TECHNISCHE HOCHSCHULE DEGGENDORF

Master Artificial Intelligence for Smart Sensors/ Actuators

"Case Study- Smart Sensors & Actuators SS/22"

Written Report

With the title of

"Rear and Side Detection for Bicycles"

By

Name	Matr.no
Aayush Surana	12200044
Jaydip Borad	12203473
Dharmesh Patel	12203573
Milind	12200002

Lecturer & Supervisor – Dr.-Ing. Prof. Jürgen Wittmann

Content

Acknowledgement	2
Abstract	3
Timeline	6
Project Concept	7
Sensor’s description	10
Arduino code explanation	11
Mobile App: Front End	12
Mobile App: Back End	18
Mobile App: Connectivity, Features and Storage	20
Mobile App: APK Implementation	23
Results	24
Future scope of the project	27
Appendix: A. Arduino code B. ESP32 code C. Mobile Application’s back-end code D. Schematic drawings	34
Literature	35

Acknowledgement

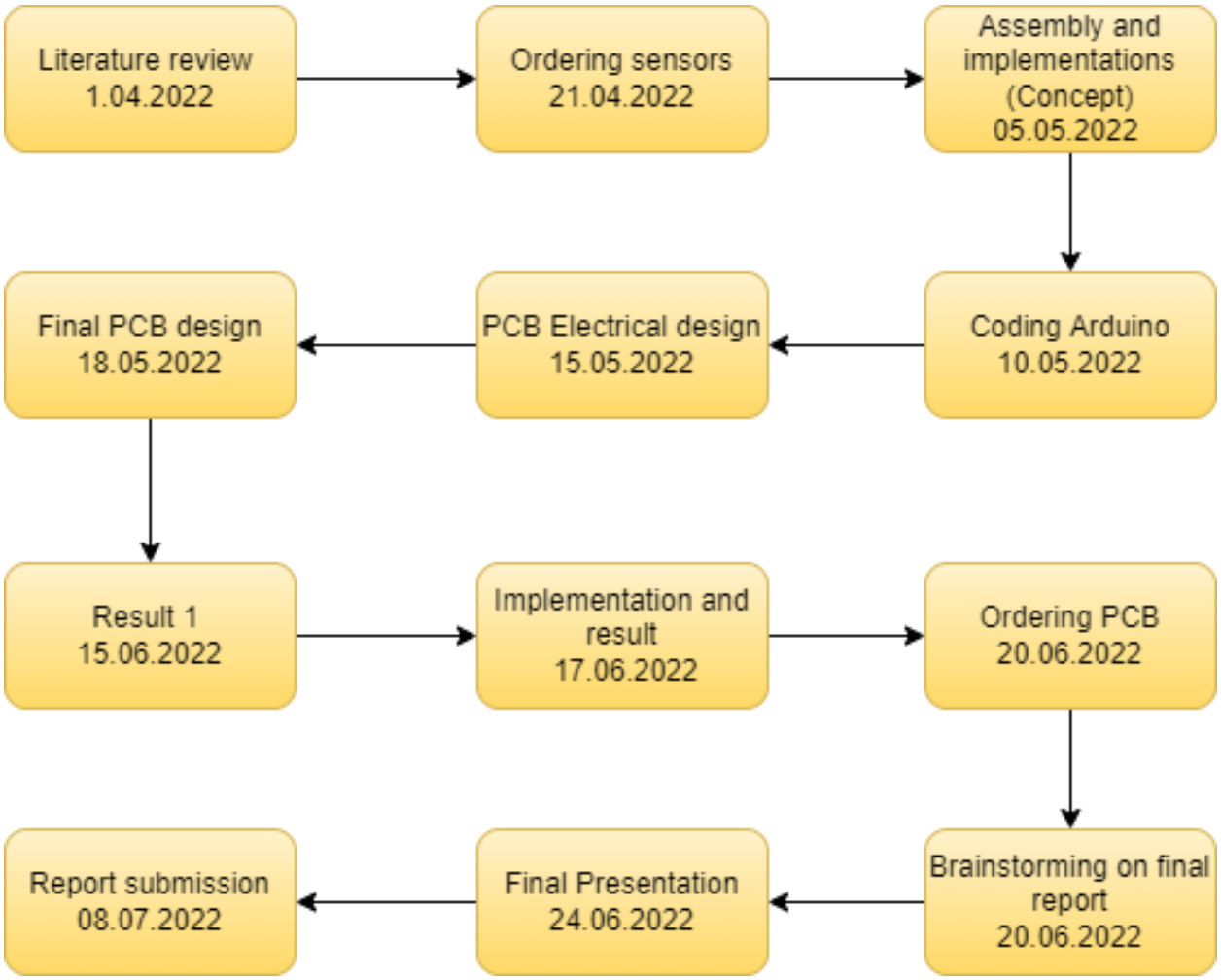
We are grateful because we managed to complete the case study sensors and actuators in the rear and side detection for bicycles with the given time and guidance from Prof. Jürgen Wittmann at Technische Hochschule Deggendorf (Technology Campus - Cham). This case study could not be completed without the effort and cooperation of prof. Jürgen Wittmann for the guidance and encouragement in finishing the case study sensors and actuators project.

Abstract

Nowadays people riding bicycles on the roadside face an oncoming threat. Based on the statistics on bicycle accidents in total, 129,207 two-wheeled vehicle users were engaged in incidents on German roads in 2019, 4.5 percent less than in 2018. The number of motorcycle accidents declined by about 9%, from 31,419 to 27,927, while the number of deaths reduced from 619 to 542. In 2019, there were 13,925 accidents involving two-wheeled motor vehicles with insurance plates. Our goal with this project is to create a device that can warn a cyclist of an impending harm. Following extensive study, our device can detect an approaching vehicle from the left, right, or rear side within 5m and notify its speed and existence to the ride on their phone with minimal errors. The rider can adjust the device scanning radius up to 7ms.

.

Timeline

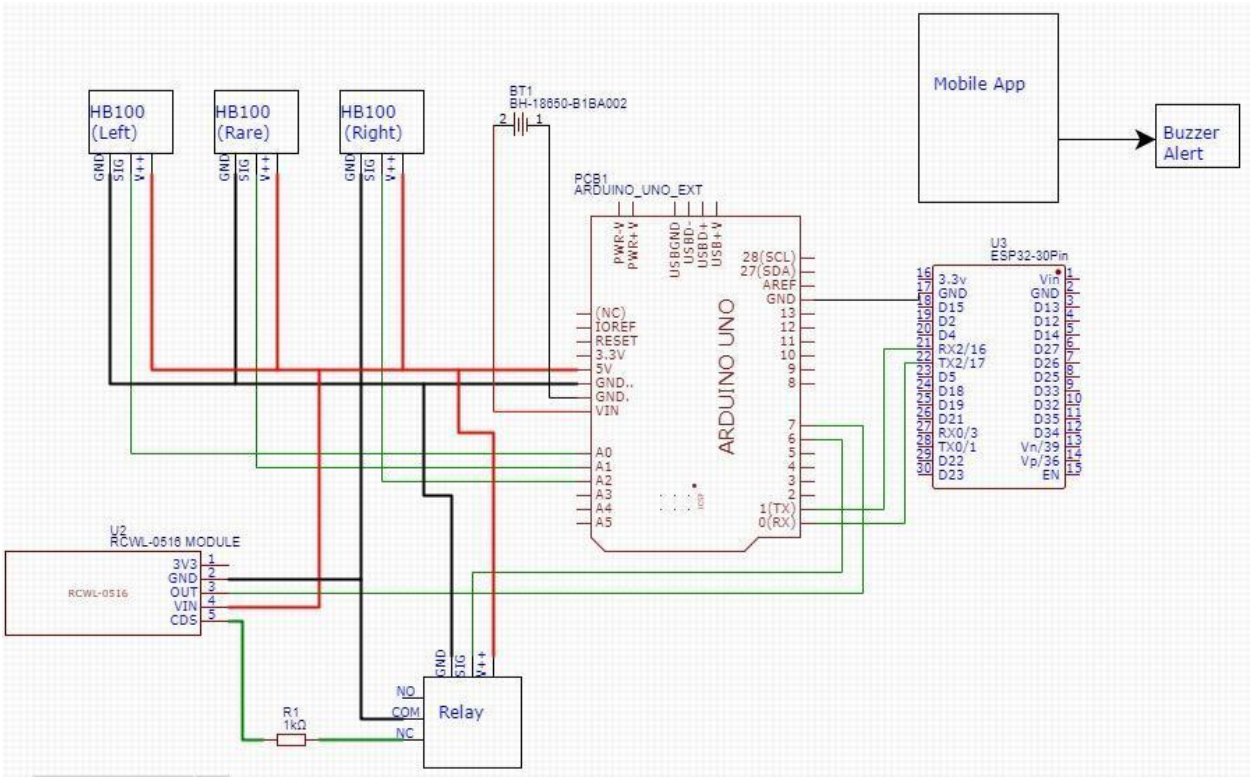

Literature review
1.04.2022
Ordering sensors
21.04.2022
Assembly and implementations (Concept)
05.05.2022
Coding Arduino
10.05.2022
PCB Electrical design
15.05.2022
Final PCB design
18.05.2022
Result 1
15.06.2022
Implementation and result
17.06.2022
Ordering PCB
20.06.2022
Brainstorming on final report
20.06.2022
Final Presentation
24.06.2022
Report submission
08.07.2022

Project Concept

In this project, two different kinds of Doppler Radar Sensors are used: the RCWL-0561, which has a 360-detection angle and range of 5m-7m used for motion detection, and HB100 with 160 degrees detection angle and range of 15m.

Three HB100 sensors are installed to cover the blind spots and to measure the speed and direction of the approaching vehicles. The detection range of the device is kept within 5-7ms, which means that data from the hb100 sensors is shown only when the RCWL-0561 detects the movement of any oncoming vehicle. An Arduino Uno is used to process the sensor's data, which is then displayed on the mobile's web browser via ESP32 Wi-Fi Module.

These components were chosen in order to maintain the device's affordability and enable mass production. Nowadays, everyone owns a smartphone, so we decided to create a webpage that will serve as a device alert system and worn the rider if there is an oncoming threat



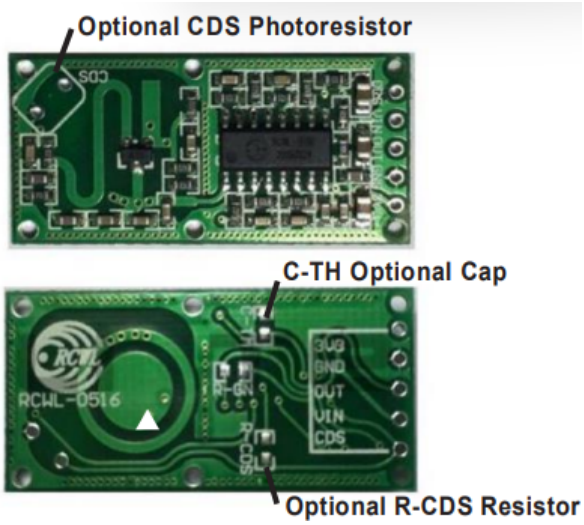
Hardware Circuitry

Sensors Description

RCWL-0561: The sensor consists of a supply pin, a signal pin, and a CDS pin

A CDS pin is used to vary the detection range of the sensor, which is done using a Relay.

The default detection range is 7ms, when the CDS pin is short-circuited using a 1M resistor, the range is set to 5ms.



