collision sensor:
  - S
    - This pin is connected to the input/output parts of the EPS
    - This is where information will be sent to the ESP
  - V

    - This pin is where the power enters the sensor
    - With the thin film sensor, it has to be connected to VIN to be able to read the values
  - G
    - This pin is supposed to be connected to ground
- This sensor returns a value from 0-4095 depending on the pressure put on the sensor
- Code to get value on pin:

```
// This gets the value in the pin
// Notice that this is "analog read" rather than "digital" read
int thinfilm = analogRead(THINFILM_PIN);

// This prints the value to the serial monitor
Serial.println("Thinfilm value: " + String(thinfilm));
```

- The resistance needed varies if the readings are too high or too low
  - If the readings are too high, use 20kΩ
  - If the readings are too low, use 5kΩ

Ultrasonic Sensor

- The ultrasonic sensor has 4 pins:
  - Vcc
    - Power enters the sensor through here
  - Trig
    - This pin is the input pin
    - Sends ultrasonic waves
  - Echo
    - This pin is the output pin
    - Reads time it took to receive ultrasonic waves
  - Gnd
    - This is the ground pin
- The sensor returns a value from 0 to
- The code to get the values from pins:

```
digitalWrite(trigPin, LOW);
delayMicroseconds(2);
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);
```

```
// Reads distance measured in centimeters
float distance = pulseIn(echoPin, HIGH) / 58.2;
```

- The resistance needed for this sensor is roughly 330Ω since it can only take 5V power and uses 15mA of current.