Zagazig University

Faculty of Engineering

Computer and Systems Engineering

Graduation Project

Advanced Driver Assistance System (ADAS)

Supervised by

Dr. Elshaimaa Nabil Nada

Authored By

Ahmad El-sayed Abdullah

Ahmed Ehab Gamil Ali

Abbas Ibrahim Abbas

Eslam Fawzi Metwally

Muhammad Essam Badawy

Moaz Talat Abdelmaged

Neveen Abbas Hassan

Hagar Ahmed Mahmoud

Hagar Tarek El-Hassanein

Hala Reda Elkarrany

Hoda Osama Ibrahim

Acknowledgment

It has been a great opportunity to get a lot of real-world project experience as well as knowledge about how to design and analyze real projects. We would like to express our gratitude to Allah for enabling us to complete our project and reach this learning point.

We would like to express our deepest appreciation to Dr. Elshaimaa Nabil Nada Ph.D. Professor, Computer and systems dep., for her supervision in harmony with us and guidance, continuous encouragement, and support during this project. We are deeply grateful to all who supported us in all aspects.

Table of Contents

Abbreviations

Advanced Driver Assistance Systems	ADAS
Blind Spot Warning	BSW
Traffic Sign Recognition	TSR
Light Detection and Ranging	LiDAR
Field Of View	FOV
Reset Control Clock	RCC
Digital Input Output	DIO
Sys Tick timer	STK
Nested vectored interrupt controller	NVIC
Electronic Control Unit	ECU
Universal synchronous asynchronous receiver transmitter	USART
Serial peripheral interface	SPI
Bluetooth Low Energy	BLE

Introduction to ADAS

Today, the automotive companies would like to help you drive more comfortably, more easily and that is why these companies make luxury vehicles. So, to make it luxurious ADAS comes over.

Advanced Driver Assistance Systems, or ADAS, are a collection of technologies designed to enhance vehicle safety and improve driver experience in the driving process. These systems use cameras, sensors, and other advanced technology to help drivers avoid accidents and navigate roads more safely by monitoring the vehicle's surroundings and providing feedback to the driver.

According to statistics, over 90% of accidents are caused by human error. ADAS can help reduce this number by providing drivers with real-time information about their surroundings and alerting them to potential dangers. This technology has the potential to save countless lives and make our roads safer for everyone.

ADAS has become increasingly popular in recent years, as automakers look for ways to differentiate their vehicles and improve safety ratings.

Here a collection of some technologies designed in ADAS:

- Lane Departure Warning (LDW): Monitors the vehicle's position within the lane and alerts the driver if the vehicle starts to drift out of the lane without signaling.
- Blind Spot Warning (BSW): Alerts the driver when there are vehicles in the blind spots, usually through visual or audible warnings.
- Traffic Sign Recognition (TSR): Utilizes cameras and image processing to identify and interpret traffic signs, displaying relevant information to the driver.
- Forward Collision Warning (FCW): Uses sensors to detect the distance between your vehicle and the vehicle ahead, providing an alert if a potential collision is detected.
- …. etc.

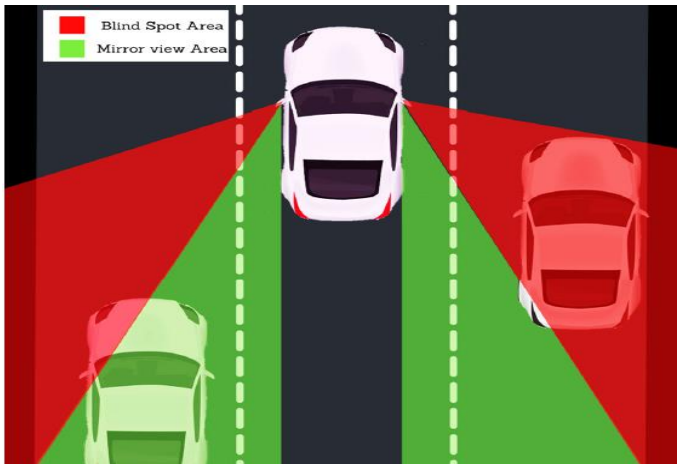
Blind Spot Problem

Blind spots are a serious problem for drivers, as they limit visibility and increase the risk of accidents. Blind spots are areas around a vehicle that cannot be seen by the driver, even with the use of mirrors.

According to the National Highway Traffic Safety Administration (NHTSA), blind spot-related accidents cause an estimated 840,000 injuries and 300 deaths each year in the United States alone.

Blind spots can be especially dangerous when changing lanes or making turns. In fact, NHTSA data shows that approximately 25% of lane-change crashes involve a vehicle in a blind spot. This is why it is crucial for drivers to be aware of blind spots and take extra precautions when navigating the road.

Real-life examples highlight the dangers of blind spots. For instance, a study by the American Automobile Association (AAA) found that blind spot-related accidents are more likely to occur when changing lanes on highways or turning at intersections. In addition, large trucks and SUVs have larger blind spots than smaller cars, making them more prone to accidents.



Volvo's Solution: Blind Spot Warning System

Blind Spot Warning System, a system of protection developed by Volvo. This system was first introduced on the 2001 Volvo SCC concept car, then placed into production on the 2003 Volvo XC90 SUV and produced a visible alert when a car entered the blind spot while a driver was switching lanes, using cameras and radar sensors mounted on the door mirror housings to check the blind spot area for an impending collision. Volvo won an Autocar Safety and Technology award for the introduction of this feature.

LiDAR vs. Camera-Based Systems

LiDAR and camera-based systems are two technologies used to detect blind spots in vehicles.

While both systems have their advantages and disadvantages, LiDAR is generally considered more accurate and reliable than cameras.

For blind spot monitoring is introduced by using vehicle-mounted cameras. A blind spot monitoring system based on direct vision is using RNN (Recurrent Neural Network) model to predict the number of cars turning up in blind spots:

- Range of Sensor: detection range is up to 20 m in length behind cameras and 4 m across both sides.
- FOV: the field of view is up to 360 deg.
- Alert type: visual alert (Video)

For blind spot monitoring is introduced by using LiDAR technology:

- Range of Sensor: detection range is up to 20 m.

- Latency: system latency is less than 500 msec, which is achieved from electromagnetic waves.
- FOV: the field of view is up to 180 deg.
- Alert type: visual alert (LEDs and LCD TFT screen) and audio (Buzzer).

LiDAR uses laser beams to detect the surrounding environment, allowing it to detect objects even in low-light conditions. Cameras, on the other hand, rely on visual cues and can be affected by glare or other types of interference.

Hardware Components

TF Luna LiDAR Sensor Methodology

Description

Principle of Distance Measurement: Based on Time-of-Flight principle (TOF). TF Luna emits modulation wave of near infrared ray on a periodic basis, which will be reflected after contacting object. TF Luna obtains the time of flight by measuring round-trip phase difference and then calculates relative range between the product and the detection object, as shown in Figure 1

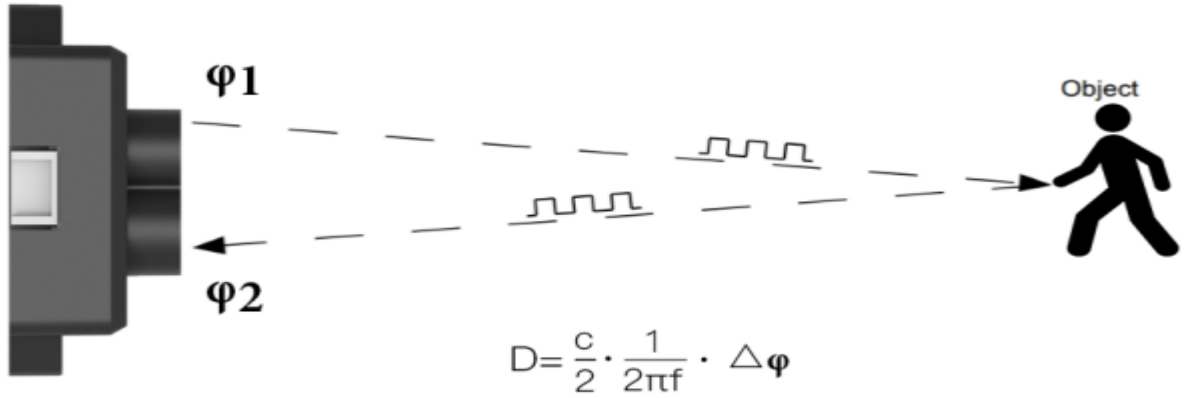


Figure 1 Schematics of TOF Principle

Key Characteristic Parameters

Description	Parameter Value
Operating Range	Up to 8 meters
Accuracy	±6cm@ (0.2-3m) ±2%@ (3m-8m)
Measurement unit	cm
Range resolution	1cm
FOV	2°
Frame rate	1 to 250Hz (adjustable)

Note that only frame rates meeting the formula 250/n (where n is a positive integer) can be set. The default frame rate is 100 Hz.

Distance Measurement Characteristics

The detection blind zone of TF-Luna, 0-20cm, within which the output data is unreliable.

The operating range of TF-Luna detecting black target with 10% reflectivity, 0.2-2.5m。

The operating range of TF-Luna detecting white target with 90% reflectivity, 0.2-8m。

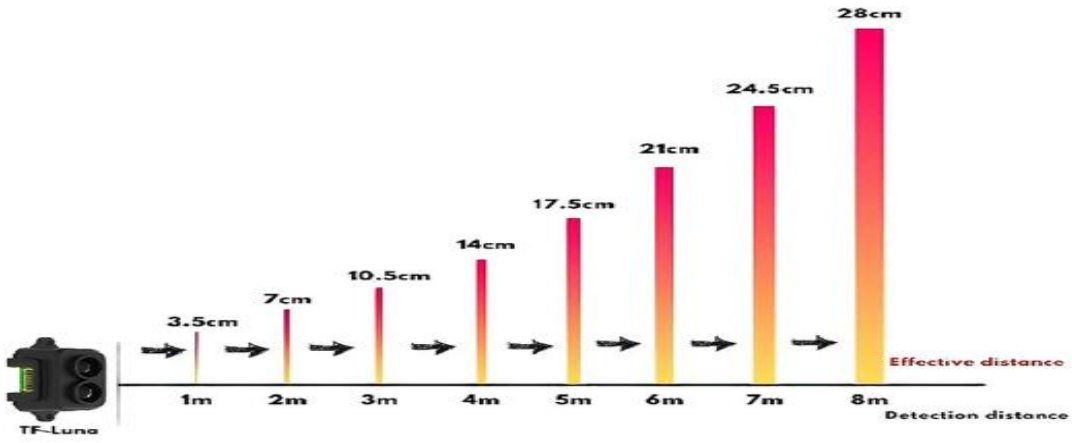
The diameter of light spot depends on the FOV of TF-Luna (the term of FOV generally refers to the smaller value between the receiving angle and the transmitting angle), which is calculated as follows**: d=2∗D·tanβ**

In the formula above, d is the diameter of light spot; D is detecting range; β is the value of the half receiving angle of TF-Luna, 1°.

Imagine TF-Luna as having a "viewing angle" in front of it, much like a cone-shaped field. This field extends from the center line of the sensor, spreading out at an angle of 2 degrees on either side.

When TF-Luna emits a near-infrared light wave and receives its reflection, it can detect objects within this narrow cone-shaped field. Any object that falls within a 2-degree angle from the center line of TF-Luna can be detected by the sensor.

D	1m	2m	3m	4m	5m	6m	7m	8m
d	3.5 cm	7cm	10.5cm	14cm	17.5cm	21cm	24.5cm	28cm



Electrical Characteristics

Table 4: Major Electrical Parameters of TF-Luna

Supply voltage	5V±0.1V
Average current	≤70mA
Peak current **Average power**	150mA 350mW
Communication level	LVTTL (3.3V)

Note that the communication level of TF-Luna is LVTTL (Low Voltage Transistor-Transistor Logic) with a voltage level of 3.3V. LVTTL is a logic family commonly used for digital communication interfaces.

In the case of TF-Luna, LVTTL refers to the voltage levels used for transmitting and receiving data between the sensor and other devices or microcontrollers. The voltage level of 3.3V indicates that TF-Luna operates with a logic high voltage of approximately 3.3 volts and a logic low voltage close to 0 volts.

Functional Descriptions and configuration

Description about Line Sequence and Connection

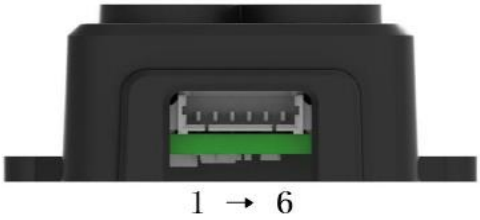


Figure 4 TF-Luna's pin numbers

Table 6: The Function and Connection Description of each pin

No.	Function	Description
1	+5V	Power supply
2	RXD/SDA	Receiving/Data
3	TXD/SCL	Transmitting/Clock
4	GND	Ground
5	Configuration Input	Ground: I2C mode /3.3V: Serial port Communications mode
6	Multiplexing output	Default: on/off mode output I2C mode: Data availability signal on but not switching value mode

Serial Communication Protocol

TF-Luna adopts the serial port data communication protocol, as given in Table 6.

Table 6: Data Communication Protocol of TF-Luna

Communication interface	UART
Default baud rate	115200
Data bit	8
Stop bit	1
Parity check	None

Data Output Format of Serial port

Table 7: Data Format and Code Explanation

Byte0 -1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Byte8
0x59 59	Dist_L	Dist_H	Strength_L	Strength_H	Temp_L	Temp_H	Checksum

Data code explanation	
Byte0	0x59, frame header, same for each frame
Byte1	0x59, frame header, same for each frame
Byte2	Dist_L distance value low 8 bits
Byte3	Dist_H distance value high 8 bits
Byte4	Strength_L low 8 bits
Byte5	Strength_H high 8 bits
Byte6	Temp_L low 8 bits
Byte7	Temp_H high 8 bits
Byte8	Checksum is the lower 8 bits of the cumulative sum of the numbers of the first 8 bytes.

So, to reveal the required measured data by the TF Luna sensor, two bytes that are the frame header (**0x59**) must be found.

Note that the measured values are spitted into two bytes and the arrangement of the two bytes has the lowest byte first then the highest byte.

Dist. (Distance): Represents the output of the distance value detected by TF-Luna, with the unit in cm by default. This value is interpreted into the decimal value in the range of 0-1200. When the signal strength is lower than 100, the detection is unreliable, TF-Luna will set distance value to -1.
Strength: Represents the signal strength with the default value in the range of 0-65535. After the distance mode is set, the longer the measurement distance is, the lower the signal strength will be; the lower the reflectivity is, the lower the signal strength will be.
Temp(Temperature): Represents the chip temperature of TF-Luna.

Degree centigrade = Temp / 8 -256

User-defined Parameter Configuration

Note that all commands are sent as hexadecimal digits (HEX).

Table 8: Command frame definition

Byte0	Byte1	Byte2	Byte3 ~ ByteN-2	ByteN-1
Head	Len	ID	Payload	Checksum

Remarks	
Byte0	Head: frame header（0x5A）
Byte1	Len: the total length of the frame（include Head and Checksum，unit: byte）
Byte2	ID: identifier code of command
Byte3-N-2	Data: data segment. Little endian format
ByteN-1	Checksum: sum of all bytes from Head to payload. Lower 8 bits

Command convention:

- To send commands each command must have the frame header **0x5A** in byte 0**.**

- Byte 1 is the frame total length (no. of frame bytes including the length byte).
  For example: System reset command (**0x5A 0x04 0x02 0x60**), no of bytes including the Len byte equal to 4 bytes.

- Byte 2 is the ID of the command, see table 9 to reveal commands IDs and lengths.