Specifications:

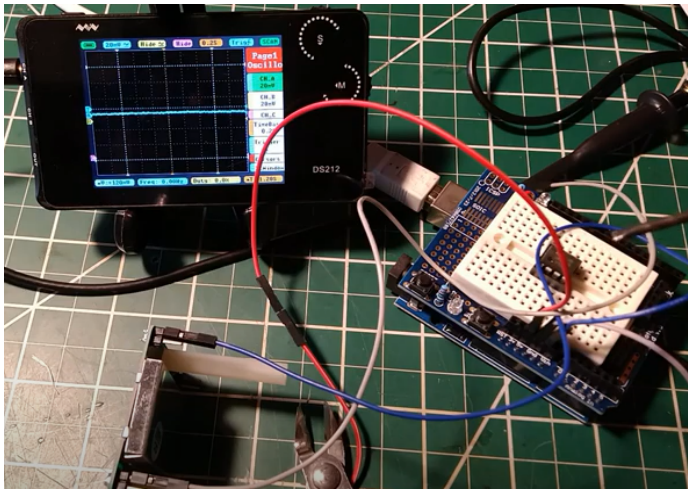
- **Power:** 4-28VDC at less than 3mA.
- **Detection Range:** approximately 5-9m.
- **Frequency:** 3.2GHz.
- **Transmitting Power:** typically, 20mW, whereas the maximum is 30mW.
- **Output Level:** -3.4V High <0.7 Low Output Drive: ~100mA.
- **Output Timing:** -2sec re-trigger with motion.
- **Operating Temperature**: -20 till 80 Celsius approx.
- **Storage Temperature:** -40-100 Celsius.
- **Terminals:** 0.1 Pitch solder holes.

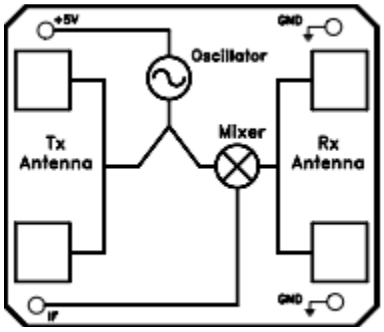
HB100:

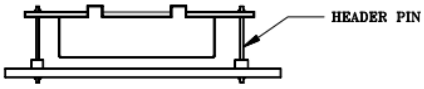
The sensor measures the speed of incoming vehicles. Among the many speed detection methods researched, radar detection did appear to be the most accurate way to detect the speed of approaching vehicles. It is utilized generally in commercial speed guns used by police to check for fast vehicles. The sensor has an oscillator and a mixer. The oscillator generates a sinusoidal signal with a frequency of 10.525 GHz and radiates it, in the form of a radio wave in one direction using a micro-strip patch antenna, which is thereby reflected back by the running vehicle, and the incoming echo frequency is detected by the built-in receiver. The subtraction of

the outgoing and incoming frequencies, done via Mixer, is then reflected in the IF (intermediate frequency) pin. The frequency of the output signal is proportional to the speed of the vehicle. The output is a low amplitude frequency signal in the audio range.

The sensor is a 3-pin circuit. Vcc, Gnd, and output pin.







The sensor output changes depending on the object's motion.

If the target object is approaching the sensor, the RMS value of the sensor output is positive, whereas it's negative.

Using the Doppler shift equation to calculate the relation between the shift in frequency and velocity of the moving object.

$$f_d = 2 * \cos A * f * v/c$$

Where,

f_d = shift in frequency in Hz (i.e. frequency of output signal from radar)

v = velocity of moving object

c = velocity of an electromagnetic wave in vacuum

f = frequency of a transmitted wave from the sensor (10.525 GHz)

Assuming the object is directly ahead so that the value of angle **cosA=1,** the expression for the velocity of the object reduces to:

v = f_d /19.49074 km/hour

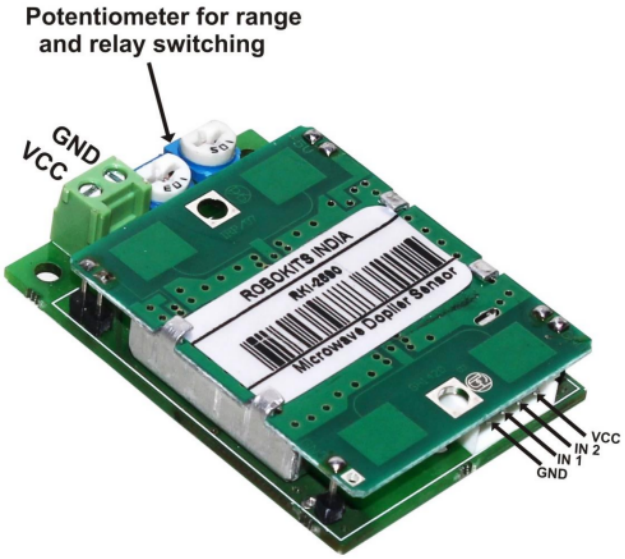
For **v = 0, f_d = 0**

For a speed limit of 60 Km/hr., i.e. v = 60 Km/hr., output frequency f_d = 1169.428Hz.

If the output signal of the sensor has a frequency exceeding 1169.444Hz, the vehicle exceeds the speed limit of 60 km/hr.

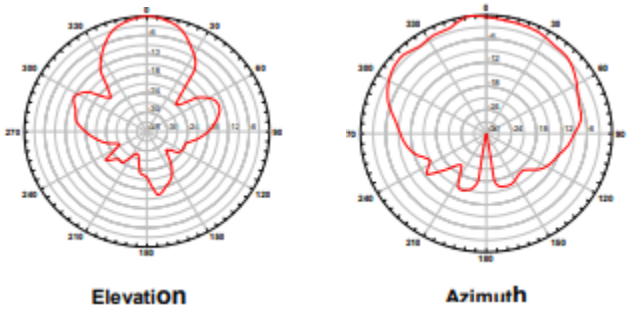
Features:

- Low output current, therefore an amplifier is used in order to further process the signal.
- CW or pulsed operation
- Flat profile
- Long detection range up to 15m
- Contains the potentiometer for range and relay switching duration.
- Output low signal can be used for trig a relay for applications
- Minimum logic low signal is 25 seconds



Applications:

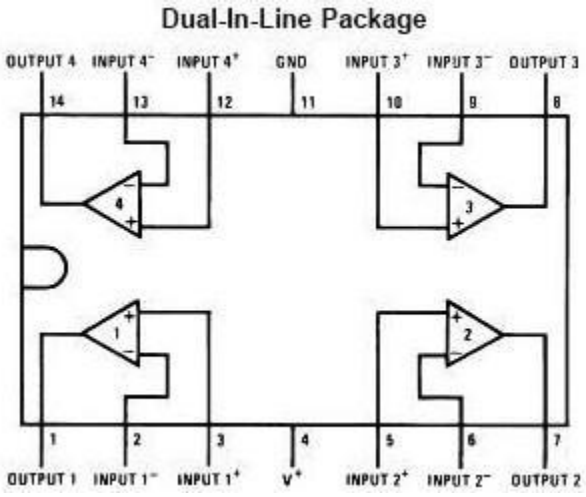
- Speed measurement
- Motion detection equipment
- Vehicle speed measurement and automatic doors
- Alarms



Amplifier:

Varieties of amplifiers are available in the market, and LM324 best fits our system's needs. The LM324-N series comprises four independent op-amps which were manufactured to work from one power supply over a large range voltage and a split of power supplies is also possible.
The resulting input voltage is possibly more than V+ without affecting the device. We can protect the input voltages from dropping negatively more than −0.3 VDC (at 25°C) by connecting an input clamp diode with a resistor to the IC input pin.

It has two class outputs, class-A output stage for small signal levels and it works as class-B for a large signal mode, which provides additional benefit to the amplifier as it allows it to work as a source and sink large output currents, So we can increase the power capability of the simple amplifies by providing NPN and PNP transistor as an external current booster.



LM324N IC

For the output current sinking application, the output voltage is required to increase approximately 1 diode drop above the ground to bias the on-chip vertical PNP transistor. The Resistor-Capacitor is coupled to the output of the amplifier for A.C. applications. A resistor has to be used from the output to the ground, to increase the class A bias current and prevent crossover distortion.

Features:

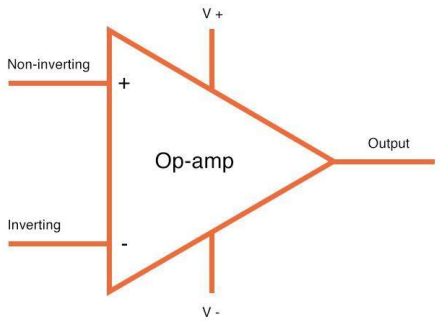
- Frequency compensated internally for unity gain
- Big DC Voltage Gain(100dB)
- 1MHz Bandwidth
- Lower Input Biasing Current(45nA
- Power Supply Range:
    - For Single Supply 3V to 32V
    - Dual Supplies ±1.5V to ±16V
- Lower Supply Current Drain(Essentially Independent of Supply Voltage)
- Lower Input Offset Voltage and Offset Current:5nA
- Resulting Input Voltage Range equivalent to the supply voltages
- Larger Output Voltage Swing 0V to V+ − 1.5V

Advantages:

- Single Supplies can be used.
- Four independent op-amps in a Single Package.
- Can be used in ALL Forms of Logic.
- Power Drain Suitable for Battery Operation.

Operational Amplifier:

It is a high-gain electronic amplifier with a differential input and single-end output. an op-amp gives an output voltage that is thousands of times bigger than the voltage difference between its input pins. The op-amp is one type of differential amplifier.



op-amp representation

Where differential input voltage consists of a difference of two voltages, one from non-inverting input(+) and inverting input(-), which is called the differential input Voltage.

The output voltage is given by the equation:

$$V_{\text{out}} = A_{\text{OL}}(V_+ - V_-)$$

AOL = the open-loop gain of the amplifier(without feedback)

Open-loop:

The amplifier's open-loop gain (AOL) is extremely high, on the scale of 100,000. V+ and V- drive the amplifier's output almost to the source voltage by a small difference. Saturation is setting the output to its maximum value, which is equivalent to the source voltage. Therefore, using an open-loop architecture for amplification is not viable. The amplifier can function as a comparator in the absence of a feedback network. When the input voltage Vin is given to the non-inverting input and the inverting input is kept at ground zero (0 V) either directly or through a resistor Rg, the output will be maximum positive if Vin, the input voltage applied to the non-inverting input, is positive; the output of the amplifier will be maximum negative if Vin is negative. The op-amp's AOL is the circuit's gain.

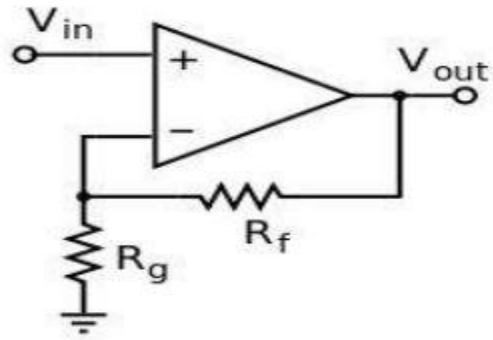
Closed-loop:

For practical purposes, feedback loops are used to regulate the amplifier's performance. By providing a portion of the output voltage to the inverting input, negative feedback is typically applied. This regulates the circuit's total gain and response, which is largely governed by the feedback network rather than the op-amp properties. The performance of the circuit is not significantly impacted by the value of the open-loop op-amp's response AOL if the feedback network is composed of components with values that are tiny in comparison to the input impedance of the op-amp. The response of the op-amp with its input-output and feedback circuit

is described using transfer functions. Op-amps are often utilized either in an inverting configuration or a non-inverting arrangement.

Non-Inverting Configuration:

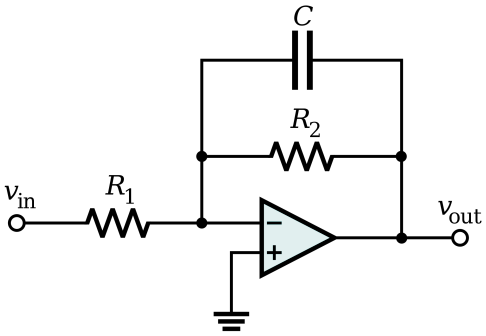
The input signal is applied to the non-inverting terminal (V+) in the non-inverting mode, while the feedback is provided to the inverting terminal (V-). The closed-loop gain ACL = Vout / Vin depends on whether negative feedback via the voltage divider Rf, Rg. The formula for this configuration's voltage gain is 1 + Rf/Rg. The phase between the input and the output is not reversed.



Non Inverting op-amp

Inverting Configuration and low pass filter:

Op-amp input is applied to the inverting terminal V- via a resistor R1 in the inverting configuration. The non-inverting terminal is wired to the ground, while the inverting terminal is supplied by feedback from the output through a resistor R2. The ratio of the feedback resistance to the input resistance, in this design, determines the overall gain of the inverting terminal.
Gain = R1/R2.
In order to design a low-pass filter in this setup, an additional capacitive reactance must be attached to the feedback path.



Inverting amplifier with low pass filter

An active low pass filter of the first order is represented by the circuit above. The threshold frequency value above which all frequency components are muted is known as the circuit's cutoff frequency. The cutoff frequency (in hertz) for the operational amplifier circuit seen in the diagram is defined as

$$f_c = \frac{1}{2\pi R_2 C}$$

or translated into radians per second as follows:

$$\omega_c = \frac{1}{R_2 C}$$

All frequencies above the fc value are muted, and the gain in the passband is -R2/R1.

The op-amp's gain can be raised to a high enough value to enable operation in the saturation area. In this situation, the output is a nearly digital pulse that may be connected to any TTL device's digital I/O pin.

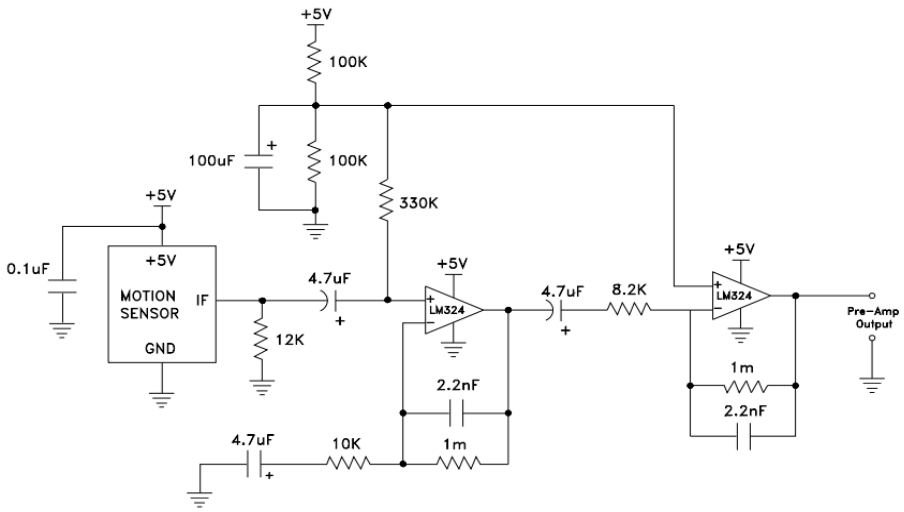
Using the digital pin of a microcontroller, the frequency of the output signal from the amplifier may be determined. A microcontroller can be used to calculate speed, measure frequency, and make the appropriate comparisons. On the display device connected to the microcontroller, the output is shown.

Amplification of Signals:

Prior to further processing, the sensor's sinusoidal wave output, which has a relatively modest amplitude, needs to be amplified. The sensor's output falls inside the audio frequency band. We are worried about the frequency below the speed limit, though. i.e., around 1.1 Khz. Therefore, a low pass active filter audio amplifier is preferred for the design.

The LM324 Quad op-amp amplifier IC is the one employed in this scenario. Four operational amplifiers make up this IC.

The radar's output signal is a few millivolts or less. Therefore, a high gain amplifier is required to change the voltage level to one that can drive TTL logic. To reduce the impact of noise,



The final design of operational circuit

amplification is accomplished in two steps. Below is a diagram that shows the entire amplifier circuit in use.

Two of the LM324 IC's four stages are used to amplify the radar output.

Non-Inverting stage:

The non-inverting amplifier configuration is used for the first stage. Since the sensor's output ranges from 0.01 to 0.2 Vdc, AC coupling of the output is required. A resistor (12K) is linked to the ground, and a capacitor (4.7 F) supplies the input to the non-inverting terminal. A resistive divider network is used to produce a dc offset of almost 2.5 volts. To create 2.5 volts at the center, two 100K resistors are connected between +5 volt and the ground, and a capacitor is connected in parallel to one resistor to reduce noise received with the power supply. The feedback path resistor value (1M) and the resistor value from the inverting terminal to the ground together determine the gain of the first stage (10k). The non-inverting amplifier has a gain of nearly 100.

Gain= (1+1M/10K) = 100.

Inverting stage:

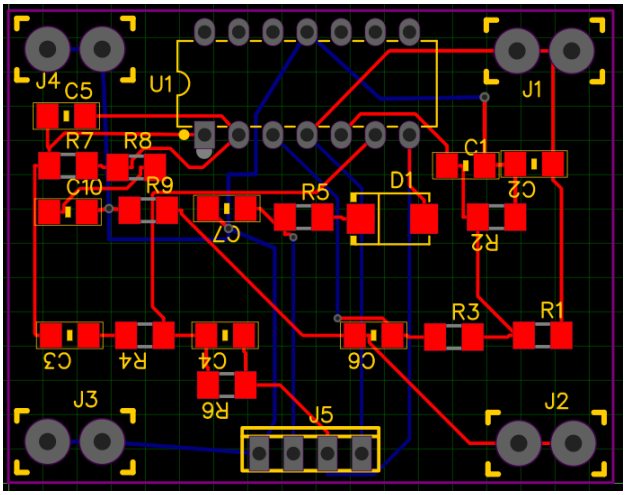
An offset sinusoidal signal with a modest dc value is the first stage's output. The capacitor at the output terminal filters the dc component at the output. An op-amp’s inverting amplifier makes up the second stage. Through an 8.2K resistor, the signal is delivered to the inverting terminal, and a 2.5-volt dc offset is given to the non-inverting terminal. The feedback resistor (1M) and the resistor (8.2 K) at the inverting input terminal values determine the gain for this stage. The equation provides the gain at this point.

Gain = -1.1M/8.2K = 122.1

This setup functions as an active low-pass filter using the 2.2nF capacitor in the amplifier's feedback loop. The system's accuracy is increased and the high-frequency noises are filtered.
To maximize the amplifier's overall gain, the two stages are cascaded. The result of the gain in each stage adds up to the overall benefit.

The total gain of the amplifier is 100 *122, or 12,200.

The second stage op-amp in the amplifier hits saturation due to the high gain of the device. The output is driven with extremely high gain to the saturation point because we are only interested in the signal's frequency and not its waveform's shape. The output of the second stage, which performs the function of a comparator, is a digital waveform that can be sensed by a microcontroller's digital pin.



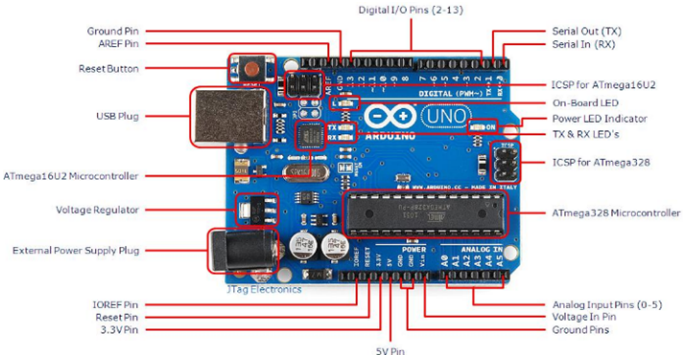
PCB Layout

Micro-Controller:

Arduino is a single-board microcontroller that aims to increase accessibility for the application of interactive objects or settings. The hardware is either a 32-bit Atmel ARM or an open-source hardware board built around an 8-bit Atmel AVR microprocessor. The user may attach multiple extension boards to current models thanks to their USB interface, 6 analog input pins, and 14 digital I/O pins. It includes a straightforward integrated development environment (IDE) that works on standard personal computers and enables users to create Arduino applications in C or C++.

An Atmega 328 microcontroller is built inside the development board known as Arduino Uno. Different sets of digital and analog I/O pins on the board are mapped to the microcontroller's input and output pins. The five analog pins can detect analog voltage as low as 4 mv. Digital

pins 2 through 13 can be utilized for input or output. Serial data transmission only occurs on pins 0 and 1. Ground and a supply voltage of +5 volts are required for Arduino to function. A USB cable linked to a USB plug or an external adapter can both be used to provide power. A 16Mhz crystal that is mounted to the circuit board powers the processor.



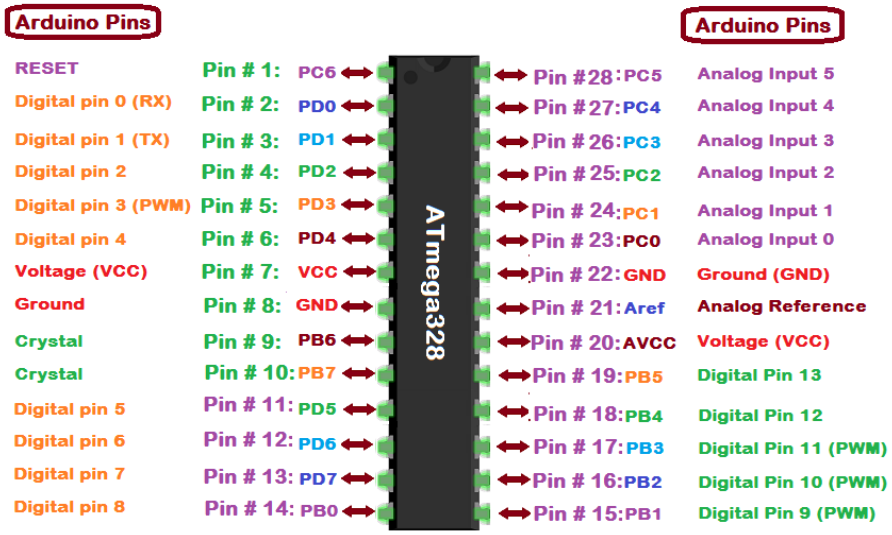
Features:

- Simple sensor and display device interface
- Using the Arduino IDE makes debugging and testing simple.
- Using a USB cable, the target circuit is simple to program.
- A wide variety of libraries are available in the repository for interacting with frequently used peripherals. Open-source project.