- Byte 3~N-2(Payload): multi-bytes data or command frame is transmitted in **little endian format.**
  For example, decimal number 1000 is equivalent to **0x03E8** in hexadecimal. Then the representation in little endian will be (0xE8 0x03) Then it will be saved in the data or command frame as: 0x5A 0x06 0x03 0xE8 0x03 0x4E.
  one more example, decimal number 115200 is equivalent to **0x1C200** in HEX it will be represented as: (0x00 0xC2 0x01).

  Caution: After setting parameters, the 'Save setting' command needs to be sent.

Table 9: General Parameter Configuration and Description

Parameters	Command	Response	Remark	Default setting
Obtain firmware version	5A 04 01 5F	5A 07 01 V1 V2 V3 SU	Version V3.V2.V1	
System reset	5A 04 02 60	5A 05 02 00 60	Succeeded	/
		5A 05 02 01 61	Failed	/
Frame rate	5A 06 03 LL HH SU	5A 06 03 LL HH SU	1-250Hz①	100Hz
Trigger detection	5A 04 04 62	Data frame	After setting the frame rate to 0, detection can be triggered with this command	
Output format	5A 05 05 01 65	5A 05 05 01 65	Standard 9 bytes(cm)	√
	5A 05 05 02 66	5A 05 05 02 66	Pixhawk	/
	5A 05 05 06 6A	5A 05 05 06 6A	Standard 9 bytes (mm)	/
Baud rate	5A 08 06 H1 H2 H3 H4 SU	5A 08 06 H1 H2 H3 H4 SU	Set baud rate②	115200
Enable/Disable output	5A 05 07 00 66	5A 05 07 00 66	Disable data output	/
	5A 05 07 01 67	5A 05 07 01 67	Enable data output	√
Obtain Data Frame	5A 05 00 01 60	Data Frame(9bytes-cm)	Only works in I2C mode	/
	5A 05 00 06 65	Date Frame(9bytes-mm)		
Restore factory settings	5A 04 10 6E	5A 05 10 00 6E	Succeeded	
		5A 05 10 01 6F	Failed	
Save settings	5A 04 11 6F③	5A 05 11 00 6F	Succeeded	
		5A 05 11 01 70	Failed	

Note: Bytes with yellow undertone represent checksum.

STM32f10x Microcontroller

STM32f10x microcontroller, part of the STM32 family from STMicroelectronics, can be a powerful and suitable choice for implementing a Blind Spot Warning system. Here are a few reasons why:

- Processing Power: The STM32F10x microcontroller series offers a range of ARM Cortex-M3 cores, which provide significant processing power for handling complex algorithms and real-time processing required in a Blind Spot Warning system. The microcontroller's high clock speed and efficient architecture allow for fast and responsive data processing.

- Peripheral Support: The STM32F10x series comes with a variety of built-in peripherals, such as,RCC ,GPIOs, USARTs, and couple of core peripherals such as NVIC and Systick,etc…. which are crucial for interfacing with sensors, communication modules, and other components required in a Blind Spot Warning system. These peripherals simplify the integration of various sensors and actuators, enabling efficient data acquisition and processing.

- Communication Capabilities: Communication interfaces like UART has about 5 channels and two are available for our controller, SPI used for the interfacing with TFT LCD for visual Alert , I2C can be used to communicate with LiDAR sensor too, and CAN will be used in our vehicle future network, all these peripherals are available on STM32F10x microcontrollers. These interfaces allow seamless communication with other system components, including radar sensors, ultrasonic sensors, and display modules, which are commonly used in Blind Spot Warning systems.

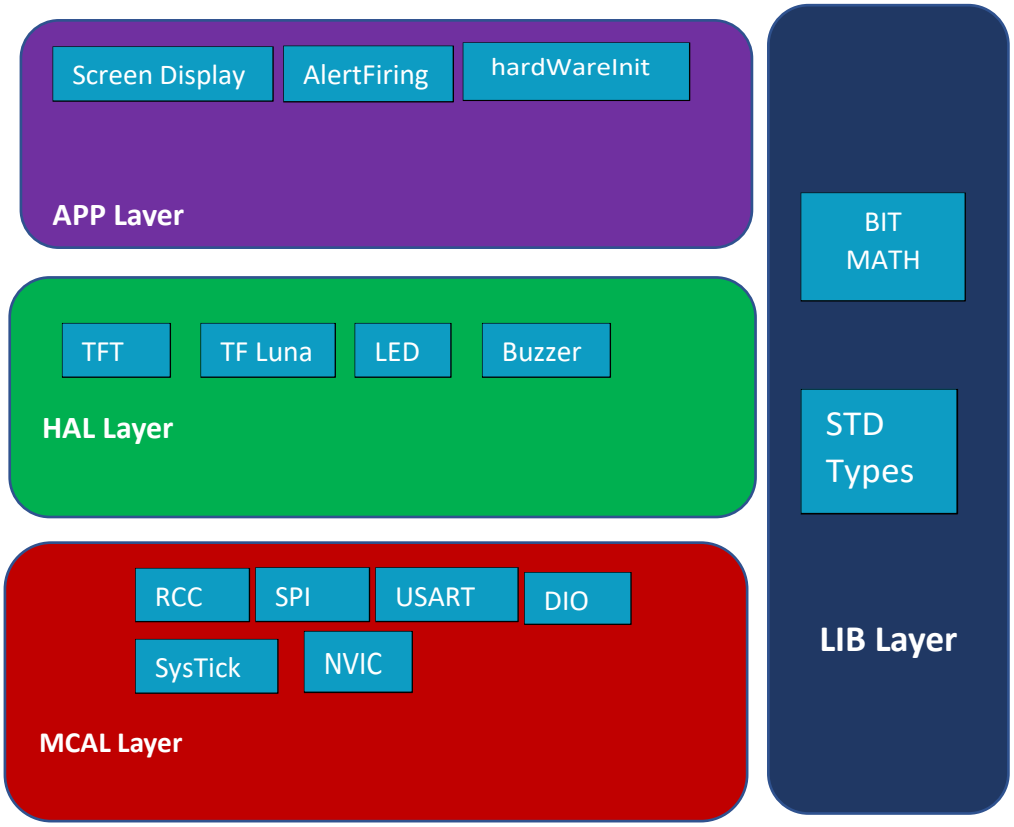
- Low Power Consumption: The STM32F10x series offers various low-power modes, enabling energy-efficient operation. This is advantageous for automotive applications like Blind Spot Warning systems, where power consumption needs to be minimized to ensure compatibility with the vehicle's electrical system and to optimize battery life.

- Development Ecosystem: STMicroelectronics provides a robust development ecosystem for STM32 microcontrollers, including integrated development environments (IDEs) like STM32CubeIDE, Keil, Eclipse and a comprehensive suite of software libraries and middleware. These resources simplify software development, accelerate the implementation of Blind Spot Warning algorithms, and facilitate system integration.

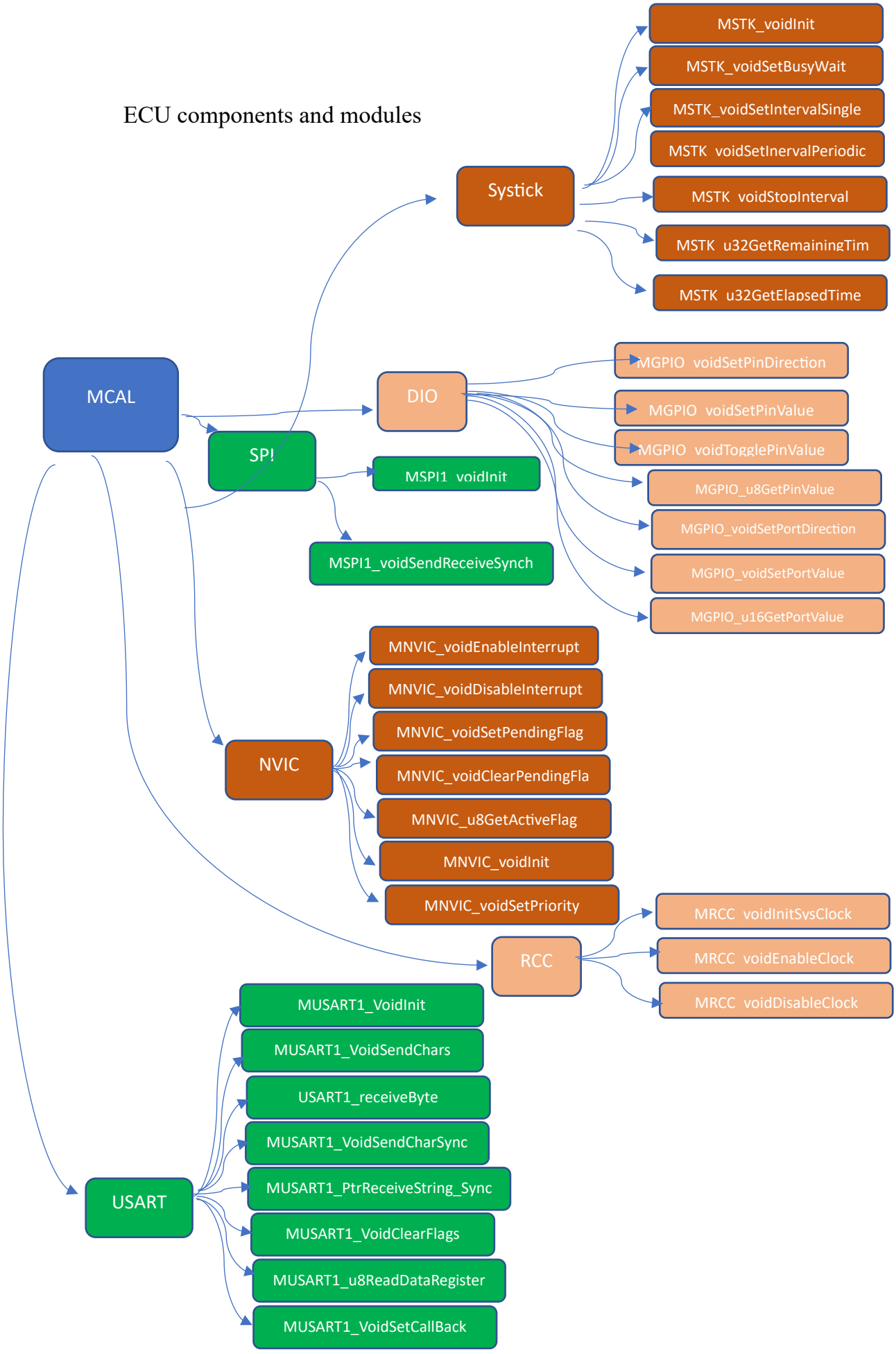
It's important to note that implementing a Blind Spot Warning system involves various components and considerations beyond just the microcontroller. The overall system design includes sensor selection, signal processing algorithms, sensor fusion, and interface with the vehicle's electrical system. However, the STM32F10x microcontroller can serve as a powerful and versatile foundation for building such a system.

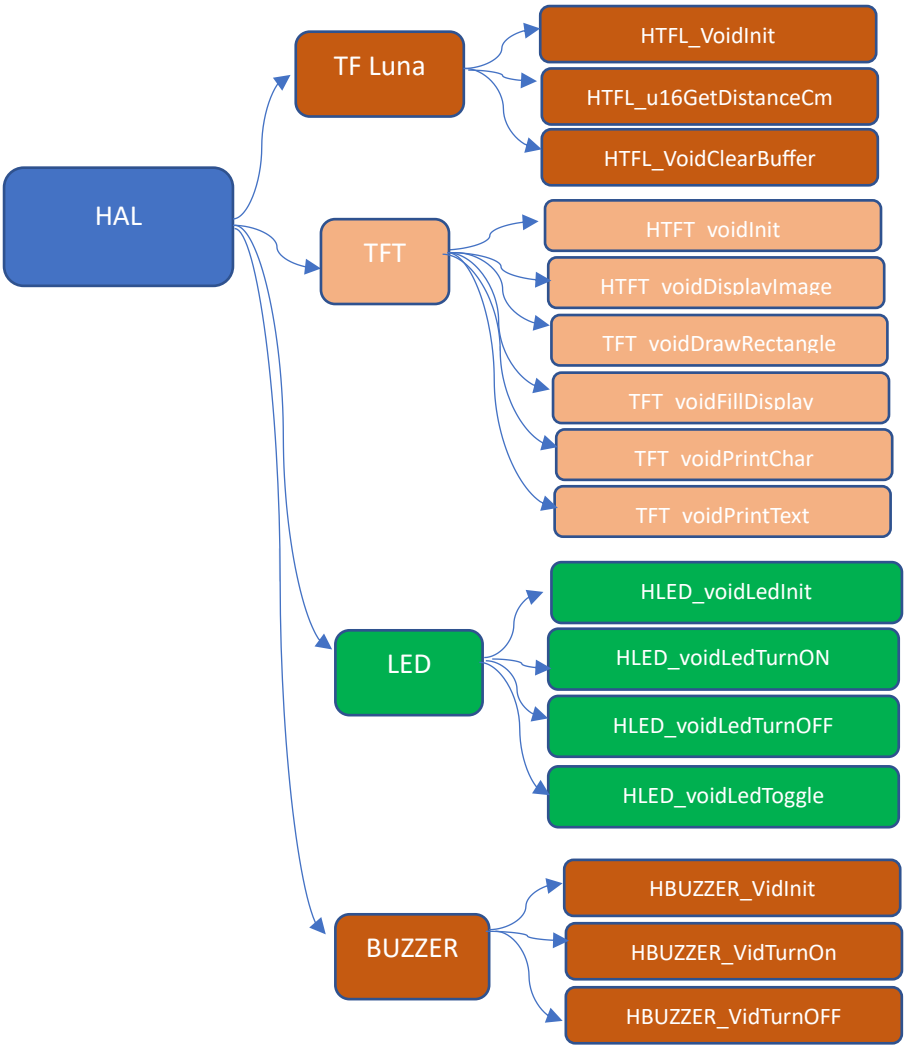
Static Design Analysis

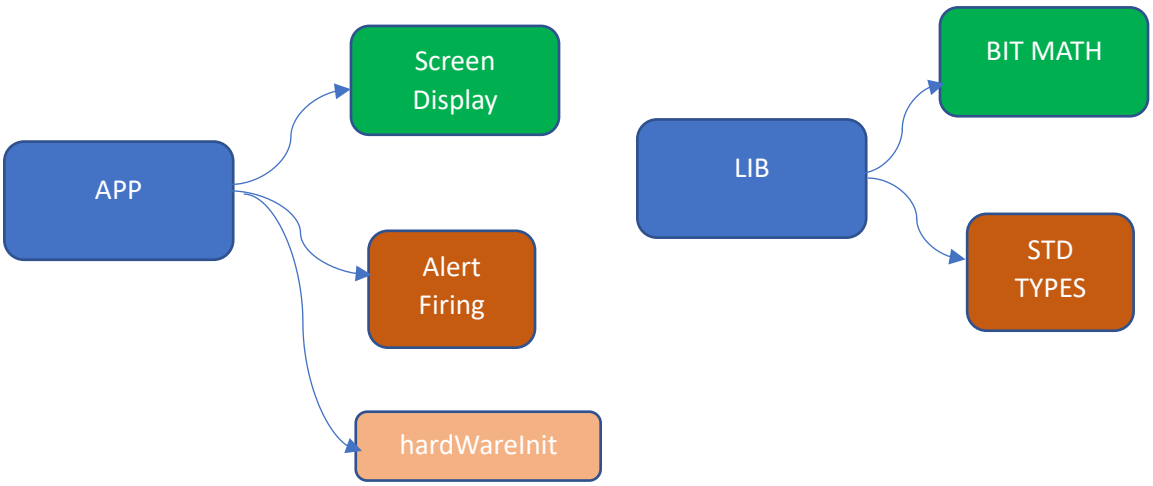
Layered Architecture:



ECU components and modules



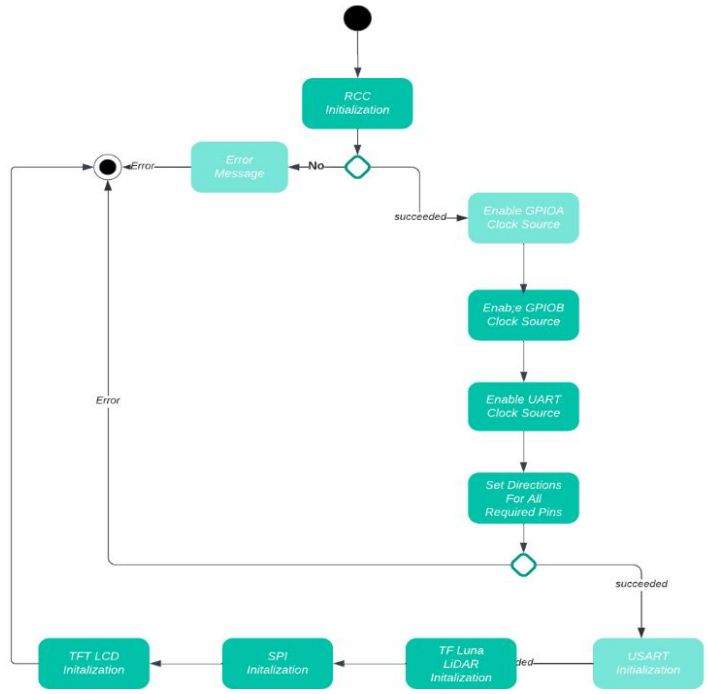

HAL
TF Luna
HTFL_VoidInit
HTFL_u16GetDistanceCm
HTFL_VoidClearBuffer
TFT
HTFT voidInit
HTFT voidDisplayImage
TFT voidDrawRectangle
TFT voidFillDisplay
TFT voidPrintChar
TFT voidPrintText
LED
HLED_voidLedInit
HLED_voidLedTurnON
HLED_voidLedTurnOFF
HLED_voidLedToggle
BUZZER
HBUZZER_VidInit
HBUZZER_VidTurnOn
HBUZZER_VidTurnOFF


APP
Screen Display
Alert Firing
hardWareInit
LIB
BIT MATH
STD TYPES

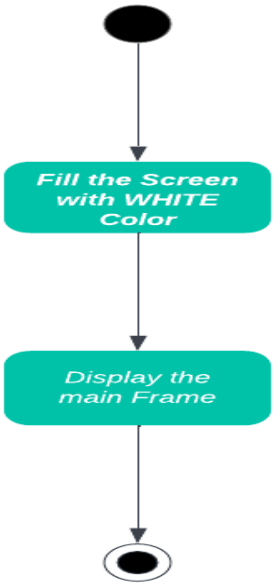
Dynamic Design Analysis

State Machine for ECU components

- hardware initialization

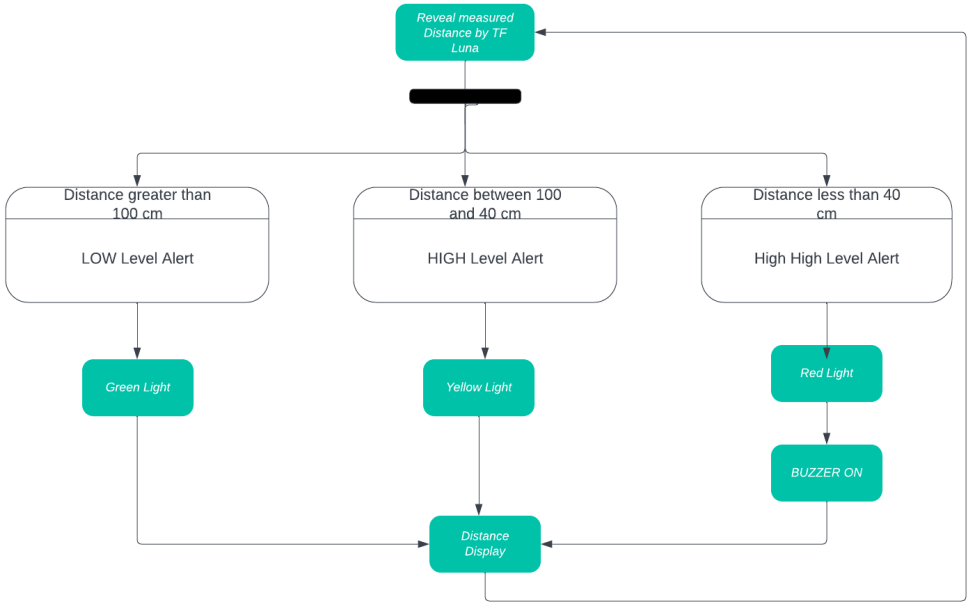


- screen display

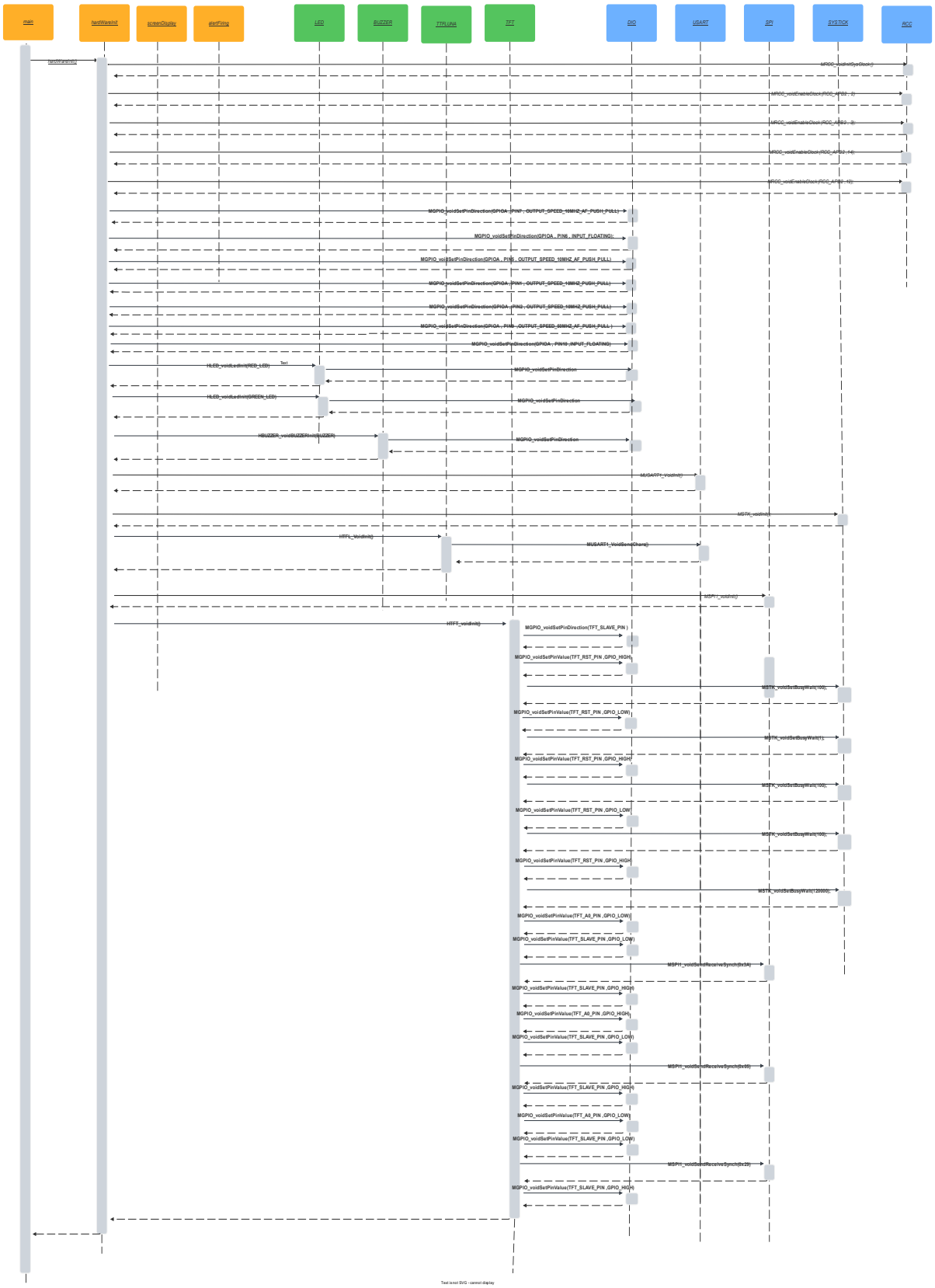


- Blind Spot Alert System

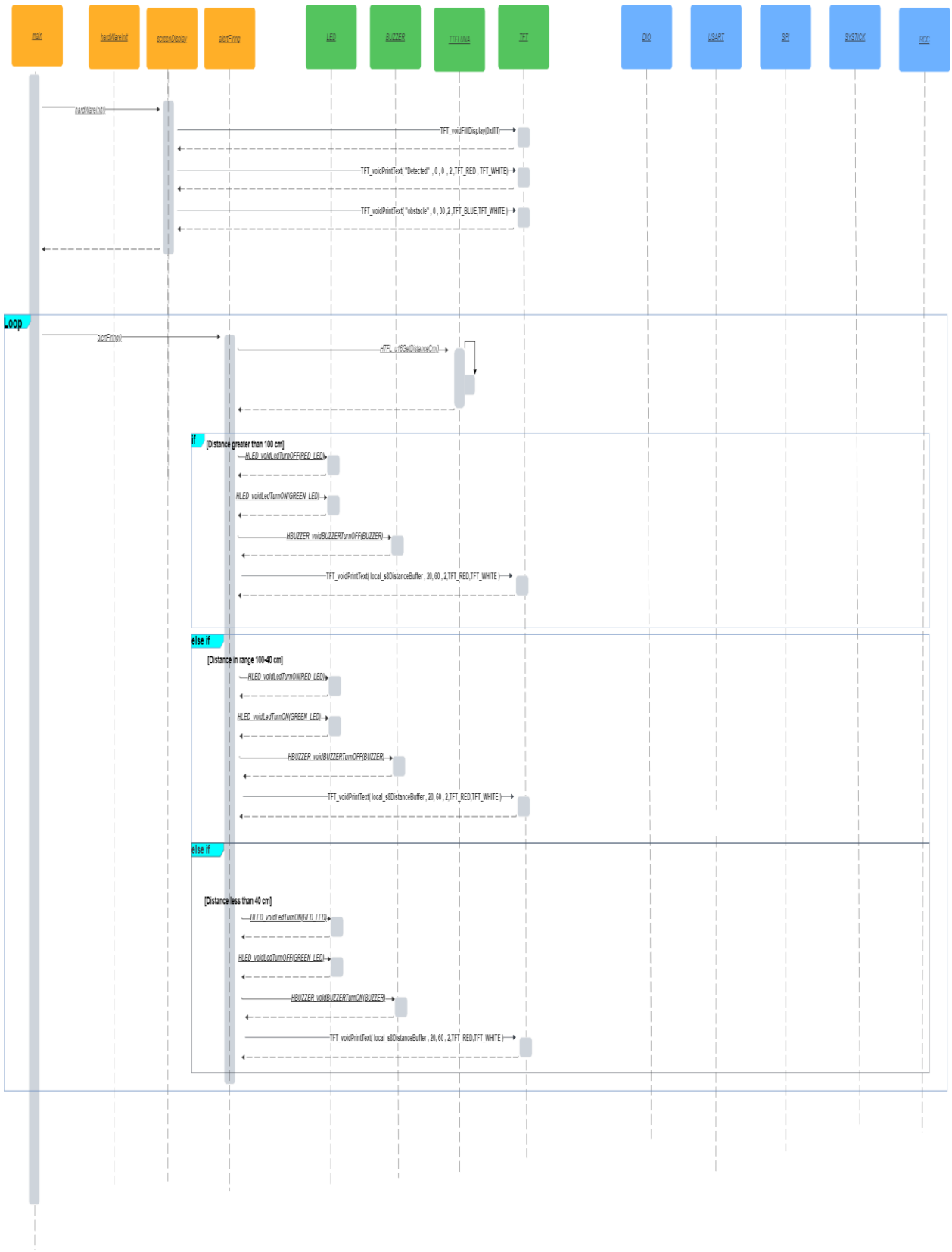
Function Name	alertFiring
Characteristics	Non-Reentrant ,Synchronous



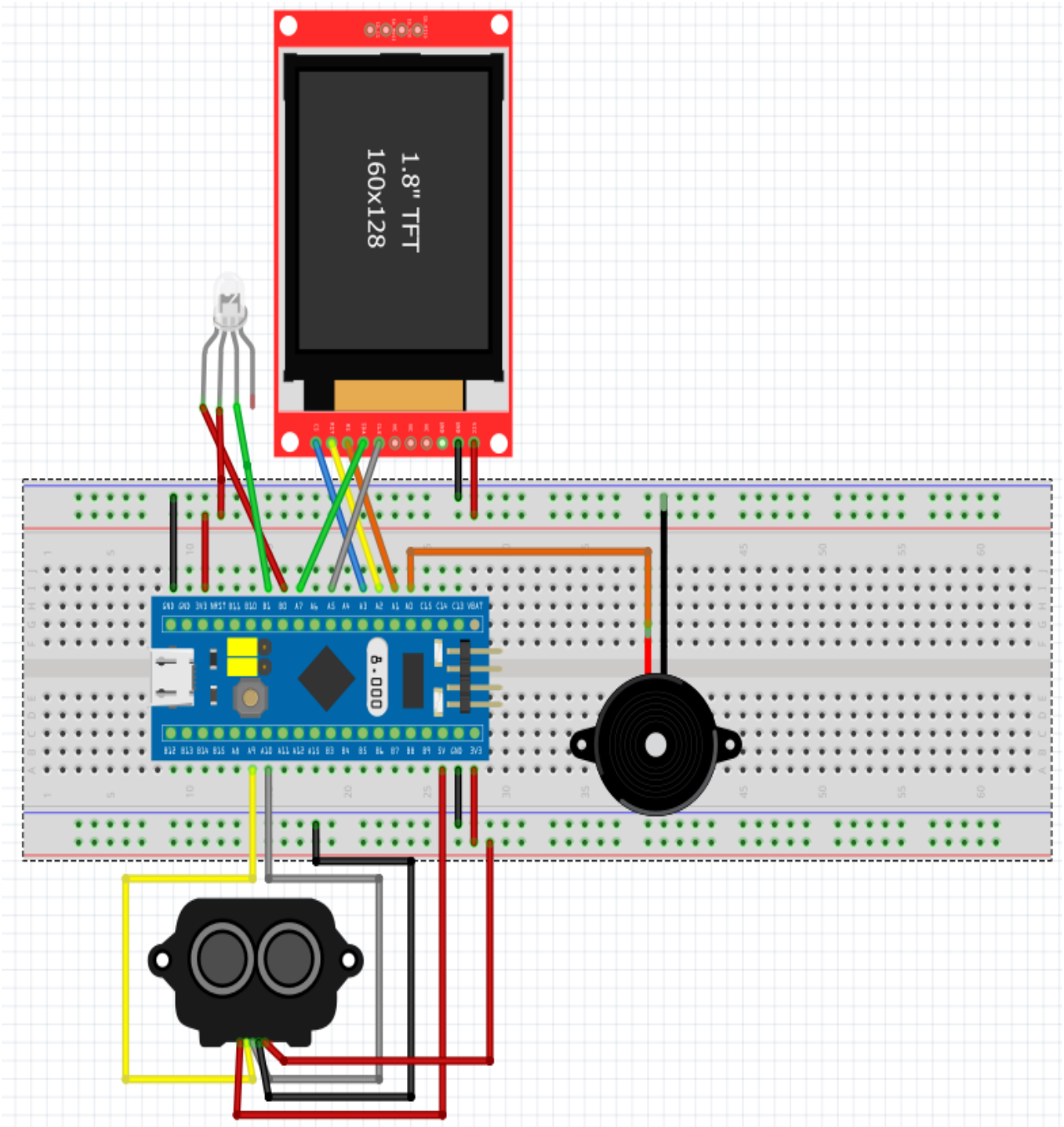
Sequence Diagram for ECU components



Cont. Sequence Diagram for ECU components.