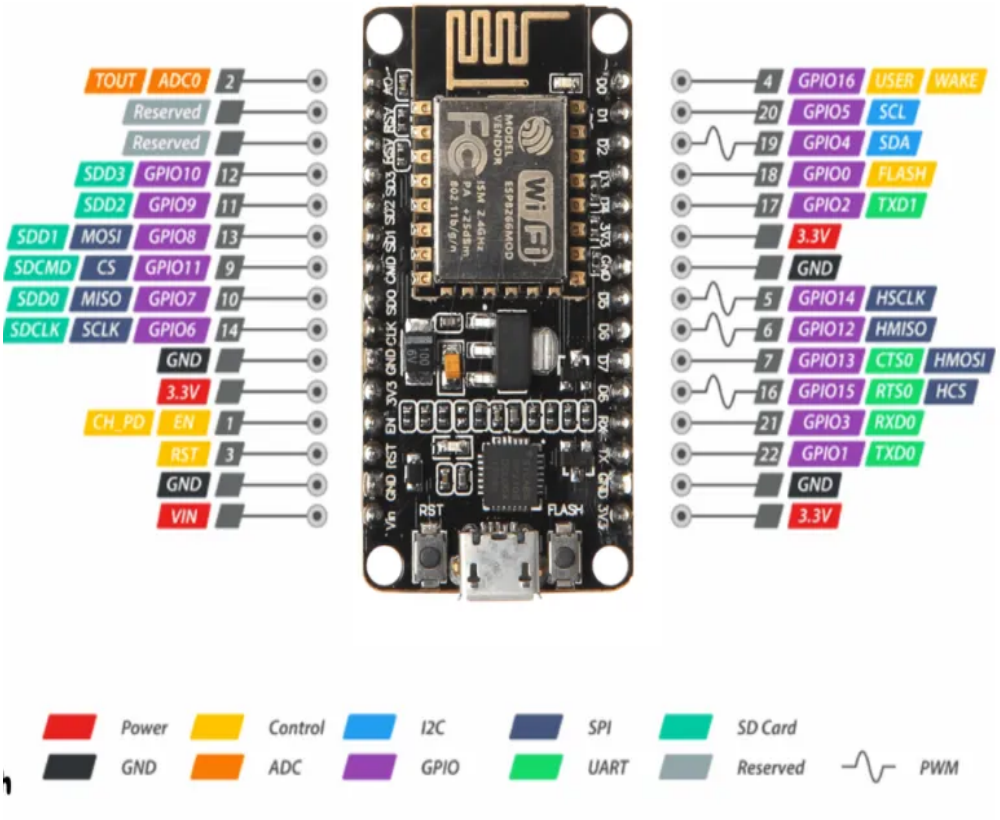
Atmega 328:

The microcontroller unit attached to the board, the Atmega 328, handles all processing. The high-performance Atmel 8-Bit AVR RISC-based microcontroller features 32 KB of read-while-write ISP flash memory, 1 KB of EEPROM, 2 KB of SRAM, 23 general-purpose I/O lines, 32 general-purpose working registers, three flexible timer/counters with compare modes, internal and external interrupts, serial programmable USART, a byte-oriented 2-wire serial interface, SPI serial port, and a 6-Channel 10- Internal oscillator watchdog timer with five software-selectable power-saving modes. The gadget runs on 1.8 to 5.5 volts.

ATmega328 Pinout



ESP32 Wi-Fi Module:



Pin Layout

Features:

- Open-source
- Interactive
- Programmable
- Low cost
- Simple
- Smart
- WI-FI enabled

Apart from these,

Arduino-like hardware IO

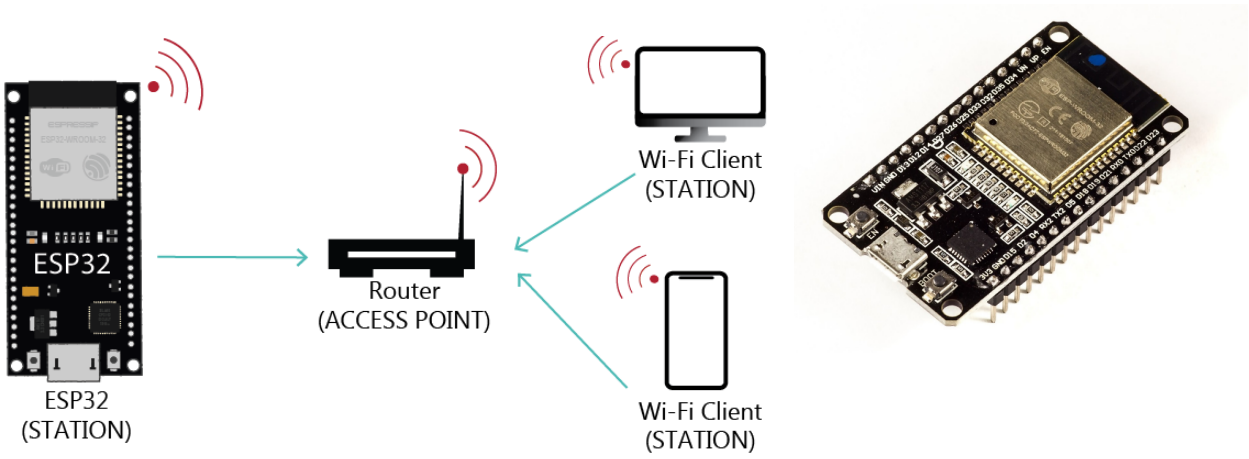
The unnecessary effort involved in setting and operating hardware may be greatly reduced by advanced hardware IO APIs. Interactive Lua script code that is similar to Arduino code.

Nodejs style network API

An event-driven API for network applications makes it easier for programmers to write Node. js-style code that runs on an MCU with a 5mm*5mm footprint. accelerate the creation of IoT applications significantly.

Lowest cost WI-FI

affordable integrated WI-FI MCU ESP8266 development kit that makes prototyping easier. We provide the best platform for creating IoT apps at the lowest cost.

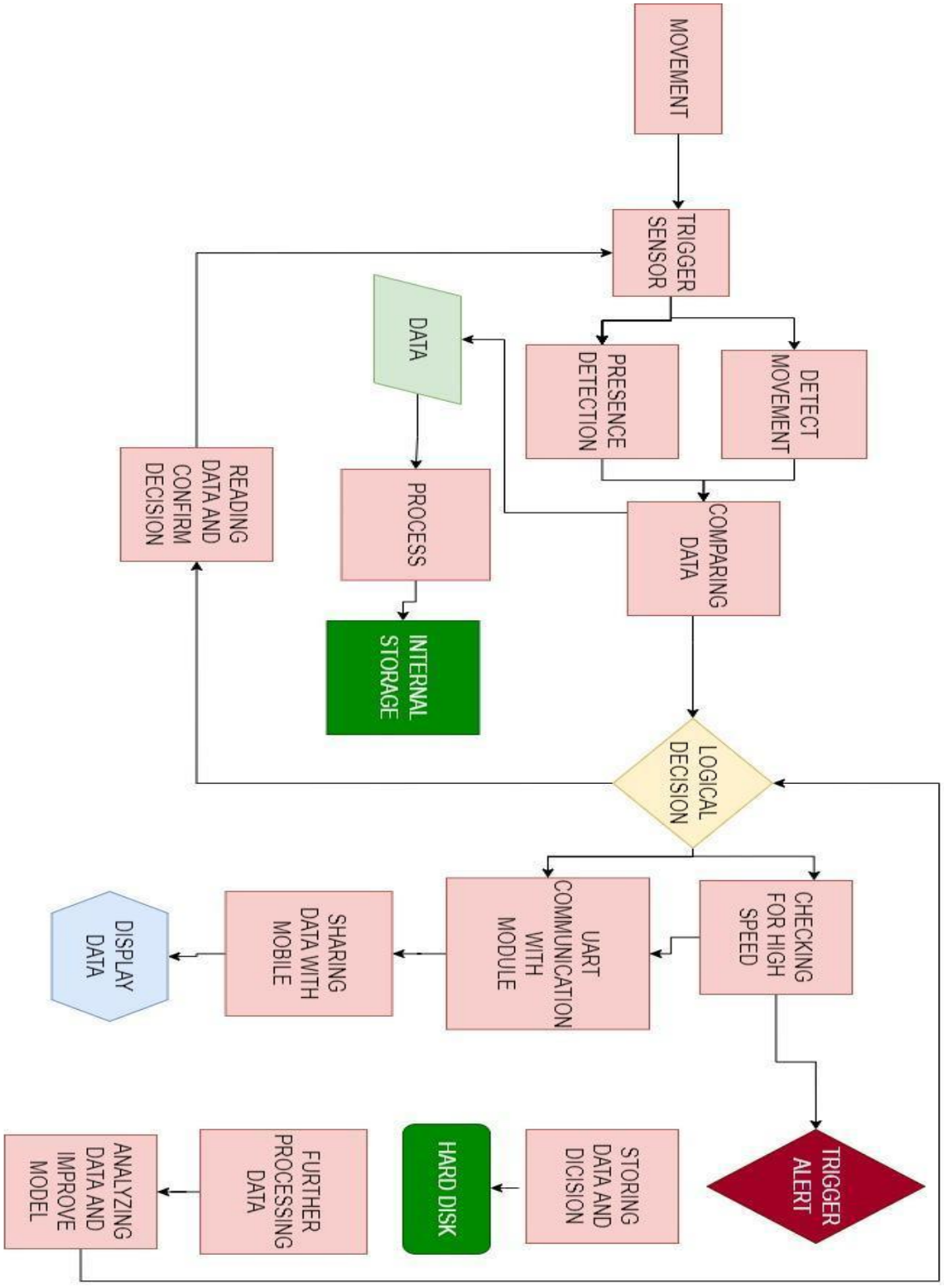


The ESP32 board can act as a Wi-Fi Station, Access Point, or both. The SSID and password variables hold the SSID and password of the network we want to connect to, in this case, the rider's mobile phone.

Wi-Fi.begin() function helps in establishing a connection to the network and it is done by passing SSID and password as arguments. The while loop keeps on checking if the connection was already established by using Wi-Fi.status(). When the connection is successfully established, it returns WL_CONNECTED. When the ESP32 is set as a Wi-Fi station, it can connect to other networks (like your router). In this scenario, the router assigns a unique IP address to your ESP32 board. To get your board's IP address, you need to call WiFi.localIP() after establishing a connection with your network.

The web server library creates a data channel with the help of the function handle index(), creating a handshake between the station and the client.

The handshaking is done by sending a JSON formatted request doc. file and getting the response in the form of a string. After the handshaking, a boolean messageReady is set to true and sensor data is transmitted to the app. Output = "hb1" + "hb2" + "bh3" + "flag"



Algorithm structure

Mobile Application: Front End (User Interface)

Overview:

The user interface of the application is built by Using MIT App Inventor API. It provides pre-built layouts for the various types of phones and tablets which makes the front-end developer's job easy as it produces quick mock-ups of the design, which in turn results in faster optimization of the dimensions and overall look.

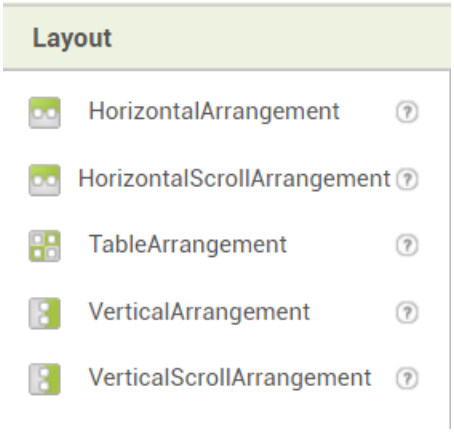
MIT App Inventor User Interface API:

MIT App Inventor is an intuitive, visual programming environment that allows everyone to build fully functional apps for Android phones, iPhones, and Android/iOS tablets. Following are a few components that are available for developing the front end of the application-

1. **Layouts:**

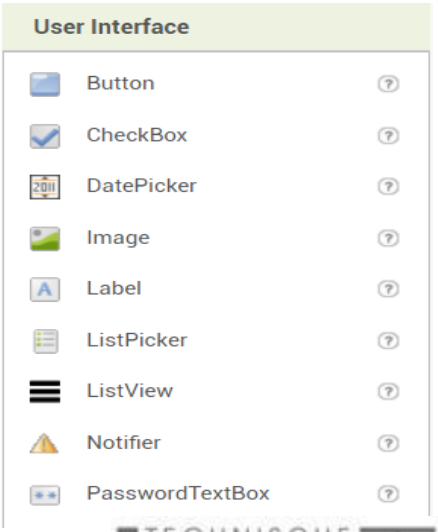
   API provides various kinds of layout boxes.

   a. Horizontal Arrangement layout puts the elements and widgets horizontally in the box.
   b. The Horizontal Scroll Arrangement box lets the developer and user scroll down inside the box.
   c. Table Arrangement builds tabular layout for the widgets
   d. The vertical arrangement layout arranges widgets in the vertical orientation one after the other
   e. Vertical Scroll Arrangement allows developers and users to slide and scroll in the entire layout, reducing the restrictions inside the box.



2. **Widgets:**

   Widgets allow the user to interact with the application. They also intake and display the information to the user. MIT App Inventor provides various kinds of widgets, out of them major ones, which we have used in the User interface of our application-



a. Button: a two-state switch, it has two states ‘normal’ and ‘on-press’. Both states can be assigned different logics or functions.