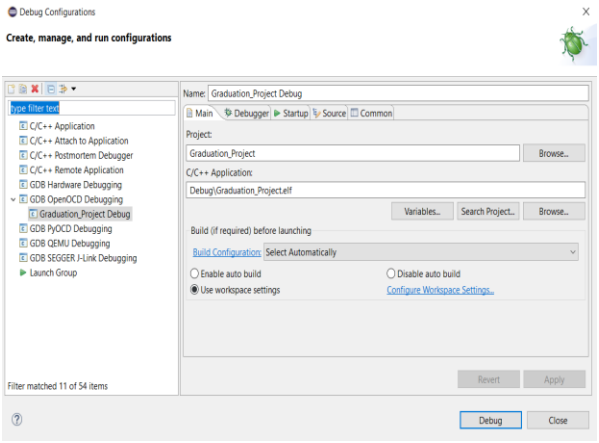


Circuit Wiring Connection.



System HW Debug and **Product Test using ECLIPSE.**

- Create new Debug session.



- Jump to main Entry point.

```
47    /* The Entry Point (main function) */
48 /*=======================================*/
49 int main (void)
50 {
51
52    hardWareInit();     /* CALL the HardWare initialization API */
53    screenDisplay();    /* CALL the Screen Display API          */
54
55
56    while(1)
57    {
58        alertFiring();  /* Call Alerting System API Forever */
59    }
60 return 0;
61 }
62
```

- Context switching to hardwareInit function.

```
74 void hardWareInit(void)
75 {
76    /* RCC Initialization */
77    MRCC_voidInitSysClock();
78    MRCC_voidEnableClock(RCC_APB2 , 2);     /* GPIOA */
79    MRCC_voidEnableClock(RCC_APB2 , 3);     /* GPIOB */
80    MRCC_voidEnableClock(RCC_APB2 ,14);     /* UART1  */
81    MRCC_voidEnableClock(RCC_APB2 ,12);     /* SPI1  */
82
83    /* Configure pins for SPI */
84    MGPIO_voidSetPinDirection(GPIOA , PIN7 , OUTPUT_SPEED_10MHZ_AF_PUSH_PULL); //MOSI
85    MGPIO_voidSetPinDirection(GPIOA , PIN6 , INPUT_FLOATING);           //MISO
86    MGPIO_voidSetPinDirection(GPIOA , PIN5 , OUTPUT_SPEED_10MHZ_AF_PUSH_PULL); //SCK
87
88    /* Configure pins for TFT */
89    MGPIO_voidSetPinDirection(GPIOA , PIN1 , OUTPUT_SPEED_10MHZ_PUSH_PULL);
90    MGPIO_voidSetPinDirection(GPIOA , PIN2 , OUTPUT_SPEED_10MHZ_PUSH_PULL);
91
92    /* USART PIN Initialization */
93    MGPIO_voidSetPinDirection(GPIOA , PIN9  ,OUTPUT_SPEED_50MHZ_AF_PUSH_PULL ); /* TX */
94    MGPIO_voidSetPinDirection(GPIOA , PIN10 ,    INPUT_FLOATING           ); /* RX */
95
96
97    /* ALERTS Pins Initialization */
98    HLED_voidLedInit(RED_LED);
99    HLED_voidLedInit(GREEN_LED);
100   HBUZZER_voidBUZZERInit(BUZZER);
101
102          /* UART Initialization */
103   MUSART1_VoidInit();
104
105
106          /* TF_Luna Initialization*/
107   HTFL_VoidInit();
108
109          /* Initialize SPI1 */
110   MSPI1_voidInit();
111
112          /* Initialize TFT */
113   HTFT_voidInit();
114
115 }
```

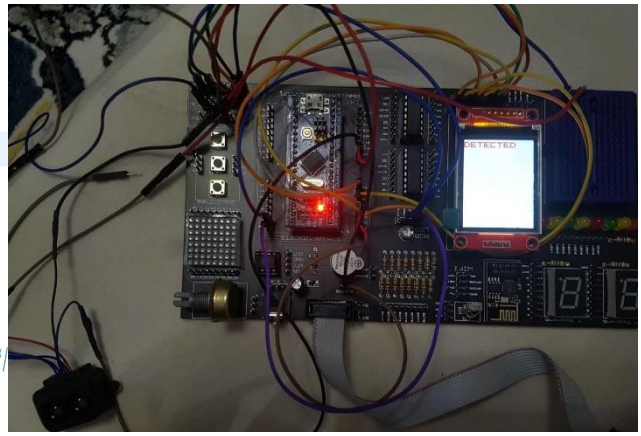
- Context switching back to main.

```
47    /* The Entry Point (main function) */
48 /*=======================================*/
49 int main (void)
50 {
51
52    hardWareInit();     /* CALL the HardWare initialization API */
53    screenDisplay();    /* CALL the Screen Display API          */
54
55
56    while(1)
57    {
58        alertFiring();  /* Call Alerting System API Forever */
59    }
60 return 0;
61 }
```

- Context switching to screenDisplay function.

```
129 void screenDisplay(void)
130 {
131    /* Set Background */
132    TFT_voidFillDisplay(0xffff);
133
134    /* Default printed text on the screen first and second line */
135    TFT_voidPrintText( "Detected"  , 0 , 0  , 2 ,TFT_RED , TFT_WHITE);
136    TFT_voidPrintText( "obstacle" , 0 , 30 ,2 ,TFT_BLUE,TFT_WHITE );
137 }
138 /*=============================================================================*/
139
```

- Product output after executing line 135



Product output after executing line 136

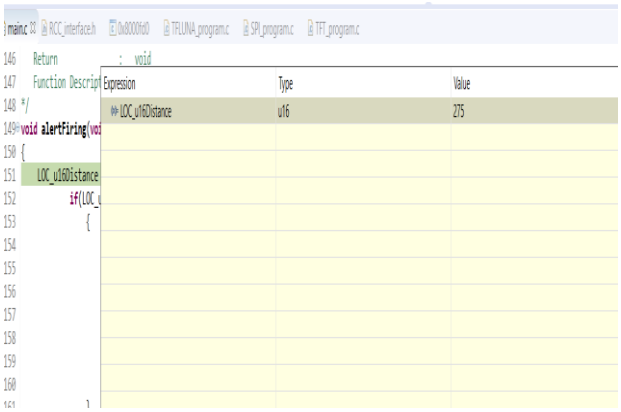
```
149 void alertFiring(void)
150 {
151     LOC_u16Distance = HTFL_u16GetDistanceCm();
152             if(LOC_u16Distance >= 100)
153                 {
154                     /* Alert Level  : LOW */
155                     HLED_voidLedTurnOFF(RED_LED);
156                     HLED_voidLedTurnON(GREEN_LED);
157                     HBUZZER_voidBUZZERTurnOFF(BUZZER);
158
159                     sprintf(local_s8DistanceBuffer, ": %.4i cm", LOC_u16Distance );
160                     TFT_voidPrintText( local_s8DistanceBuffer , 20, 60 , 2,TFT_RED,TFT_WHITE );
161                 }
162                 else if(LOC_u16Distance <100 && LOC_u16Distance >40 )
163                 {
164                     /* Alert Level : HIGH */
165                     HLED_voidLedTurnON(RED_LED);
166                     HLED_voidLedTurnON(GREEN_LED);
167                     HBUZZER_voidBUZZERTurnOFF(BUZZER);
168
169
170                     sprintf(local_s8DistanceBuffer, ": %.4i cm", LOC_u16Distance );
171                     TFT_voidPrintText( local_s8DistanceBuffer , 20 , 60 , 2,TFT_RED,TFT_WHITE );
172                 }
173                 else if(LOC_u16Distance <= 40)
174                 {
175                     /* Alert Level : HIGH HIGH */
176                     HLED_voidLedTurnON(RED_LED);
177                     HLED_voidLedTurnOFF(GREEN_LED);
178                     HBUZZER_voidBUZZERTurnON(BUZZER);
179                     _delay_ms(150);
180                     HBUZZER_voidBUZZERTurnOFF(BUZZER);
181
182
183                     sprintf(local_s8DistanceBuffer, ": %.4icm", LOC_u16Distance );
184                     TFT_voidPrintText( local_s8DistanceBuffer , 20 , 60 , 2,TFT_RED,TFT_WHITE );
185                 }
186
187 }
```

- Context switching back to main

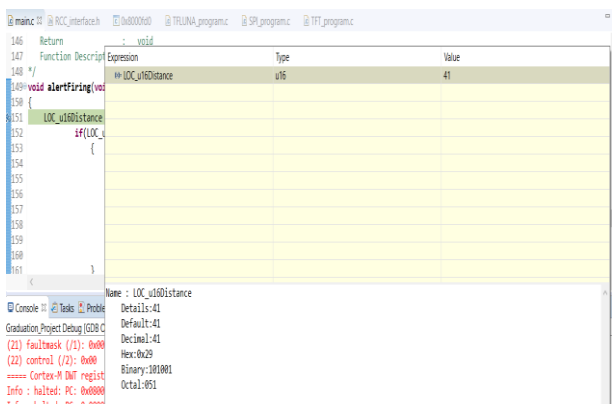
```
47     /* The Entry Point (main function) */
48 /*=====================================*/
49 int main (void)
50 {
51
52     hardWareInit();     /* CALL the HardWare initialization API */
53     screenDisplay();    /* CALL the Screen Display API          */
54
55
56     while(1)
57     {
58         alertFiring();  /* Call Alerting System API Forever */
59     }
60 return 0;
61 }
```

```
149 void alertFiring(void)
150 {
151     LOC_u16Distance = HTFL_u16GetDistanceCm();
152             if(LOC_u16Distance >= 100)
153                 {
154                     /* Alert Level  : LOW */
155                     HLED_voidLedTurnOFF(RED_LED);
156                     HLED_voidLedTurnON(GREEN_LED);
157                     HBUZZER_voidBUZZERTurnOFF(BUZZER);
158
159                     sprintf(local_s8DistanceBuffer, ": %.4i cm", LOC_u16Distance );
160                     TFT_voidPrintText( local_s8DistanceBuffer , 20, 60 , 2,TFT_RED,TFT_WHITE );
161                 }
```

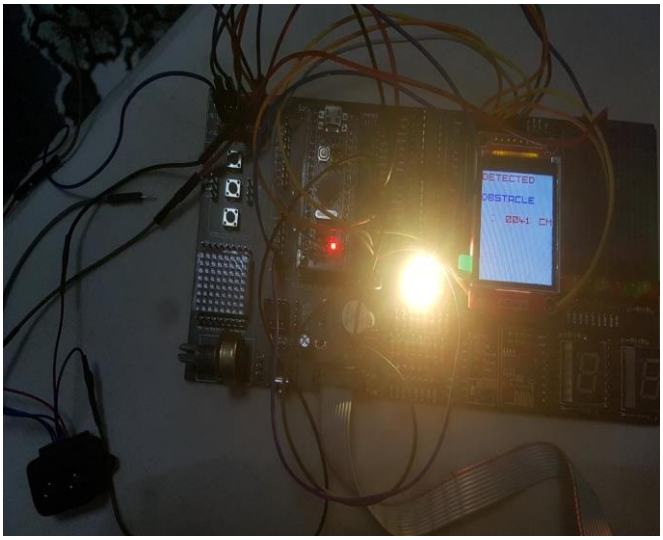
- Software detection reading.

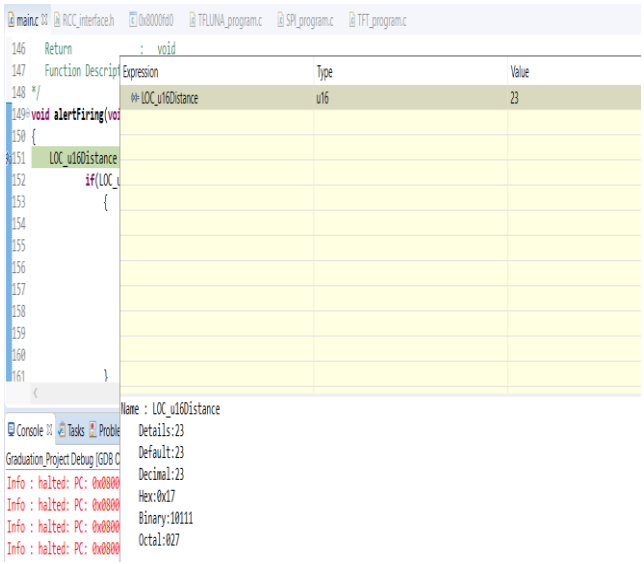


- High Level Alert

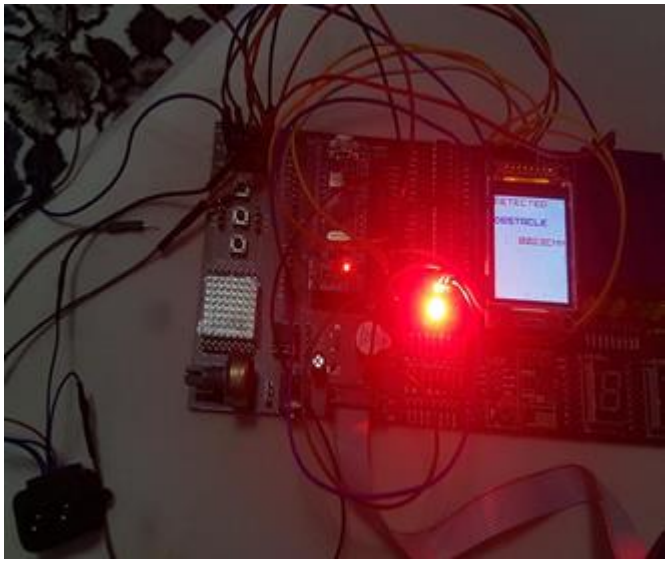


- Product output (LOW level Alert)





- Product alert output.



Summary

In conclusion, the Blind Spot Alert system developed by Bosch is an effective solution to address the safety concerns posed by blind spots in vehicles. By utilizing LiDAR and camera-based systems, the hardware components of the system work together to detect blind spots and alert drivers through visual and sound indicators. The static and dynamic design of the system ensures that it is reliable and responsive in detecting blind spots. Additionally, the debugging and testing process helps to ensure that the system is functioning properly.

It is important for the automotive industry to address the issue of blind spots and implement solutions such as the Blind Spot Alert system to improve overall safety on the road. With the increasing number of vehicles on the road, it is crucial to prioritize the safety of drivers and passengers alike.

Sign Recognition model

Abstract

Road safety is a critical concern for all drivers. And traffic signs are essential for guaranteeing secure and effective traffic flow. However, it can be difficult to recognize and understand traffic signs, particularly in bad weather or new surroundings. When it comes to detecting and identifying traffic signals, AI models offer an advanced and effective solution.

Introduction

Researchers with Artificial intelligence (AI) have made a significant contribution to the creation of innovative technology over the past years in a variety of fields (such as robots, health, transportation, and education). AI is widely used in the transportation industry to create automated and assistive driving systems. Advanced Driving assistance Systems (ADAS) are life-saving innovations that use many traffic data sources to provide a variety of driving aid functions, including autonomous emergency braking, driver attention alerts, speed adaption, and traffic sign detection and recognition.