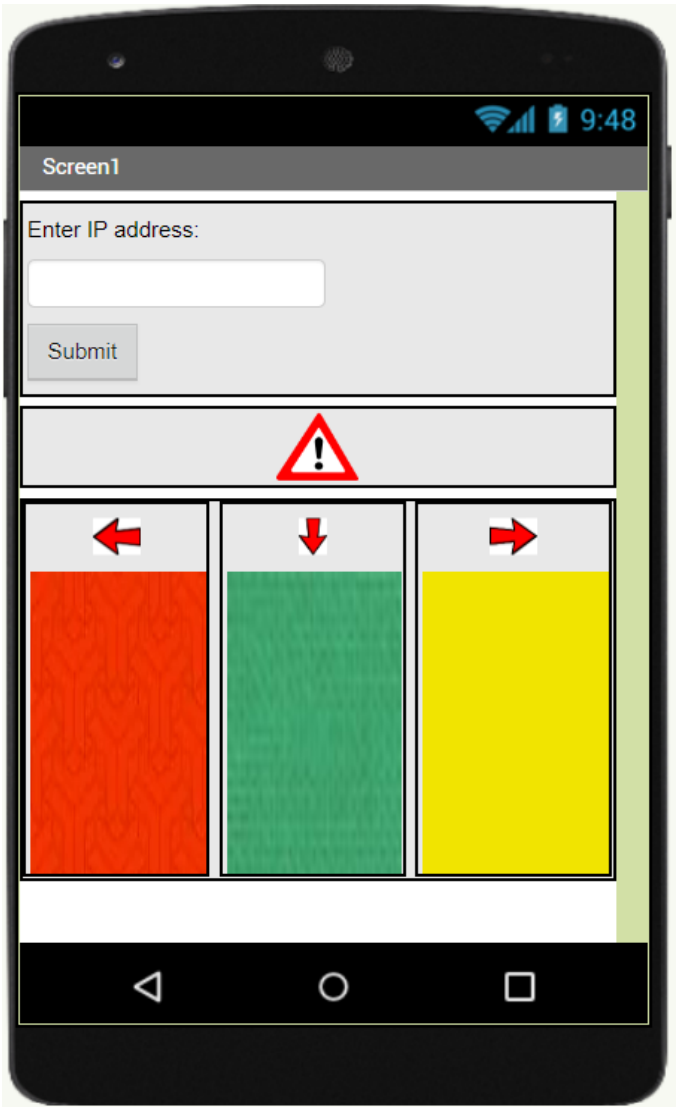
`

3. **Label**: space on the screen that is used as an output to display the data from the database or a result of a function.

   a. Image: This widget can display and refresh the state of an image. It is made up of pixel values that can adapt to the pixel values it is displaying and can be refreshed using a function.

Application’s Front End:

Following are the features and components of the User Interface application:



IP address section:

A vertical arrangement box layout is used to contain a label that directs the user to enter the IP address of the ESP32 into the Text input box. After entering the IP address in the text input box, the user ought to press the ‘Submit’ button to establish the connection between ESP32 (acting as a server) and application (acting as a client) via port 80, that is HTTP protocol.

1. Left, Rear and Right-side cards:

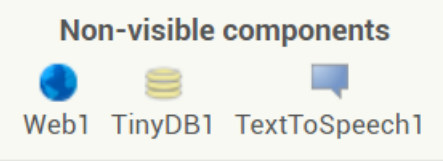
a. Red: If the approaching vehicle’s speed is above the threshold, the card will turn Red. A voice assistant gets activated warning riders of the incoming

high-speed vehicle. In the example, the left side is Red, signifying a high-speed vehicle approaching from the right.

b. Yellow: If the approaching vehicle is detected and its speed is less than the threshold, then the card will turn Yellow. A voice assistant gets an activated warning rider of the detected vehicle. In the example, the right card is yellow, which means the vehicle is detected on the right side but it has a relative speed below the threshold.

c. Green: If there is no vehicle detected by sensors, then the card will turn in Green. No voice assistance is provided in this mode. In the example above, there is no vehicle detected at the rear side of the rider.

Hidden components:



There are hidden components in the application's User Interface, which are essential for the full functioning of the app:

1. Web1: This feature enables applications to use Wi-Fi cards for mobile phones.
2. TinyDB1: This feature stores data captured by the sensors into a local database.
3. TextToSpeech1: This is the voice assistant feature that converts pre-written texts into speech, which is helpful in giving audio aid to the rider.

Early plans and Rejections:

Our early plans for the User Interface were:

1. To display the sensor data directly into a label inside the User Interface screen. Data from all four sensors were to be displayed as a live feed.
2. For audio aid, we were using an alarm or notification sound.

Above mentioned plans were discarded for the following reasons:

1. The constant feed of data would be overwhelming for the rider and will also distract him while riding, creating more potential for accidents.
2. Not every rider would be a Tech Savvy, hence those sensor data would be useless for him/her.

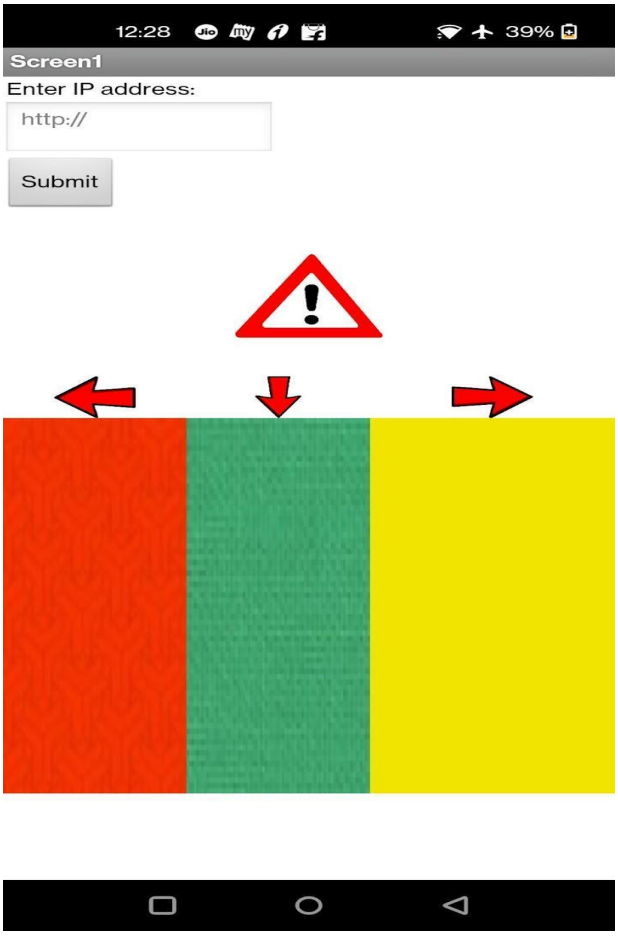
3. If the rider has some eyesight issues, it might be very hard for him/her to read the data in a small font without glasses.
4. Alarm or Notification sound can only alert the presence of a vehicle, but it does not carry detailed information about the direction of the approaching vehicle, which can delay the reaction time of the rider.

Below are the implemented solutions:

1. Constant feeds of data are replaced by bright colored cards which are big enough to grab the rider's attention and simultaneously give enough information.
2. Alarm sound is replaced by a Voice assistant which not only warns the rider about the incoming vehicle but also the intensity and direction of the object. This reduces the lag in reaction time as the rider already has the information in less time.

Deployment as APK on the phone:

Following is the screenshot of the deployment of an application using Android's '.apk' package:



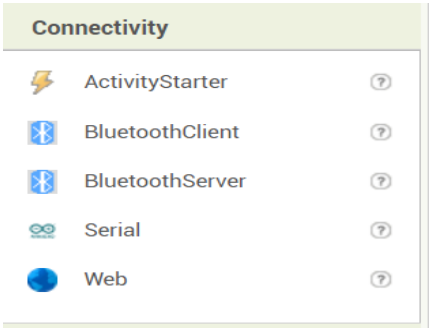
Mobile Application: Back End

Overview:

The backend of the application is built using MIT App Inventor API. This was used because of its good compatibility with Android, even with the latest versions such as Android 11 and 12. It is also compatible with the latest release of the Apple smartphone's Operating Systems.

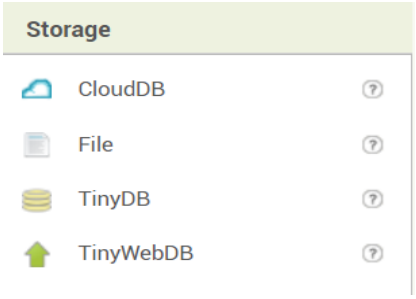
MIT App Inventor's Backend API:

MIT App Inventor is a visual programming tool that allows developers to build their program using pre-build code blocks of conditional statements, loops, Boolean logic, etc.



The back end is also provided with pre-built features for connectivity such as Wi-Fi and Bluetooth features. Wi-Fi comes with various security protocols like WEP, WPA, and WPA2.

API also provides features for storage, both on Cloud and Local systems. Developers can choose either database format, i.e., file with '.db' extension, or can simply store data in a Plain Text File with .txt extension.



It also comes embedded with various types of sensors such as Motion Sensors, Location sensors, barometers, etc. It handles and provides System Permission to the App seamlessly irrespective of the type of Operating System and its version.

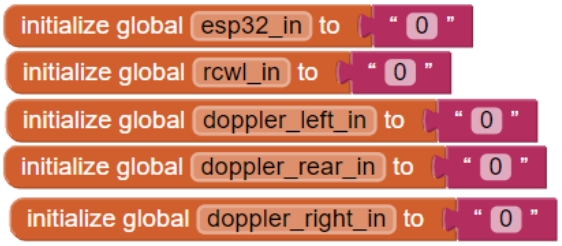
Hence, MIT App Inventor is the best choice for back-end development as it makes the prototyping very fast which results in faster implementation and debugging. It removed the stress of long compiling times for Mobile phones (which can range from 40 minutes to 90 minutes) and the incompatibility issues with different versions of Operating System..

Understanding Code Blocks

1. Declaring global variables:

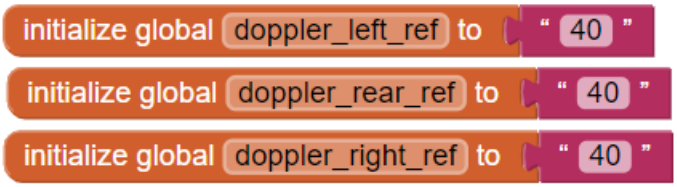
a. Incoming sensor values:

Global variables for incoming data via Wi-Fi connectivity are declared. It contains the 'ESP32_in' variable which is responsible for the intake of the entire data coming from the ESP32 module. Along with it, different sensor variables are declared and set to zero, which will be later changed by the help of list indexing.



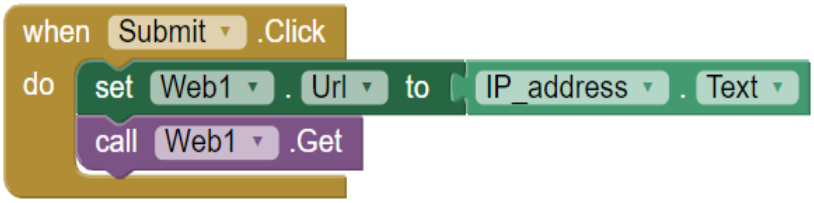
b. Reference values:

It signifies the top limit of the sensor values. Their unit is in Kilometers per hour (km/hr). Here, the top limit is set to 40, which means that the relative speed of the approaching vehicle has a limit of 40 km/hr, and if it crosses this boundary, the following code blocks will be activated.



2. Wi-Fi connection:

After the user inputs the IP address of the ESP32 module in the User Interface of the application and presses the 'Submit' button, a connection is initiated between the ESP32 (which is acting as a server) and Mobile phone (behaving as a client). The protocol used here for connection is HTTP, over port number 80.