Overview

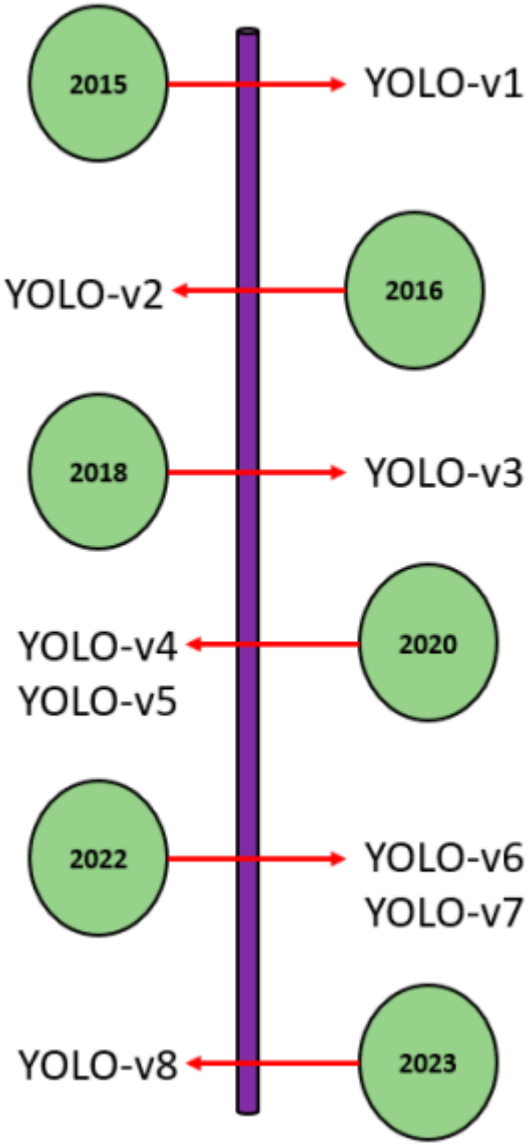
AI models can be trained to detect a wide range of traffic signs, including speed limits, stop signs, yield signs, and more. Overall, using AI models to detect traffic signs is a crucial step towards creating safer and more effective roads. We may anticipate seeing more advancements in the precision and dependability of these models as technology develops, which will help drivers and society at large even more. One of the important technologies that has contributed to speeding the development of ADAS and their real-time performance is computer vision, in particular. From an AI perspective, the key limiting factor to overpass AI challenges is the availability of high-quality data. In fact, only the availability of carefully designed data has made it possible for AI to achieve any successes that are on level with human performance. Google made significant progress in the field of image classification in 2014. Google made A new picture object classifier called Google Net that was trained on the ImageNet corpus. Since then, researchers and businesses using convolutional neural network (CNN) architectures have begun to recognize the significance of gathering and annotating high-quality data, leading to the improvement of this field.

The development of Artificial Intelligence (AI) has led to the rapid progress of self-driving cars. Companies such as Google, Tesla… have invested in the research and development of self-driving cars. Google is a leader in self-driving cars due to its strong AI foundation. Tesla was the first to apply self-driving technology to production. Other companies such as Volvo, Autoliv, Samsung, Baidu, Tencent, Alibaba, and Huawei have also made great strides in the field of self-driving cars. These companies have tested their cars on public roads and applied for patents in the field. Deep learning methods have been used to achieve outstanding performance in this task. Most studies proposed a traffic signs recognition approach based on a CNN algorithm.

YOLO was introduced to the computer vision community via a paper release in 2015 by Joseph Redmon et al. You Only Look Once (YOLO) was developed to create a one step process involving

detection and classification. Bounding box and class predictions are made after one evaluation of the input image.

The idea of YOLO differs from other traditional systems in that bounding box predictions and class predictions are done simultaneously.



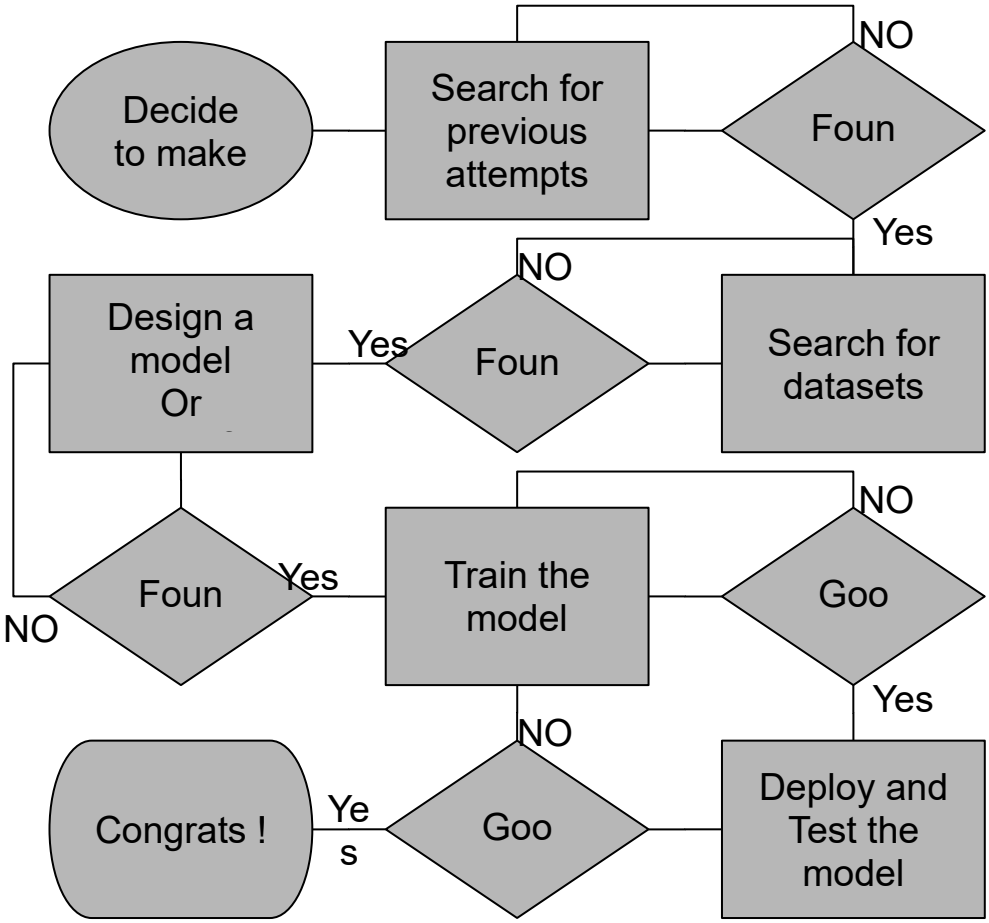
The core of the YOLO target detection algorithm lies in the model's small size and fast calculation speed. YOLO only needs to put the picture into the network to get the final detection result. The original YOLO architecture consists of 24 convolution layers, followed by two fully connected layers.

Requirements

- **Software**
  - Python programming language
  - Python CV
  - Python numpy
  - Python Ultralytics YOLOv8

- **Hardware**
  - Raspberry PI
  - Camera.

The plan took many steps as shown below



Traffic Sign dataset

There are many dataset used in traffic sign recognition models such as :

- German Traffic Sign Recognition Benchmark (GTSRB)

  This is a widely used dataset for traffic sign detection and recognition, containing more than 50,000 images of 43 different types of traffic signs. Each image is annotated with the correct traffic sign label and bounding box coordinates.

  The German TS Recognition Benchmark (GTSRB) is one of the highly known and challenging TS datasets . It is also considered the first to help evaluate significantly the problem of TS classification. GTSRB contains 50k images annotated using 43 sign categories. Following that, the German TS Detection Benchmark (GTSDB) was introduced . It is a collection of 900 images captured on German roads. Signs are labelled using only 3 categories (mandatory, danger and prohibitory).

- LISA Traffic Sign Dataset

This dataset contains over 47,000 images of traffic signs from the United States, annotated with the correct traffic sign label and bounding box coordinates. It includes a wide range of traffic signs, including regulatory, warning, and guide signs.

LISA is a TS recognition dataset that includes different videos recorded in the United States. It has 6610 frames with 7855 annotations divided into 47 sign categories.

- Belgian Traffic Sign Dataset

  This dataset contains over 10,000 images of traffic signs from Belgium, annotated with the correct traffic sign label and bounding box coordinates. It includes a wide range of traffic signs, including regulatory, warning, and guide signs.

  The Belgium TS database is quite similar to the German sets. It includes two major TS datasets for detection and recognition, with over 13k annotations of 145k images taken on Belgian roads. Signs are annotated using 63 categories for recognition and 3 categories for detection as in GTSDB.

- Chinese Traffic Sign Dataset

  This dataset contains over 10,000 images of traffic signs from China, annotated with the correct traffic sign label and bounding box coordinates. It includes a wide range of traffic signs, including regulatory, warning, and guide signs.

  Tsinghua-Tencent 100k is one of the recently published TS datasets. It contains over 100k images, collected from the Tencent Street Views of 300 Chinese cities. The dataset provides 30k TS boxes annotated using pixel masks and bounding boxes (bboxes).

- Swedish Traffic Sign Dataset

  This dataset contains over 15,000 images of traffic signs from Sweden, annotated with the correct traffic sign label and bounding box coordinates. It includes a wide range of traffic signs, including regulatory, warning, and guide signs.

  The Swedish TS detection and recognition database is a collection of 20k frames recorded over 350km of the Swedish highways and city roads. It provides 3488 TS boxes that are annotated using box coordinates, sign type, visibility status and road status. Unfortunately, these annotations only represent 20% of the data.

- The Russian TS Dataset (RTSD)

  It is another considerably large set, having 179138 labelled frames with 156 sign categories. RTSD provides 104358 TS annotations, surpassing the previously mentioned dataset.

In our model we used the German Traffic Sign Recognition Benchmark.this dataset have 43 classes of signs

Steps

1. Get the data and prepare them

We download the dataset into our working directory

```
!kaggle datasets download -d meowmeowmeowmeowmeow/gtsrb-german-traffic-sign
!unzip gtsrb-german-traffic-sign -d /content/gtsrb
```

And we make folders for the images and labels or annotations.

```
import os
import shutil
!mkdir /content/data
!mkdir /content/data/images/
!mkdir /content/data/labels/
!mkdir /content/data/images/train/
!mkdir /content/data/images/test/
!mkdir /content/data/labels/train/
!mkdir /content/data/labels/test/
```

Then move the images to their folder

```
source = "/content/gtsrb/train"
destination = "/content/data/images/train/"
for subfolder in os.listdir(source):
    subfolder_path = os.path.join(source, subfolder)
    if os.path.isdir(subfolder_path):
        for file in os.listdir(subfolder_path):
            if file.endswith(".png") :
                file_path = os.path.join(subfolder_path, file)
                shutil.move(file_path, destination)
```

```
source = "/content/gtsrb/Test"
destination = "/content/data/images/test"
for file in os.listdir(source):
  if file.endswith(".png") :
    file_path = os.path.join(source, file)
    shutil.move(file_path, destination)
print("done!")
```

Then read the csv file that contain the (Width, Height, Roi.X1, Roi.Y1, Roi.X2, Roi.Y2, ClassId, Path) of the images and strip this data to a text file with the name of the image, each file contain (ClassId, x_center, y_center, width, height

```
train_df = pd.read_csv('/content/gtsrb/Train.csv')
train_df.head()
```

And modify the path to just have the name of the image

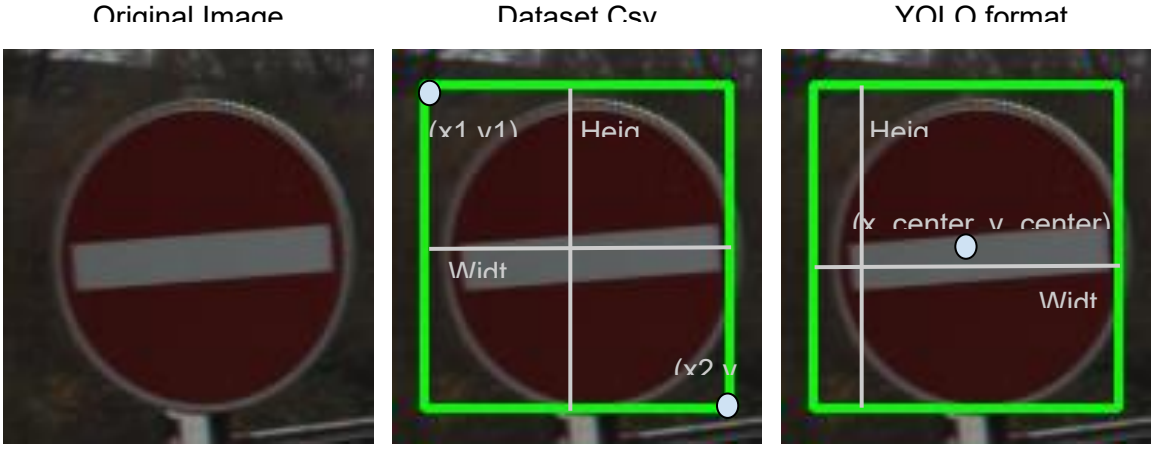
	Width	Height	Roi.X1	Roi.Y1	Roi.X2	Roi.Y2	ClassId	Path
0	27	26	5	5	22	20	20	Train/20/00020_00000_00000.png
1	28	27	5	6	23	22	20	Train/20/00020_00000_00001.png
2	29	26	6	5	24	21	20	Train/20/00020_00000_00002.png
3	28	27	5	6	23	22	20	Train/20/00020_00000_00003.png
4	28	26	5	5	23	21	20	Train/20/00020_00000_00004.png

To be this

```
df = pd.read_csv("/content/train.csv")
df.head()
```

	Width	Height	Roi.X1	Roi.Y1	Roi.X2	Roi.Y2	ClassId	Path
0	27	26	5	5	22	20	20	00020_00000_00000.png
1	28	27	5	6	23	22	20	00020_00000_00001.png
2	29	26	6	5	24	21	20	00020_00000_00002.png
3	28	27	5	6	23	22	20	00020_00000_00003.png
4	28	26	5	5	23	21	20	00020_00000_00004.png

Now we need to change the format of the annotation to YOLO format. The dataset has csv files for train and test data having (Width, Height, x_min, y_min, x_max, y_max, ClassId, path of the image) .



YOLO format needs just text file have the same name as the image having (ClassId, x_center, y_center, width, height) of all the object inside that image

```python
df = pd.read_csv("/content/train.csv")
def convert_to_yolo(x1, y1, x2, y2, w, h):
  x_center = (x1 + x2) / 2
  y_center = (y1 + y2) / 2
  x_center /= w
  y_center /= h
  width = (x2 - x1) / w
  height = (y2 - y1) / h
  return [x_center, y_center, width, height]
for index, row in df.iterrows():
  width = row["Width"]
  height = row["Height"]
  x1 = row["Roi.X1"]
  y1 = row["Roi.Y1"]
  x2 = row["Roi.X2"]
  y2 = row["Roi.Y2"]
  class_id = row["ClassId"]
  path = row["Path"]
  yolo_values = convert_to_yolo(x1, y1, x2, y2, width, height)
  txt_file_name = "/content/data/labels/train/"+path.replace(".png", ".txt")
  open(txt_file_name,"w").close()
  with open(txt_file_name,"w") as f:
     f.write(f"{class_id} {' '.join(map(str,yolo_values))}\n")
```

We get txt file have the name of the image.Each line in the text file represents an annotation for an object in the image

```
1   17 0.5 0.5 0.816667 0.8267716
```

2. Different models of YOLO

1. YOLOv1:The original YOLO based on the Darknet framework consisted of two sub-variants. The first architecture consisted of 24 convolutional layers with the final layer providing a connection into the first of the two fully connected layers.

2. YOLOv2:was introduced by Joseph Redmon in 2016. The motive was to remove or at least mitigate the inefficiencies observed with the original YOLO while maintaining the impressive speed factor.

3. YOLOv3: The third iteration of the YOLO model family originally by Joseph Redmon, known for its efficient real-time object detection capabilities.

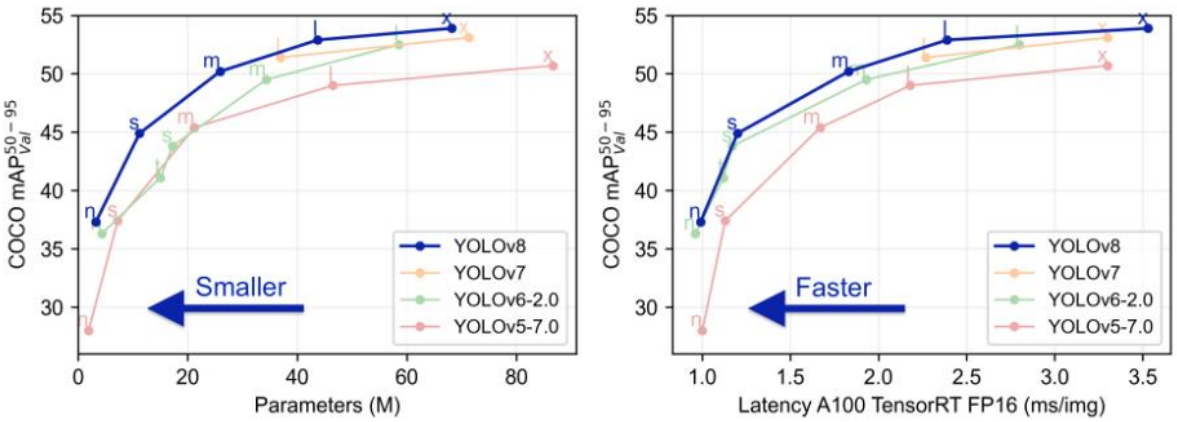
4. YOLOv4: A darknet-native update to YOLOv3 released by Alexey Bochkovskiy in 2020.

5. YOLOv5: An improved version of the YOLO architecture by Ultralytics, offering better performance and speed tradeoffs compared to previous versions.

6. YOLOv6: Released by Meituan in 2022 and is in use in many of the company's autonomous delivery robots.

7. YOLOv7: Updated YOLO models released in 2022 by the authors of YOLOv4.

8. YOLOv8: The latest version of the YOLO family, featuring enhanced capabilities such as instance segmentation, pose/keypoints estimation, and classification.



And what version of it :

Model	Image Size	mAP val (50 - 95)	Speed on CPU (ms)	Speed on GPU (ms)	Params (M)	FLOPs
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Based on MS COCO dataset

When we used GTSRB dataset with different models we got these results

Model	Image Size	mAP val	Speed on CPU (ms)	Speed on GPU (ms)	Params (M)	# epochs	loss
YOLOv8n *	640	96	180.4	2.0	3.01	15	0.357
YOLOv8n *	640	96.6	150	2.0	3.01	20	0.3189
YOLOv8s *	640	96.9	549.6	8.6	11.2	15	0.3144

Based on our results using GTSRB (* using google colab)

Now we decide that the best model is YOLOv8n.

YOLOv8n Internal Structure

	from	n	params	module	arguments
0	-1	1	464	ultralytics.nn.modules.conv.Conv	[3, 16, 3, 2]
1	-1	1	4672	ultralytics.nn.modules.conv.Conv	[16, 32, 3, 2]
2	-1	1	7360	ultralytics.nn.modules.block.C2f	[32, 32, 1, True]
3	-1	1	18560	ultralytics.nn.modules.conv.Conv	[32, 64, 3, 2]
4	-1	2	49664	ultralytics.nn.modules.block.C2f	[64, 64, 2, True]
5	-1	1	73984	ultralytics.nn.modules.conv.Conv	[64, 128, 3, 2]
6	-1	2	197632	ultralytics.nn.modules.block.C2f	[128, 128, 2, True]
7	-1	1	295424	ultralytics.nn.modules.conv.Conv	[128, 256, 3, 2]
8	-1	1	460288	ultralytics.nn.modules.block.C2f	[256, 256, 1, True]
9	-1	1	164608	ultralytics.nn.modules.block.SPPF	[256, 256, 5]
10	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'nearest']
11	[-1, 6]	1	0	ultralytics.nn.modules.conv.Concat	[1]
12	-1	1	148224	ultralytics.nn.modules.block.C2f	[384,128,1]
13	-1	1	0	torch.nn.modules.upsampling.Upsample	[None, 2, 'Nearest']
14	[-1, 4]	1	0	ultralytics.nn.modules.conv.Concat	[1]
15	-1	1	37248	ultralytics.nn.modules.block.C2f	[192, 64, 1]
16	-1	1	36992	ultralytics.nn.modules.conv.Conv	[64, 64, 3, 2]
17	[-1, 12]	1	0	ultralytics.nn.modules.conv.Concat	[1]
18	-1	1	123648	ultralytics.nn.modules.block.C2f	[192, 128, 1]
19	-1	1	147712	ultralytics.nn.modules.conv.Conv	[128, 128, 3, 2]

20	[-1, 9]	1	0	ultralytics.nn.modules.conv.Concat	[1]
21	-1	1	493056	ultralytics.nn.modules.block.C2f	[384, 256, 1]
22	[15, 18, 21]	1	759697	ultralytics.nn.modules.head.Detect	[43, [64, 128, 256]]

3. Results While training

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size	
20/20	2.6G	0.3189	0.4482	0.9913	30	640	2451
Validate while training	class	Images	Instances	Box(p	R	mAP50	mAP (50-95)
	all	39209	39209	0.995	0.993	0.994	0.996

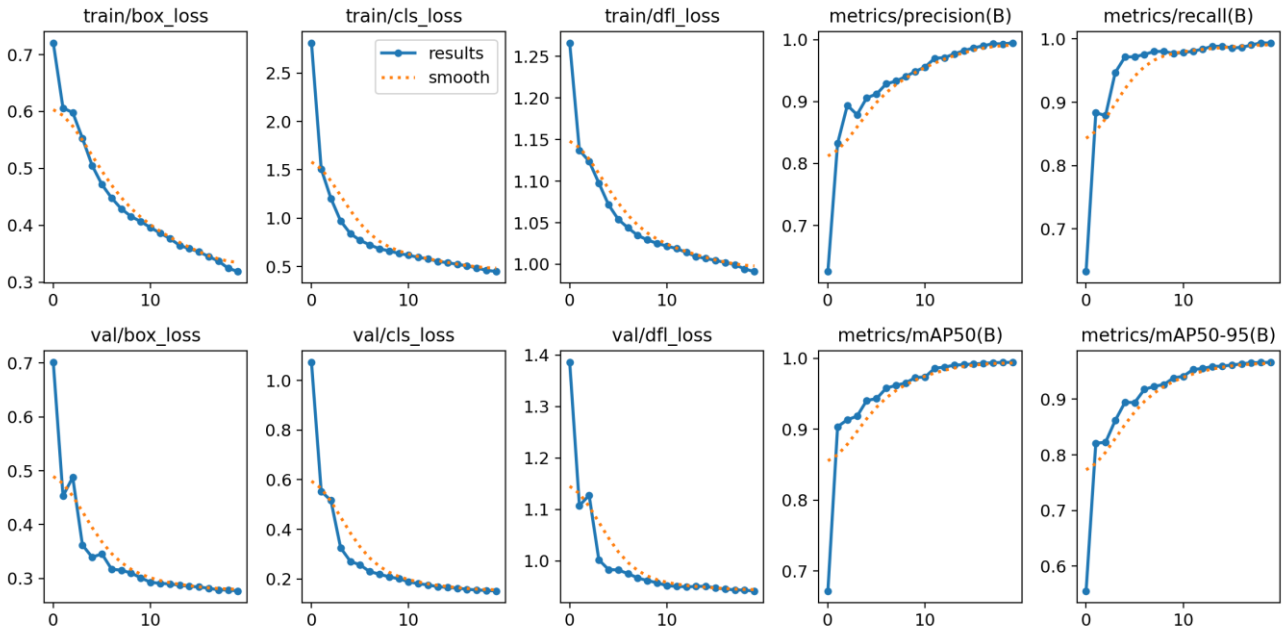
4. Test after training

Validating /content/drive/MyDrive/colab_files/different_models/runs/detect/train7/weights/best.pt...
Ultralytics YOLOv8.0.126 🚀 Python-3.10.12 torch-2.0.1+cu118 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 3014033 parameters, 0 gradients

Class	Images	Instances	Box(P	R	mAP50	mAP(50-95)
all	39209	39209	0.995	0.993	0.994	0.966
Speed limit (20km/h)	39209	210	1	0.998	0.995	0.967
Speed limit (30km/h)	39209	2220	0.999	1	0.995	0.972
Speed limit (50km/h)	39209	2250	1	0.999	0.995	0.964
Speed limit (60km/h)	39209	1410	0.999	0.999	0.995	0.972
Speed limit (70km/h)	39209	1980	0.999	1	0.995	0.973
Speed limit (80km/h)	39209	1860	1	0.999	0.995	0.964
End of speed limit (80km/h)	39209	420	0.999	1	0.995	0.959
Speed limit (100km/h)	39209	1440	0.998	1	0.995	0.967
Speed limit (120km/h)	39209	1410	1	0.997	0.995	0.967
No passing	39209	1470	1	1	0.995	0.969
No passing vehicle over 3.5 tons	39209	2010	1	1	0.995	0.972
Right-of-way at intersection	39209	1320	1	1	0.995	0.976
Priority road	39209	2100	0.999	1	0.995	0.976
Yield	39209	2160	1	1	0.995	0.977
Stop	39209	780	0.999	1	0.995	0.975

No vehicle	39209	630	0.999	1	0.995	0.959
Vehicle > 3.5 tons prohibited	39209	420	0.999	1	0.995	0.976
No entry	39209	1110	1	0.999	0.995	0.958
General caution	39209	1200	1	1	0.995	0.978
Dangerous curve left	39209	210	0.965	0.925	0.987	0.962
Dangerous curve right	39209	360	0.977	0.978	0.993	0.945
Double curve	39209	330	1	0.998	0.995	0.981
Bumpy road	39209	390	0.99	1	0.995	0.974
Slippery road	39209	510	0.998	1	0.995	0.979
Road narrow on the right	39209	270	0.998	1	0.995	0.97
Road work	39209	1500	1	0.999	0.995	0.98
Traffic Signal	39209	600	0.999	1	0.995	0.965
Pedestrian	39209	240	0.998	1	0.995	0.967
Children crossing	39209	540	0.999	1	0.995	0.975
Bicycle crossing	39209	270	0.994	1	0.995	0.965
Beware of ice/snow	39209	450	0.999	1	0.955	0.979
Wild animal crossing	39209	780	1	1	0.995	0.971
End speed + passing limit	39209	240	0.999	1	0.995	0.952
Turn right ahead	39209	689	0.966	0.953	0.991	0.95
Turn left ahead	39209	420	0.936	0.912	0.981	0.934
Ahead only	39209	1200	0.997	1	0.995	0.963
Go straight or right	39209	390	0.992	0.992	0.995	0.96
Go straight or left	39209	210	0.997	0.981	0.995	0.96
Keep right	39209	2070	0.995	0.999	0.995	0.972
Keep left	39209	300	0.979	0.983	0.994	0.956
Roundabout mandatory	39209	360	0.999	1	0.995	0.968
End of no passing	39209	240	0.999	1	0.995	0.94
End of no passing vehicle > 3.5 ton	39209	240	0.999	1	1	0.958

Speed: 0.2ms preprocess, 2.0ms inference, 0.0ms loss, 1.1ms postprocess per image



5. Test the model

```python
import cv2
from ultralytics import YOLO
from timeit import timeit

model = YOLO('runs/detect/train7/weights/best.pt')
cap = cv2.VideoCapture("overview.gif")
while cap.isOpened():
    success, frame = cap.read()
    frame = cv2.flip(frame,1)
    if success:
        results = model(frame)
        if cv2.waitKey(1) & 0xFF == ord("q"):
            break
    else:
        break
cap.release()
cv2.destroyAllWindows()
```

Time for processing a single image	
Using colab GPU	Using CPU
12.606207296472784 (ms)	342.7004797806717 (ms)

Test random image

```python
from ultralytics import YOLO
from random import choice
import os
import shutil
import cv2
imgs=[]
for (dirname, dirs, files) in os.walk("gtsrb/test/"):
    for filename in files:
        if ( filename.endswith(".png") or filename.endswith(".jpg")):
            imgs.append(filename)
        else:continue
file_jpg = choice(imgs)
print(file_jpg)
model = YOLO("weights/best7.onnx",task="detect")
results = model.predict(source=("gtsrb/test/"+file_jpg), save=True)
print(results[0].boxes.conf)
renamed = file_jpg[:-4]
renamed = renamed + "original.png"
shutil.copy(os.path.join("gtsrb/test/",file_jpg)
            ,os.path.join(results[0].save_dir,renamed))
cv2.imshow("original",cv2.imread(results[0].save_dir+'/'+renamed))
cv2.imshow("YOLOv8 Inference",cv2.imread(results[0].save_dir+'/'+file_jpg))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

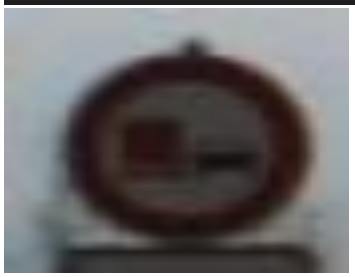
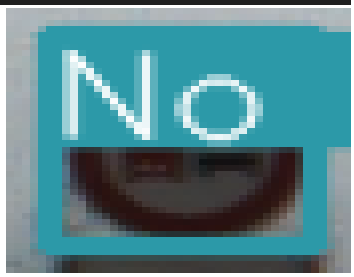


image 1/1 : gtsrb\test\04092.png: 640x640 1 No passing vechile over 3.5 tons, 106.4ms



Simple use of the model to detect signs using a camera