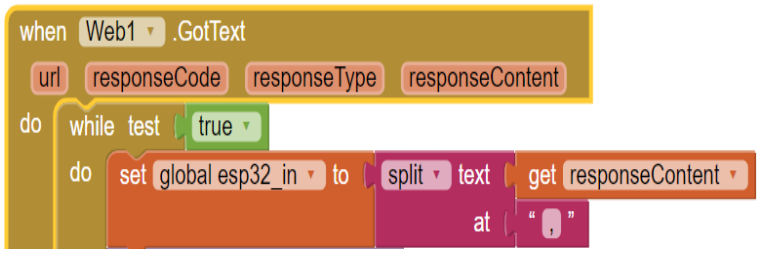


Our first choice was HTTPS (port number 443) due to its security robustness, but since we need to reduce our rider's response time in case of unfortunate emergencies, HTTP was agreed upon due to its fast connection. Using HTTPS was discarded as the connection's encryption and decryption could create a lag and also increase computational power for ESP32 modules.

An application sends a GET request to the server, which is one of the methods of HTTP. GET request as the application only intend to receive data in the "Read-only" mode from the server.

3. Response from the server:

After the GET request, the server responds to data for various sensors in the format of a String data type, where each sensor data is separated by commas, for example, left side "HB100, rear side HB100, right side HB100, RCWL data".



A While loop has been used here which is set Boolean True so that until the connection breaks or is interrupted (which will turn into Boolean False), an incoming response must be received and processed further.

After receiving data in String format, the Split function is used to strip the input string at commas (",") and store them in a list.
This generated list contains sensor data in the order:

[left side HB100, rear side HB100, right side HB100, RCWL data]

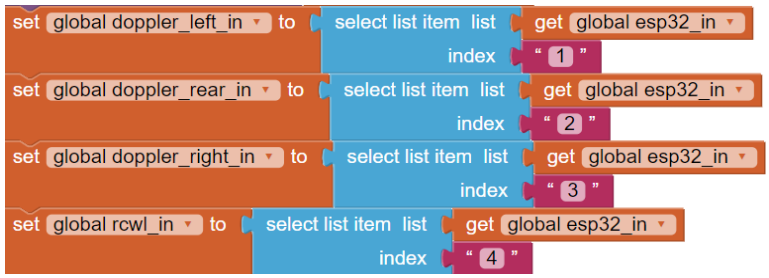
The purpose of doing this is to increase readability and extraction of the sensor data for further processing.

4. Data Storage:

The sensor data after conversion into list format, is stored in a database in the local device. MIT App Inventor gives seamless permissions for storage in the Internal Memory of the device. Tinydb is a database used for storing values.

This storing of sensor data is done for the future scope of this project, where we can implement a learning module (like Machine Learning) into our app instead of Hardwiring which is done currently. This will enhance the safety of riders by predicting the profiles of incoming vehicles

5. Indexing Sensors:

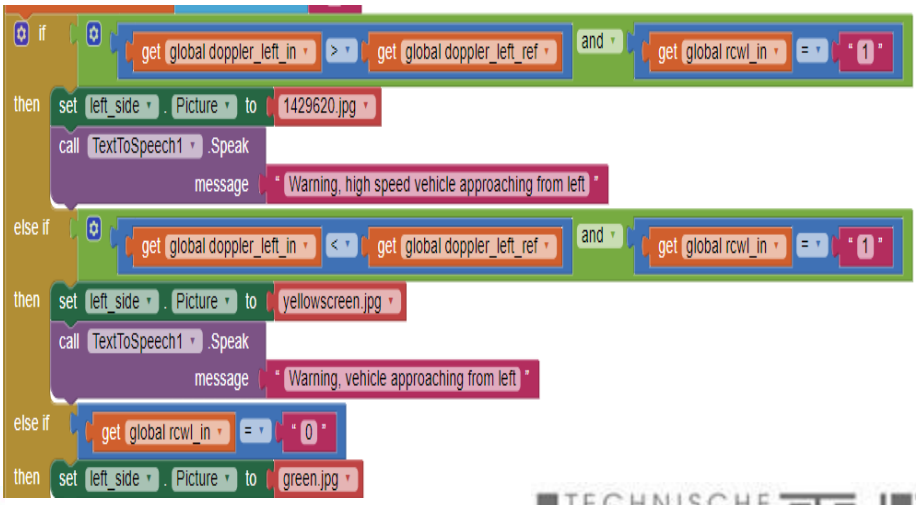


After processing data into list format, it needs to be extracted for each sensor for further calculation. In most programming languages, indexing of the list starts from 0 but in the case of MIT App Inventor, the index starts from 1. Hence, extraction of data is done using index wise:

Left side HB100 has index 1, rear side HB100 has index 2, right side HB100 has index 3 and RCWL sensor has index 4.

6. Left side conditions:

After taking data separately for the left HB100 sensor and RCWL, it is put under three conditions, if, else if and else.



The first condition checks if the RCWL sensor has detected an

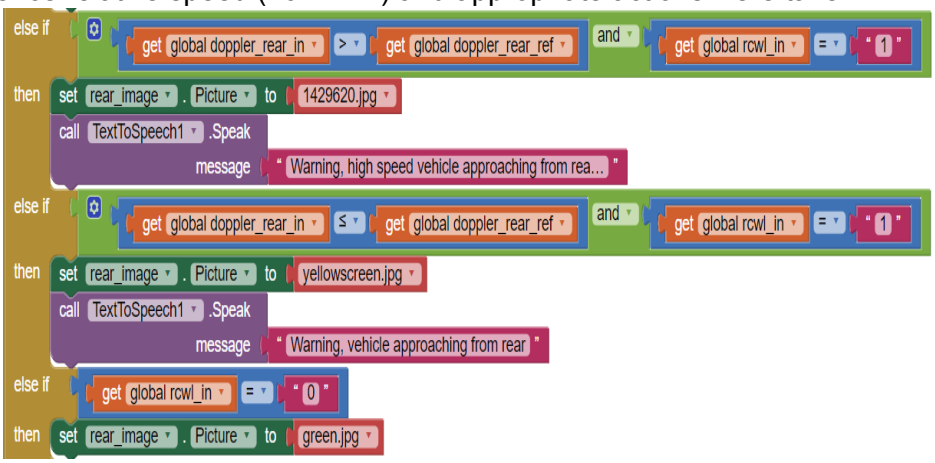
object (value coming from RCWL is 1) and if the incoming sensor data from the left HB100 is greater than a reference value (40 km/hr), then two operations will occur – the left Card on the User Interface of the application will turn bright Red and a Voice Assistant will be activated which will warn the user about the incoming vehicle at high speed from left.

In the second condition, if an object is detected by the RCWL and the relative speed of the incoming object recorded by the left HB100, is less than the reference relative speed, the card on the left side of the User Interface will turn Yellow and Voice assistant will warn about the incoming object from the left but this time it will not mention that the vehicle has high speed.

In the third condition, if there is no object detected by the RCWL sensor (the value sent by the sensor is zero), then the card on the left side of the User Interface will turn Green. This time there will not be a voice from the assistant as there is no threat nearby.

7. Rear side conditions:

Using three condition statements, data incoming from the Rear side HB100 is compared with the reference relative speed (40 km/hr) and appropriate actions were taken.



At the first, condition, if the object is detected by the RCWL sensor (incoming value is 1) and the relative speed of the incoming vehicle from the Rear side is greater than a Reference value, then the middle card will turn Red and the voice assistant will alarm about a high-speed vehicle approaching from the rear side.

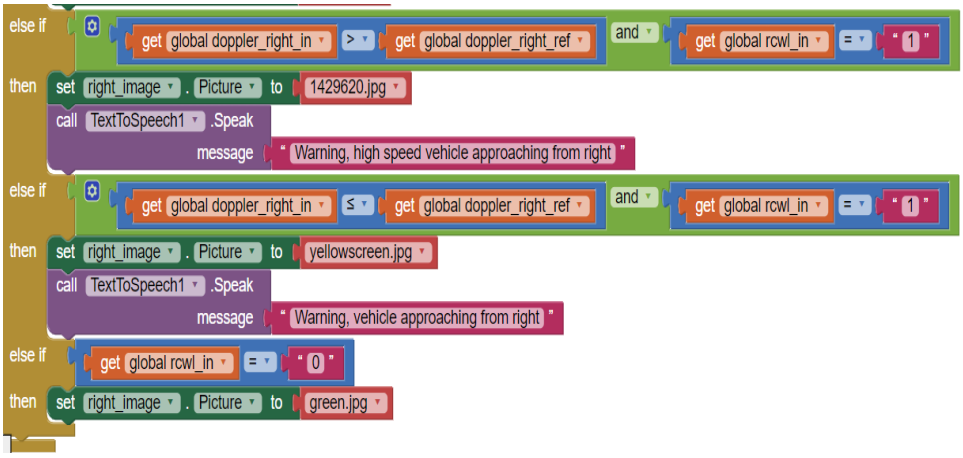
In the second condition, if the Object is detected by RCWL and value from the rear sensor is less than the reference value, then the card in the middle at User Interface will

turn Yellow color and the voice assistant will alarm that there is a vehicle approaching from the rear side. This will alarm the rider about the incoming vehicle so that he can be alert about the potential danger.

The last condition checks that if there is no Object detected by the RCWL sensor (incoming value from the sensor is zero), then the card in the middle at the User Interface turns Green, representing that there is no potential threat to the rider from the rear side. The Voice assistant stays silent to avoid disturbing the comfort of the rider unnecessarily.

8. Right side conditions:

Similarly, three conditions are used for right side object detection. In the first condition, if the RCWL sensor detects an Object (value given by the sensor is 1) and the relative speed of the incoming object from the right side is more than the reference relative speed, then the card on the right side of the User Interface turns Red and the Voice Assistant alerts rider of the high-speed incoming vehicle from his/her right side.



On the other hand, if the object is detected by RCWL on the right side but the speed is less than the reference value. Then, the card on the right side of the UI changes to Yellow, and the voice assistant just warns of the presence of an approaching vehicle from the right side.

At last, if there is no object detected by RCWL on the right side of the rider, the Voice assistant remains silent as the rider should focus on the road if there is no emergency and the card becomes Green, telling the user that he/she is out of the danger zone.

Scope for future improvements

In Mobile Application:

1. Embedding Machine Learning:

Currently, our application is very hard-wired and has been coded using classic if-else statements. There are no learning modules present in the software. In further iterations, we intend to add a machine learning module (possibly using the sci-kit library).

To reach this goal we must rebuild our application using the Python programming language and compile it for both android and ios devices using the buildozer compiler.

We are already collecting sensor data in a database file in the current version of our app. We intend to use this collected data for training of the machine learning model and hence, reinforce further the rider's safety.

2. Rider's riding pattern:

Currently we are only monitoring approaching vehicles using our sensors. Further, we intend to add more sensors on bikes and synchronize them together to capture the riding pattern of the rider, especially, how he/she behaves when alerted about the vehicle approaching from a certain side at a certain speed.

This will help us get the data about the rider's behavior at the time of emergencies and we can train our machine learning model using it, which in turn will predict the action of the rider by also checking how safe that action would be.

For example, if the sensors capture a high-speed vehicle approaching from the left side of the rider, then our pre-trained machine learning model will predict that the rider is going to turn towards the right to save himself. And since the model knows that the rider is going to turn towards the right, it will check forward sensors to see if it is safe to turn towards the right or not, so that the rider must not hit another object in order to save himself from an approaching vehicle. The Machine learning model

would suggest the best and optimized course of action to the rider using Voice assistant.

3. Personalization:

For now, our app has no feature to store user-specific profiles for recording and prediction. In case, if different riders share the same bicycle, it has no clue how to differentiate between them and provide personal assistance as different people could have different riding skills.

We wish to add a login and signup feature, which will distinguish among the riders and will create and store in a different database file for different users. This will help the prediction module to predict the optimized course of action for a particular user, enhancing safety and comfort.

4. Privacy and Security:

Since we will be collecting user's personal data, their security and privacy is one of our top priorities. We intend to use the Fernet encryption module to encrypt the databases containing users' data so that only the application can read it and it must not be readable by any third party.