```python
import cv2
from ultralytics import YOLO
model = YOLO('weights/best7.onnx')
cap = cv2.VideoCapture(1)
while cap.isOpened():
    success, frame = cap.read()
    if success:
        results = model(frame)
        annotated_frame = results[0].plot()
        cv2.imshow("YOLOv8 Inference", annotated_frame)
        if cv2.waitKey(1) & 0xFF == ord("q"):
            break
    else:
        break
cap.release()
cv2.destroyAllWindows()
```

Mobile Application

Introduction

Smartphones has become an essential part of our lives In the few years, making every person have at least 1 smartphone with him 90% of the time, with each device having handful of features like Wireless communication and amazing processing power, putting this in mind, we can use such devices for almost anything from security to controlling other smart devices, which is what we did exactly in this project by making use of the Bluetooth Low Energy (BLE), WIFI and processing power of our smartphones, we can introduce features like car authentication and localization using BLE and car controlling and monitoring using WIFI and Websocket servers.

Flutter

Flutter is a framework created by Google, it allows building high-performance, cross-platform applications for android, iOS, web, and desktop from a single codebase. Flutter was first released in 2017 and has gained significant popularity among developers due to its fast development cycles, expressive and flexible UI, native like performance.

Key Features of Flutter

Single Codebase: With Flutter, developers can write code once and deploy it on multiple platforms such as iOS, Android, Web, Windows and Linux. This "write once, run anywhere" approach helps save time and effort.

Dart Programming Language: Flutter uses the Dart programming language, which was also created by Google. Dart is known for its simplicity, readability, and performance. It has a modern syntax,

supports both object-oriented and functional programming paradigms, and includes features like hot reload for rapid development.

Widgets: Flutter utilizes a reactive and component-based architecture. The entire UI is composed of widgets, which are reusable and customizable building blocks. Flutter provides an extensive set of pre-designed widgets for constructing user interfaces, as well as the ability to create custom widgets.

Hot Reload: One of Flutter's standout features is its hot reload capability. Developers can make changes to the code and instantly see the results in the app without restarting or losing the app state. This significantly speeds up the development process and helps in experimenting and iterating on designs.

Native-Like Performance: Flutter applications are compiled to native code, allowing them to achieve high performance and near-native speed. Flutter uses its own rendering engine, called Skia, to draw graphics and UI elements. It also provides access to platform-specific APIs and services, giving developers full control and access to native features.

Great Community: Flutter has a vibrant and growing ecosystem with a wide range of community created packages and libraries available through the Flutter package manager, called Pub. These packages cover various functionalities such as networking, state management, databases, and more, allowing developers to leverage existing solutions and accelerate development.

Material Design and Cupertino Widgets: Flutter offers ready-to-use widgets that follow the Material Design guidelines for Android apps and Cupertino design for iOS apps. This helps in creating apps that look and feel native to each platform, providing a consistent user experience.

Why Flutter?

Flutter provides a powerful framework for building beautiful, fast, and cross-platform application with a single codebase, along with dart’s simple syntax, and the many ready to use packages and libraries available on pub.dev the Flutter package manager, including BLE and WebSocket packages which was needed for this project, it makes one of the best choices for a project like this one, as I was ready to work on the actually project fairly quickly without worrying about the application’s initial setup process.

ADAS Application

The application provides functionality for two of the main advanced driver assistance systems in this project, which is BLE and Car Control System, the applications design is following the latest Google design guidelines for colors (Material 3), along with Samsung’s One UI design for easy one-handed usage of the app, giving the user one of the best user experience possible.

Application Objectives

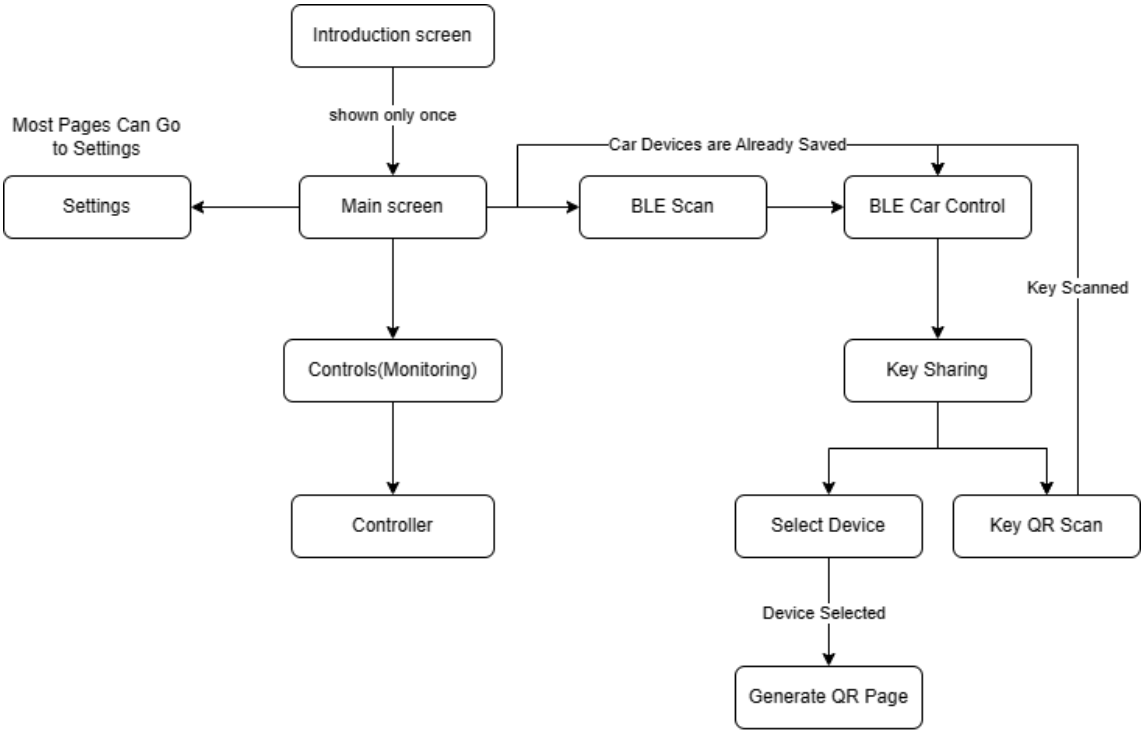
- Handling Multiple Car Devices using BLE with Optimal Security.
- BLE Localization for the Car Devices.
- Car Control Monitoring using WebSocket’s.
- Car Control using WebSocket’s Simulating the actual controller.
- Full English and Arabic Support.

- Interactive User Interface with a Positive User Experience.

Programs and Packages Used

- **Android Studio along with Flutter Framework:** The IDE and framework recommended by google to develop a cross-platform mobile flutter application.
- **GitHub desktop Application:** Version control & backups of my code for better project management with automatic compiling and distribution of the of the app with every commit.
- **Bloc:** State management package, allows control of the apps states, mainly to limit the number of UI updates that's not needed, which leads to better performance while providing easier access to variables and functions.
- **Hive:** lightweight and extremely fast key-value database, used for storing all cars devices related info and app settings locally on the phones secure storage partition.
- **Easy localization:** Used for the full Arabic – English language translations of the application.
- **Flex Color & Dynamic color:** Allows for material 3 design colors in the application either statically choosing them, or with dynamic systems colors for newer android version.
- **Flutter Reactive BLE:** Flutter library that allows for multiple devices BLE operations which allows all BLE functionalities used in the application.
- **Encrypt:** Encryption package that allows for AES encryption which is used for BLE authentication and random keys generation for each device.
- **Websocket universal:** Dart and Flutter Websocket client package, allows all the controls system functionalities used in the application.

Application Flow Diagram

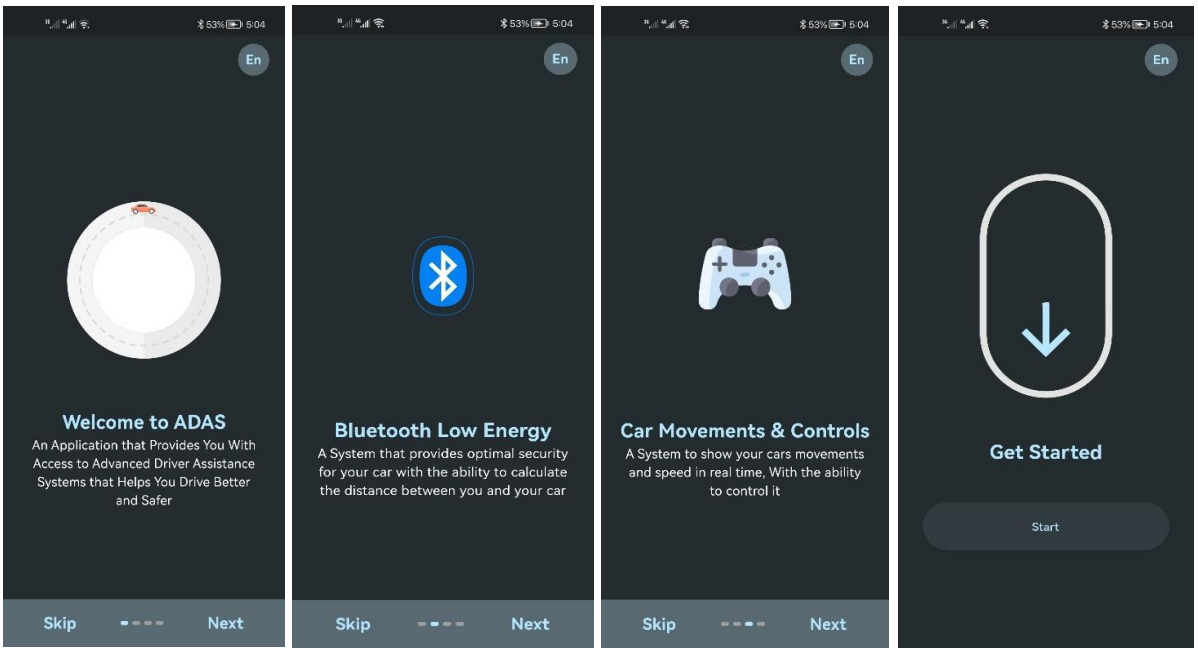


Screens and Functionalities

The app has a lot of different screens that can be used, with each having its own functionality.

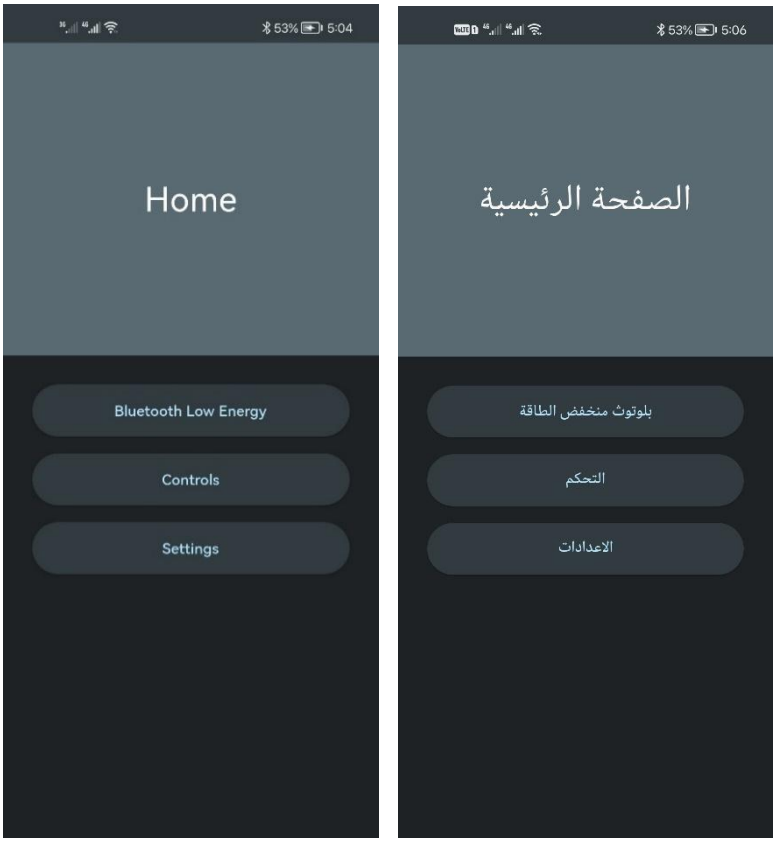
Intro screen

Used to introduce user to app functionality with animated images.



Main screen

Used to route navigate the user to the systems that he would like to use or the settings page.



BLE System

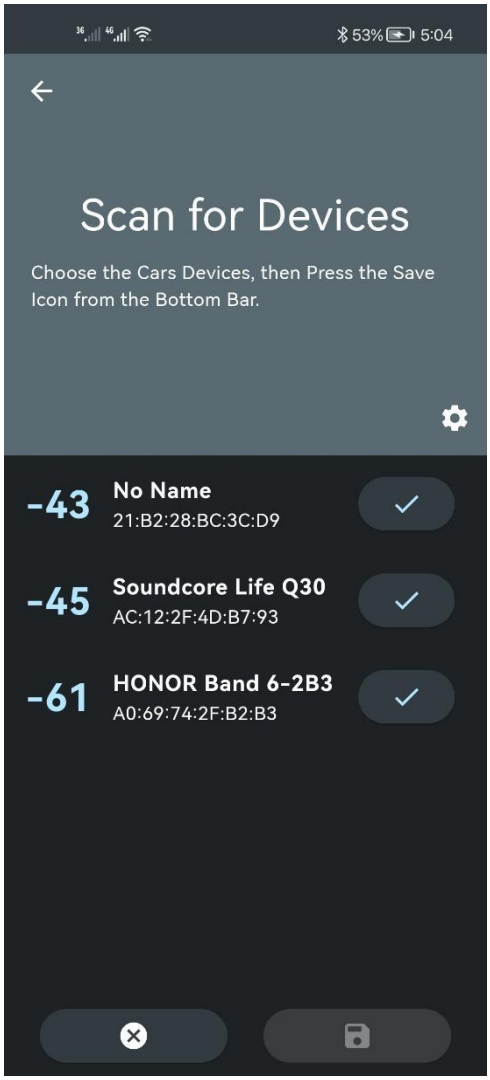
This is where the real functionality of the app starts, an advanced BLE system used securely authenticate with selected car(s), also uses an algorithm for localization that calculates the distance between the user and the car using the value of the RSSI of the BLE device.

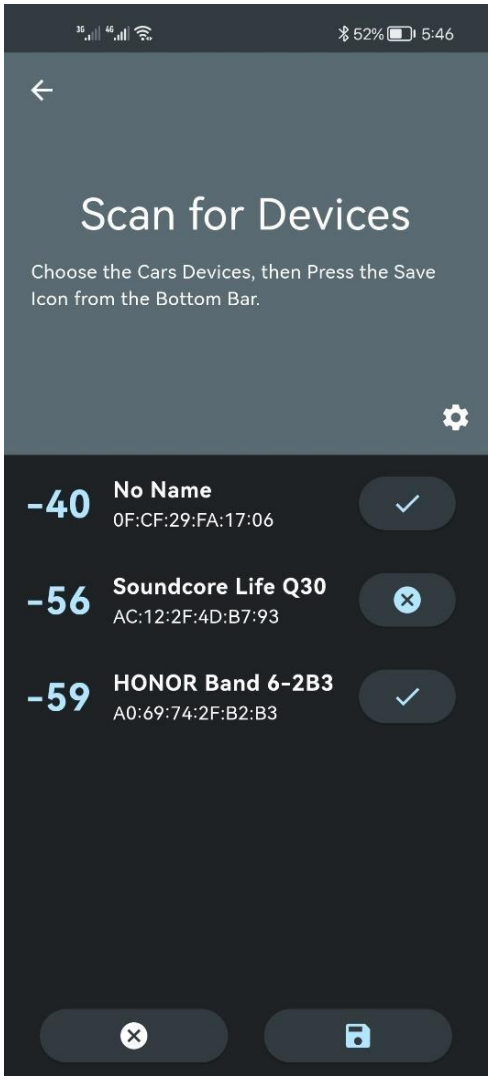
A car device is an esp32 with special code.

BLE Scan Screen

The user is first met with the scan page, where the user can start scanning for all BLE devices to search for the car(s), to allow for such scanning the user is first prompted to enable Bluetooth and the sent to location settings to enable it, which is done automatically, after that all the BLE devices will start showing up with their current RSSI Values, Names and IDs with the option to select the said device, after the user select his BLE devices (Cars) the save button will be enabled which will allow save the list of devices he just selected and create a secure encryption key for each device, and save it all on the devices hidden local storage. Then the user is then redirected to the next page.