We first thought of using RSA encryption but since it uses the Asymmetric encryption method (uses Public and Private keys), it will increase computational time and complexity which might create a lag in the reflex action of the user, hence could possibly delay his safety measures.

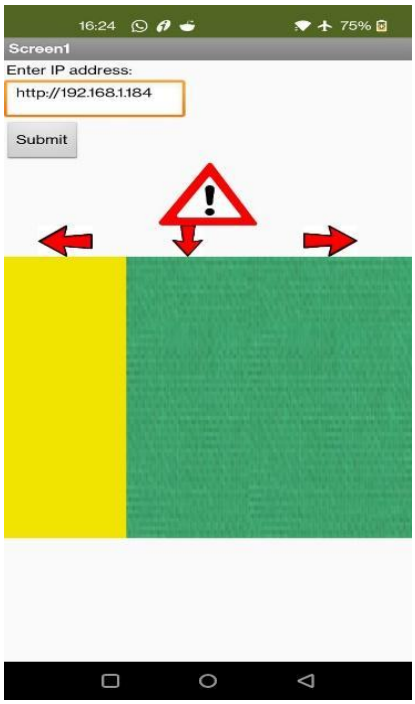
For that reason, we would be using Fernet as it is a Symmetric encryption method, it uses only one key to both encrypt and decrypt the database file and hence, decreases the reflex action time of the rider.

We believe this is the optimized compromise between users' road safety and its data privacy.

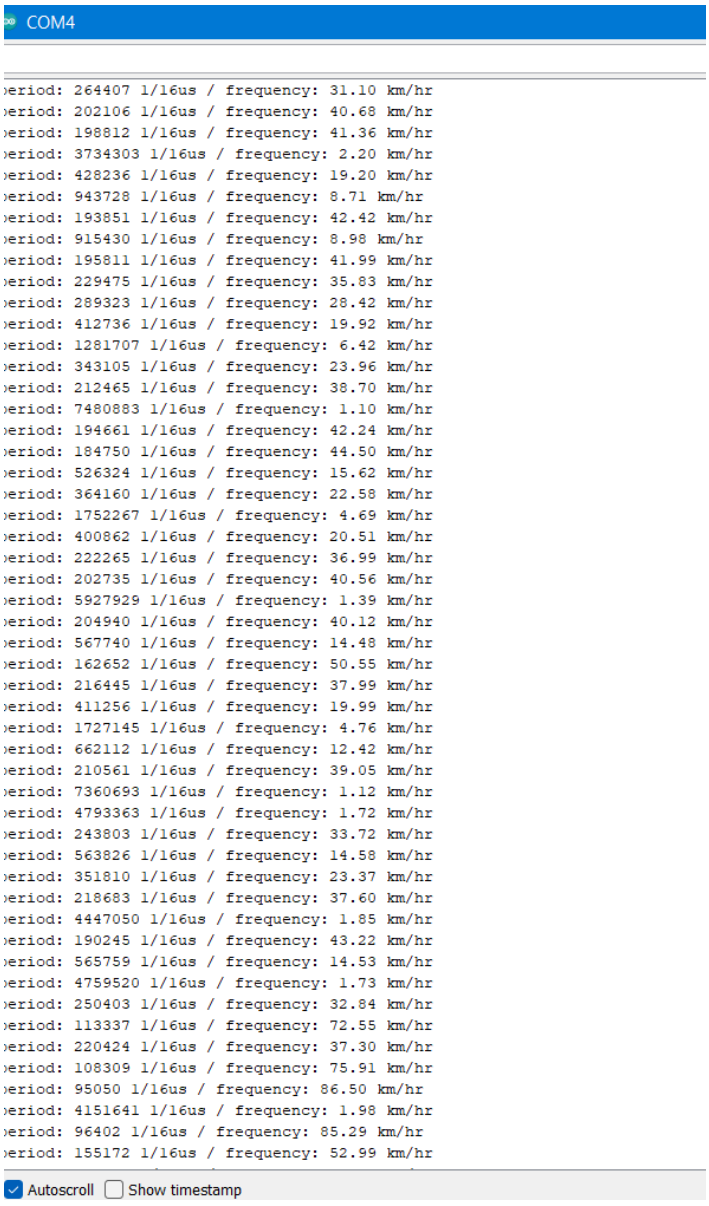
Results

Experiment steps:

1. First, we put the HB100 sensor on the carrier of the bicycle and placed it near the on-going traffic.
2. The data incoming from the HB100 was directly monitored in the Serial Monitor feature of the Arduino IDE for better calibration of sensors' orientations.
3. After calibrations, Sensors and Arduinos are connected to ESP32 module which is responsible for behaving as a web server over Wi-Fi connection.
4. Application in phone is connected to ESP32 webserver using its IP address.
5. Application takes in the object presence and relative speed detection, and the card on the app changes color along with voice assistance.



Observation:



Conclusion:

The oncoming vehicle had a speed of roughly 40km/hr, and the sensor shows a few intermediate values of around 10km/hr created by background noise, whereas it senses the speed of the vehicle rather precisely.

Appendix

Arduino Code:

```
//required library

#include "AnalogFrequency.h"

#include #include "ArduinoJson.h"

//communication variable

String message = "";

bool messageReady = false;

//frequency period library use for sensor 1

double lfrq;

long int pp;

double hb1;

//global variable define for sensor number 2

// Below: pin number for FOUT

#define PIN_NUMBER 4

 // Below: number of samples for averaging

#define AVERAGE 4 unsigned int doppler_div = 19.46;

unsigned int samples[AVERAGE];

unsigned int x;

double hb2;

uint32_t displayTimer = 0;

//analog library use for sensor 3
```


```
#define ADCPin A1

// Incoming data is summed, so fetching the results every second // will indicate speed over the previous second

// How often in mS to display the results ( 0 = print all results if possible)

#define printDelay 0

double hb3;

//*****************************************/

// RCWL-0516 Sensor (Sensor 4)

int Sensor = 12;

int LED = 3;

int flg;

void setup(){

        Serial.begin(115200);

        pinMode (Sensor, INPUT);

        Serial.println("Waiting for motion");

        pinMode(PIN_NUMBER, INPUT);

        setupADC(ADCPin);

        //frequency period

        FreqPeriod::begin();

        Serial.println("FreqPeriod Library Test");

        }

void loop() {

        pp = FreqPeriod::getPeriod();

        //Monitor serial communication
```


```
//frequency period for sensor 1
if (pp) {
        Serial.print ("period: ");
        Serial.print(pp);
        Serial.print(" 1/16us / frequency: ");
        lfrq = 16000400.0 /pp;
        hb1 = lfrq/19.46;
        Serial.print(hb1);
        Serial.println( " kmh ");
        }
//for sensor 2
noInterrupts();
pulseIn(PIN_NUMBER, HIGH);
unsigned int pulse_length = 0;
for (x = 0; x < AVERAGE; x++) {
        pulse_length = pulseIn(PIN_NUMBER, HIGH);
        pulse_length += pulseIn(PIN_NUMBER, LOW);
        samples[x] = pulse_length;
        }
interrupts();
// Check for consistency
bool samples_ok = true;
unsigned int nbPulsesTime = samples[0];
```


```
for (x = 1; x < AVERAGE; x++) {
        nbPulsesTime += samples[x];
        if ((samples[x] > samples[0] * 2) || (samples[x] < samples[0] / 2)) {
                samples_ok = false;
        }
}
if (samples_ok) {
        unsigned int Ttime = nbPulsesTime / AVERAGE;
        unsigned int Freq = 1000000 / Ttime;
        hb2 = Freq/doppler_div;
        Serial.print(hb2);
        Serial.print("km/h\r\n");
        }



//for sensor 3
if( fAvailable() && millis() - displayTimer > printDelay ){
        displayTimer = millis();
        uint32_t frequency = getFreq();
        hb3 = frequency/19.49;
        Serial.print(hb3);
        Serial.print("km/h\r\n");
        }
//RCWL-0516 Sensor4
```


```
int val = digitalRead(Sensor);
//Read Pin as input
if((val > 0) && (flg==0)) {
        Serial.println("Motion Detected");
        flg = 1;
        }
if(val == 0) {
Serial.println("NO Motion");
flg = 0;
}
//for sending data to esp32
while(Serial.available()){
        message = Serial.readString();
        messageReady = true;
        }
//Only process message if there's one
if(messageReady){
        //The only messages we'll parse will be formatted in JSON
        DynamicJsonDocument doc(1024);  // ArduinoJson version 6+
        //Attempt to deserialize the message
        DeserializationError error = deserializeJson(doc,message);
        if(error){
                Serial.print(F("deserializeJson() failed. :"));
```

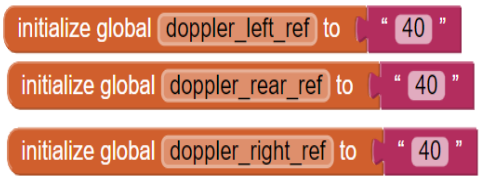

```
                Serial.println(error.c_str()); messageReady = false; return;
                }
            if(doc["type"] == "request"){
                doc["type"] = "response";
                //Get data from analog sensors
                doc["hb1"] = hb1;
                doc["hb2"] = hb2;
                doc["hb3"] = hb3;
                doc["RCWL"] = flg;
                serializeJson(doc,Serial);
                }
            messageReady = false;
            }
}
```

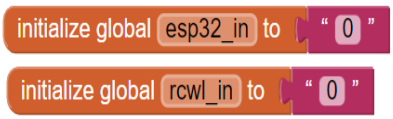

ESP32 Code:

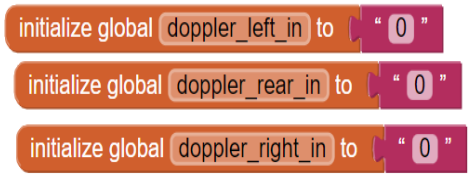
```
#include"WiFi.h"
#include"Webserver.h"
#include"ArduinoJson.h"
Webserver server;
#include "WiFi.h"
#include "WebServer.h"
#include "ArduinoJson.h"

WebServer server;
char* ssid = "ONEPLUS";
char* password = "12345678900";

void setup(){
        WiFi.begin(ssid,password);
        Serial.begin(115200);
        while(WiFi.status()!=WL_CONNECTED){
                                        Serial.print(".");
                                        delay(500);
                                        }
        Serial.println("");
        Serial.print("IP Address: ");
        Serial.println(WiFi.localIP());

        server.on("/",handleIndex);
        server.begin();
        }

void loop(){
        server.handleClient();
        }
```

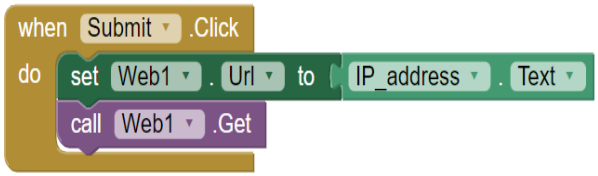

```
void handleIndex(){
        // Send a JSON-formatted request with key "type" and value "request"
        // then parse the JSON-formatted response with keys "gas" and "distance"
        DynamicJsonDocument doc(1024);
        double hb1 = 0, hb2  = 0, hb3 = 0, flg = 0;
        // Sending the request
        doc["type"] = "request";
        serializeJson(doc,Serial);
        // Reading the response
        boolean messageReady = false;
        String message = "";
        while(messageReady == false) {
                                        // blocking but that's ok
                                        if(Serial.available()) {
                                                        message = Serial.readString();
                                                        messageReady = true;
                                                        }
                                        }
        // Attempt to deserialize the JSON-formatted message
        DeserializationError error = deserializeJson(doc,message);
        if(error) {
                Serial.print(F("deserializeJson() failed: "));
                Serial.println(error.c_str());
                return;
                }
        hb1 = doc["hb1"];
        hb2 = doc["hb2"];
        hb3 = doc["hb3"];
        flg = doc["RCWL"];
        // Prepare the data for serving it over HTTP
        String output = String(hb1) + "," + String(hb2) + "," + String(hb3) + "," + String(flg);
        // Serve the data as plain text, for example
        server.send(200,"text/plain",output);}
```

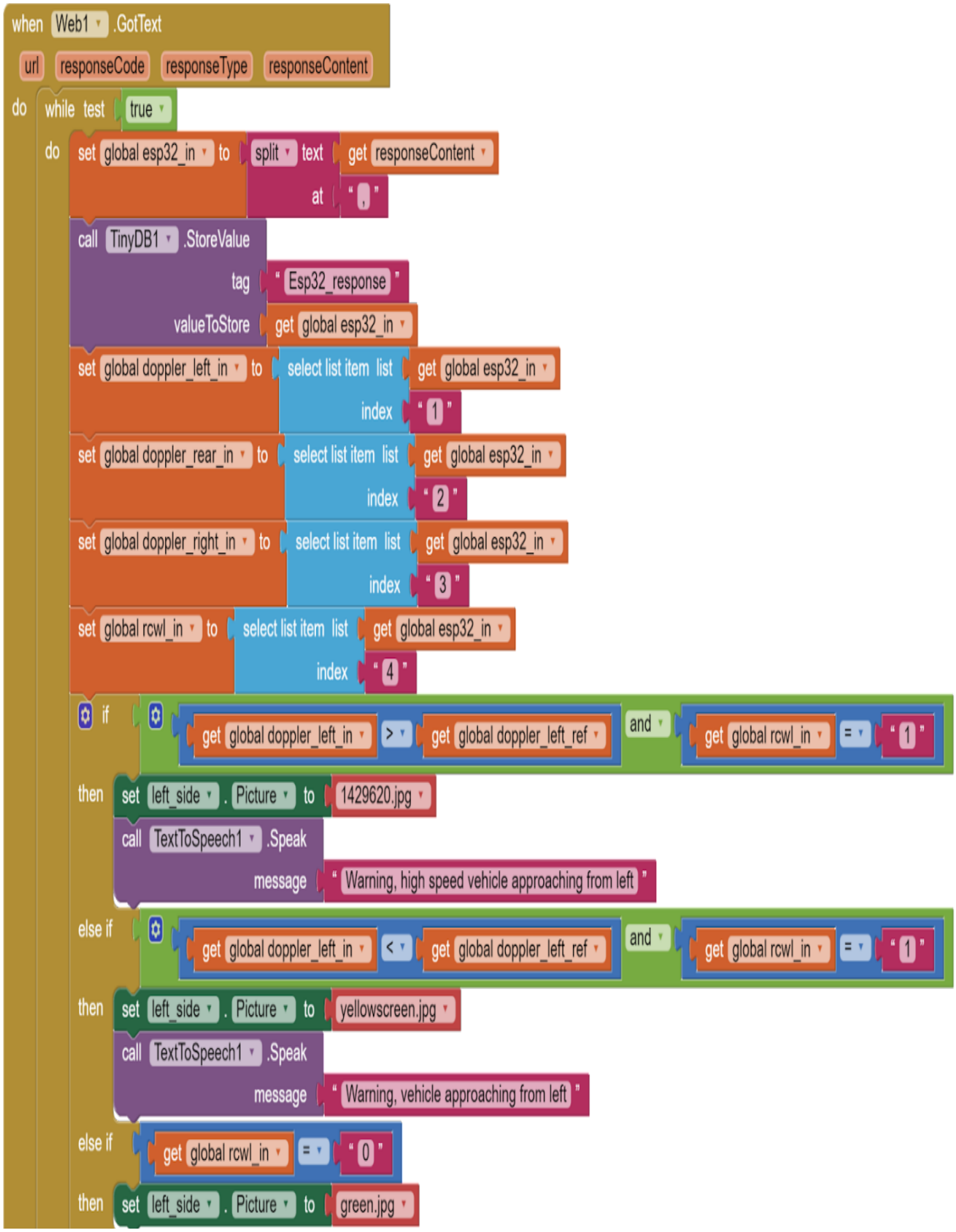

Back-end Code Blocks:

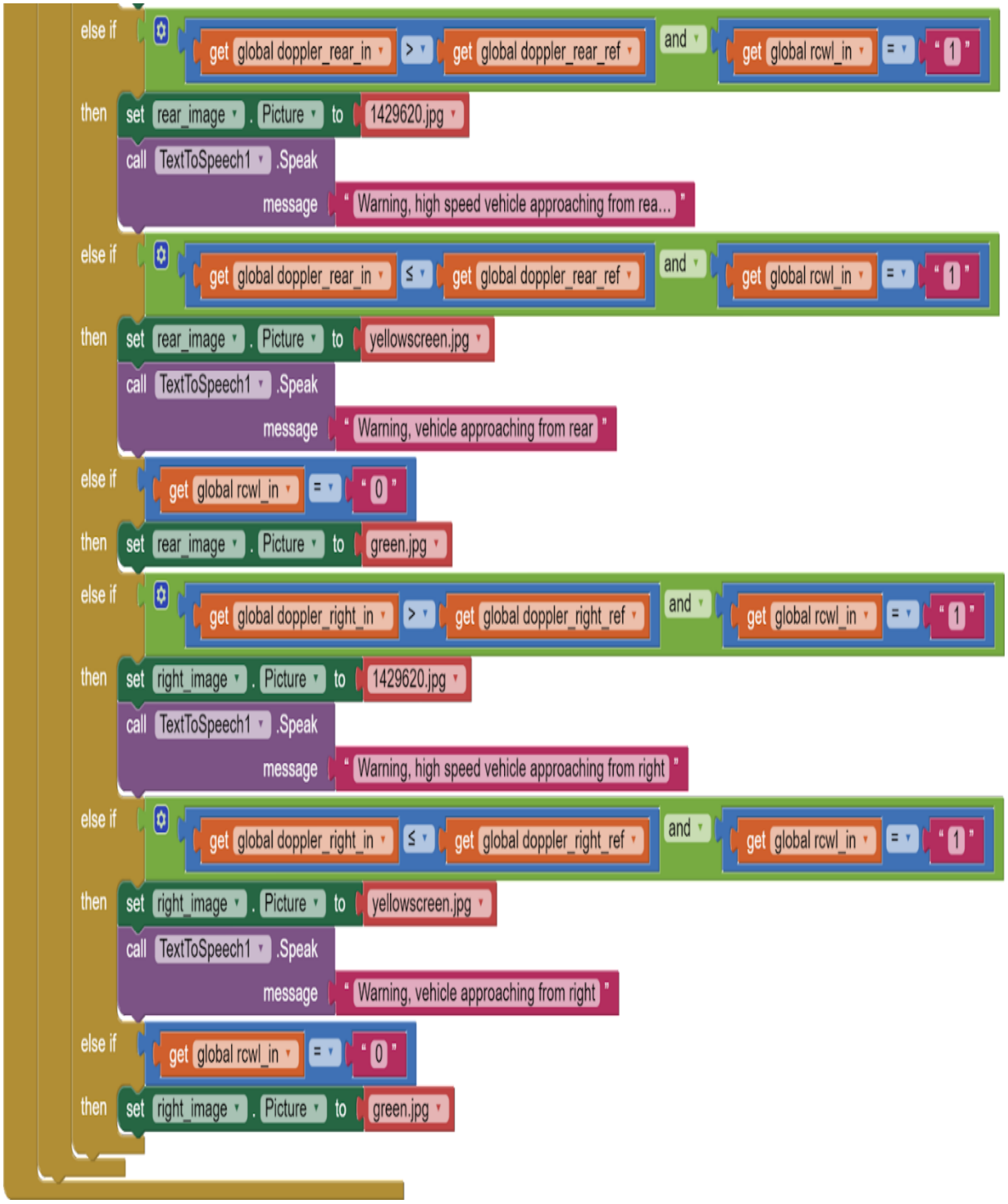




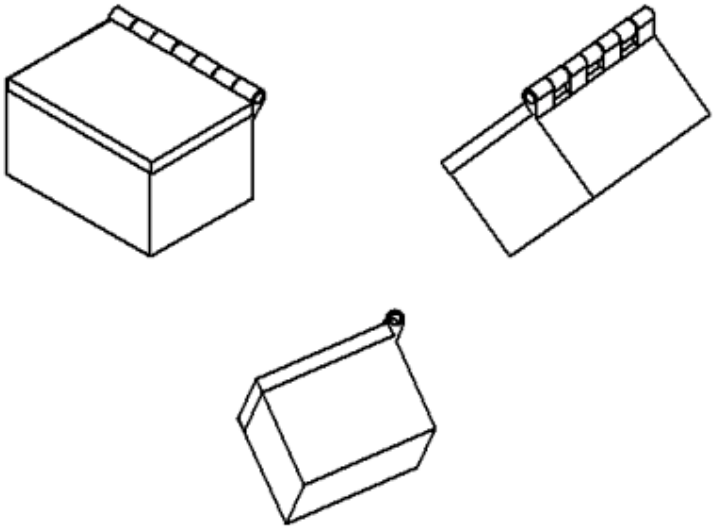





when Web1 .GotText
url responseCode responseType responseContent
do while test true
do set global esp32_in to split text get responseContent
at ","
call TinyDB1 .StoreValue
tag "Esp32_response"
valueToStore get global esp32_in
set global doppler_left_in to select list item list get global esp32_in
index "1"
set global doppler_rear_in to select list item list get global esp32_in
index "2"
set global doppler_right_in to select list item list get global esp32_in
index "3"
set global rcwl_in to select list item list get global esp32_in
index "4"
if get global doppler_left_in > get global doppler_left_ref and get global rcwl_in = "1"
then set left_side . Picture to 1429620.jpg
call TextToSpeech1 .Speak
message "Warning, high speed vehicle approaching from left"
else if get global doppler_left_in < get global doppler_left_ref and get global rcwl_in = "1"
then set left_side . Picture to yellowscreen.jpg
call TextToSpeech1 .Speak
message "Warning, vehicle approaching from left"
else if get global rcwl_in = "0"
then set left_side . Picture to green.jpg


else if
get global doppler_rear_in > get global doppler_rear_ref and get global rcwl_in = "1"
then set rear_image . Picture to 1429620.jpg
call TextToSpeech1 .Speak
message "Warning, high speed vehicle approaching from rea..."
else if
get global doppler_rear_in ≤ get global doppler_rear_ref and get global rcwl_in = "1"
then set rear_image . Picture to yellowscreen.jpg
call TextToSpeech1 .Speak
message "Warning, vehicle approaching from rear"
else if
get global rcwl_in = "0"
then set rear_image . Picture to green.jpg
else if
get global doppler_right_in > get global doppler_right_ref and get global rcwl_in = "1"
then set right_image . Picture to 1429620.jpg
call TextToSpeech1 .Speak
message "Warning, high speed vehicle approaching from right"
else if
get global doppler_right_in ≤ get global doppler_right_ref and get global rcwl_in = "1"
then set right_image . Picture to yellowscreen.jpg
call TextToSpeech1 .Speak
message "Warning, vehicle approaching from right"
else if
get global rcwl_in = "0"
then set right_image . Picture to green.jpg

Concept of Bicycle and sensor case CAD design:



Literature

- https://appinventor.mit.edu/
- https://github.com/TMRh20/AnalogFrequency/blob/master/src/AnalogFrequency.h
- https://github.com/Jorge-Mendes/Agro-Shield/blob/master/OtherRequiredLibraries/FreqPeriod/FreqPeriod.h
- https://github.com/jdesbonnet/RCWL-0516
- https://www.youtube.com/watch?v=MpVRHEUV3Xo&ab_channel=Robert%27sSmorgasbord
- https://en.wikipedia.org/wiki/Doppler_effect
- https://www.theengineeringprojects.com/wp-content/uploads/2017/07/ATmega328-Pinout.png
- https://www.researchgate.net/profile/Samarjith-Biswas/publication/325115625/figure/fig3/AS:625887674380289@1526234658056/Arduino-UNO-The-Arduino-program-contains-two-main-parts-setup-and-loop-The-name.png