The user will automatically be sent to the next page every time he uses the BLE system while having devices saved, the user can reset his devices from the settings page.

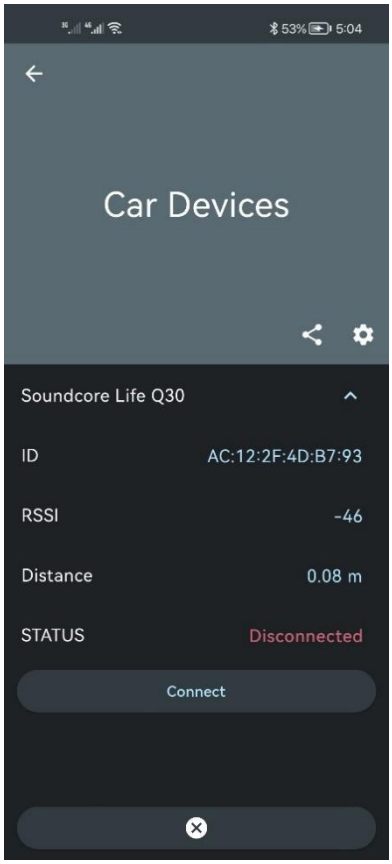


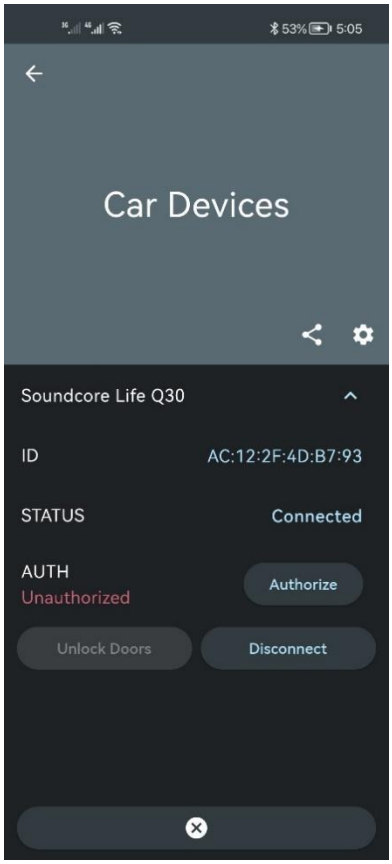


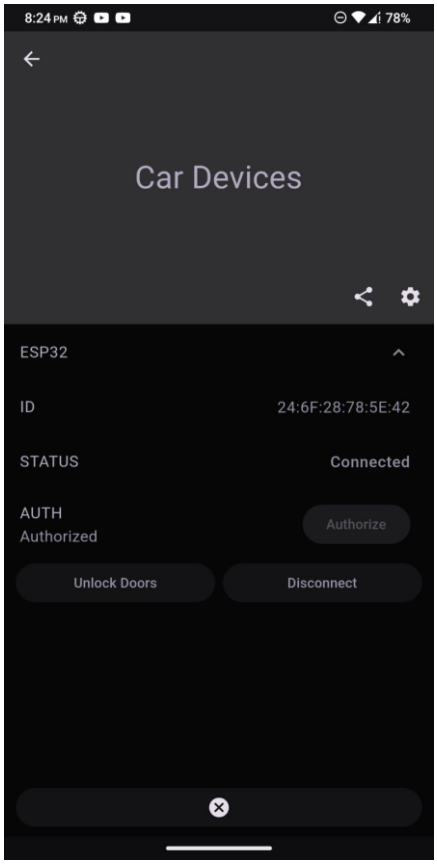
BLE Car Devices Screen

This is the page that the user will see from now on, and where the BLE controls are, the user will see the device(s) that he selected shown with it's current information and state.

The user will need to connect to the device first to be able to start the BLE authorization, with the device connected to the car device (esp32), and authorize button pressed, the device will start communicating with the esp 32 using BLE. At first the app will check the authentication state from the esp32 which is inside service with UUID "d9327ccb-992b-4d78-98ce-2297ed2c09d6" as charactristic1 with UUID "e95e7f63-f041-469f-90db-04d2e3e7619b", the will get a string with the value "unauthorized" which then will start the authorization process, if this is the first time these devices communicate, the app will need to share the keys with the esp32 which is done by using a physical button connected to the esp32. With the button pressed the app will share the special encryption key and vector that it generated when the device is created. If the device was already connected to the esp32 this step is not necessary.







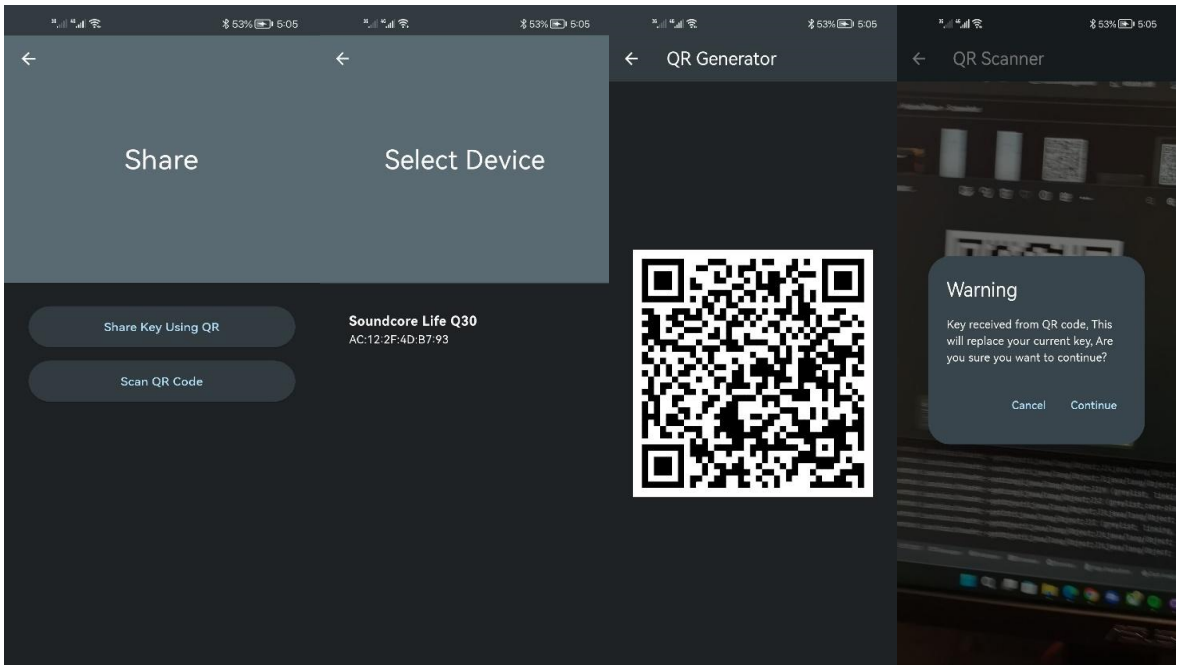
Now with both the app and the esp having the same encryption key they will start performing a handshake, where the esp32 will send a random string to the app, the app will encrypt this message and resend it to the esp32, the esp32 will encrypt the same message and check if the 2 encrypted messages are the same, if this is true then the phone using the app will be authenticated to send commands to the car like unlocking the car doors.

Share Key Screen

If the user like to share the car with let's say his son, it can be simply done using the sharing page, in the share page the user has two options which is to share a QR code or scan it.

If the user uses the share page, he will be redirected to a page with all his saved devices where if he pressed on a device, he will be redirected to another page that generates a special string which when shared with the other device can be transformed into the same key.

In the scan page, the app will open the device camera to scan the QR code, once a QR code is recognized, it will prompt the user with an alert dialog that says this will replace his key for the car if it's found. If the car is not found the key will be stored on the device and once the device is added, that key will be used.



Control System

The second system in the app is the control system, where it's connected to the car directly using a Websocket server, listening on all the broadcasted messages, and map each bit to its assigned widget on the screen, showing real time movements of the car in the page, at first the user has to input the correct Ip addressee and port for the Websocket server, and press connect, which will automatically save them for the next time, and connect to the server.

The message coming from the Websocket server consist of 7bits, which means in the following order: 1st bit represents the movements and breaks, 0 means the car can move freely, 1 means it activates the breaks stopping the car.

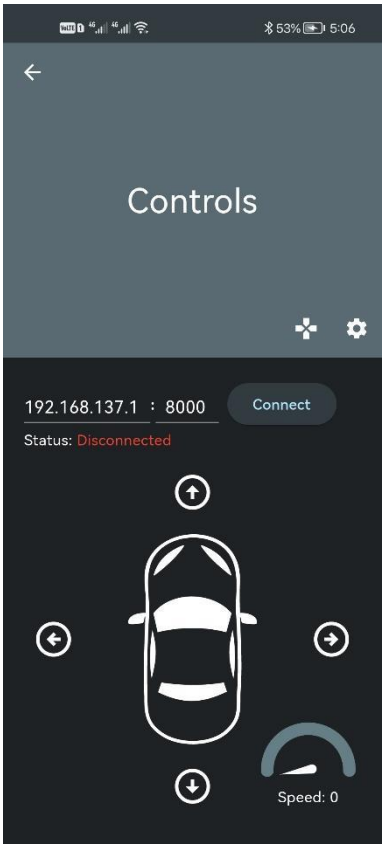
2nd and 3rd bits are for forward movement and backwards movement, having a value from 0 to 5 according to its speed.

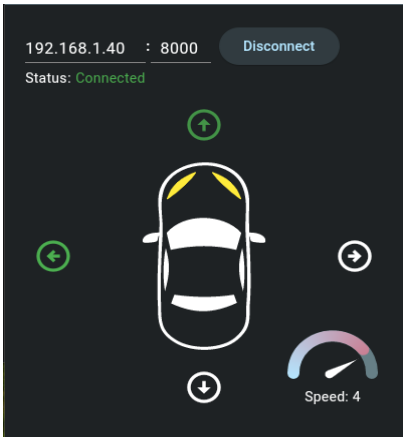
4th and 5th are for right and left movement accordingly, also having a value from 0 to 5

6th and 7th are for low and high lights having a value of 0 for off and 1 for on.

Control Screen

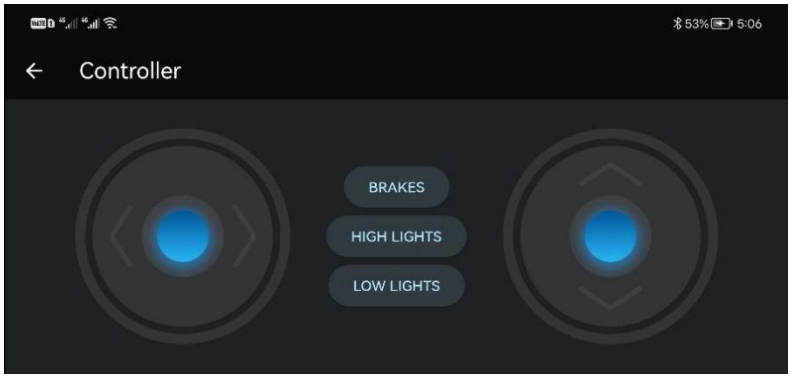
Once the app is connected the Websocket server, it will show ever move made by the controller mapped on the car mockup along with a speedometer that shows the cars speed.





Controller Screen

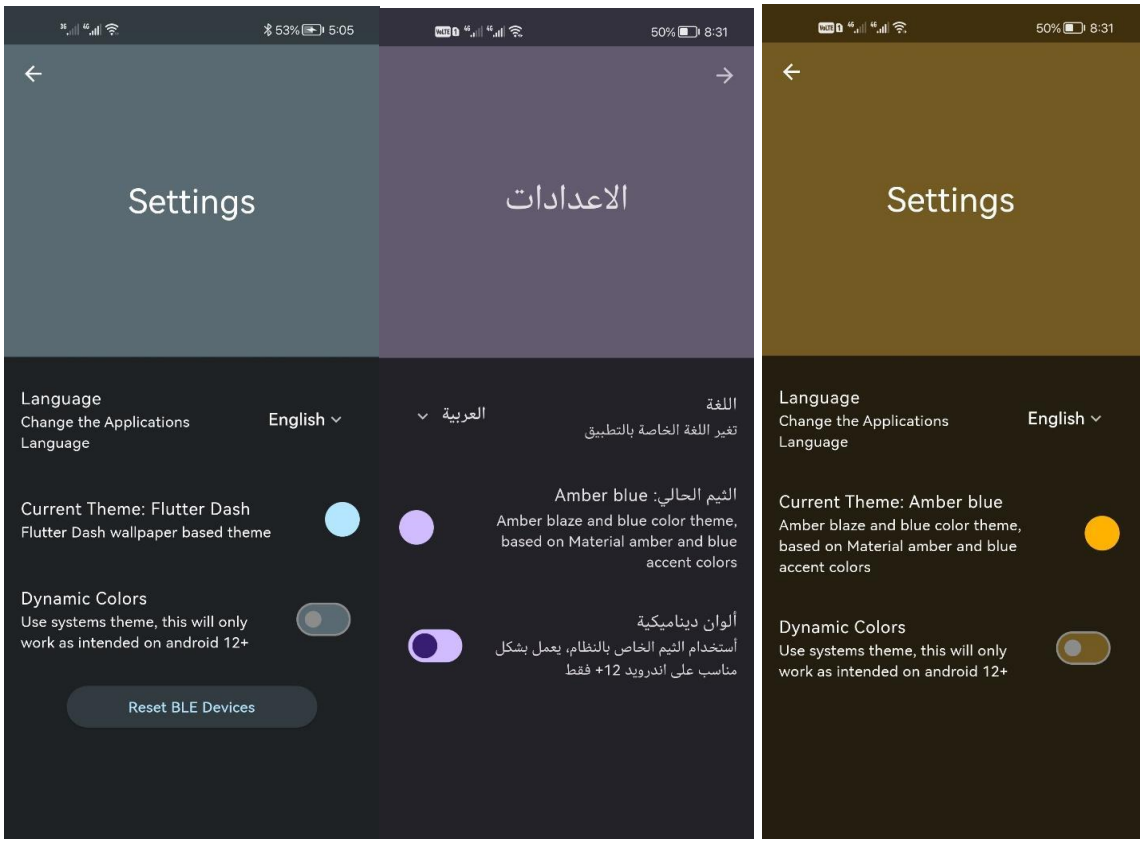
Another phone can be used as the controller by connecting to the WebSocket server at least once to save the IP and Port from the controls screen. Then pressing the gamepad icon, which will automatically connect to the server and simulate the real controller's messages sending them to the server.



Settings Screen

Here the user has the ability to change app language to either Arabic or English, also the user has the ability to change the app theme from more than 40 themes, and if he has a device with android 12 or higher, he can use the new dynamic system colors so that the app follows the systems colors.

The option to reset saved BLE devices is also available.



Conclusion

In conclusion, the implementation of the BLE system and control system in this Flutter app enhances the user's car experience. The BLE system provides secure authentication and localization using RSSI values, allowing users to connect and control their cars easily. The app's features include scanning for BLE devices, saving selected devices and creating an encryption key for each device, and performing a secure handshake for authorization. Sharing car access is simplified through QR codes. The control system utilizes a WebSocket server to display real-time car movements and enables remote control. Overall, this app brings convenience, security, and enhanced functionality to car owners.

BLE Authentication Key

Introduction

In recent years, there has been a significant rise in the use of Bluetooth Low Energy (BLE) technology due to its low power consumption and compatibility with various devices. BLE has found applications in diverse fields, including home automation, healthcare, and automotive industries. One prominent area of interest is the development of secure and convenient solutions for car access and key management.

The conventional car key systems, such as physical keys and remote keyless entry (RKE) fobs, have several limitations. They can be easily lost, stolen, or duplicated, compromising the security of the vehicle. Additionally, carrying multiple keys for different vehicles can be cumbersome for car owners. To address these challenges and enhance the security of car access systems, there is a growing interest in developing digital solutions that utilize BLE technology and encryption algorithms.

The motivation behind this graduation project module is to design and implement a BLE car key system using BLE5 in ESP32 microcontroller and a mobile app. By leveraging the capabilities of BLE5, the system aims to provide a secure and efficient means of accessing and controlling a car. The

integration of an AES algorithm for encryption and the utilization of SPIFFS file system for key and vector storage further enhance the security and reliability of the system.

The module seeks to explore the potential of BLE5 and AES encryption in the automotive domain, contributing to the advancement of secure and convenient car access solutions. By developing a functional prototype, this module aims to demonstrate the feasibility and effectiveness of the proposed system while addressing the limitations of traditional car key systems.

Overall, the module's objective is to provide a robust and secure BLE-based car key system that offers convenience, enhanced security, and ease of use for car owners, ultimately revolutionizing the car access experience.

Literature Review

1. Bluetooth Low Energy (BLE) Technology

1.1 Distinction between Classic Bluetooth and Bluetooth Low Energy

Bluetooth Low Energy (BLE) technology, also known as Bluetooth Smart, is a wireless communication protocol designed for short-range communication between devices. It is a power-efficient version of the classic Bluetooth technology and is commonly used in applications that require low power consumption, such as wearable devices, health and fitness trackers, home automation systems, and IoT (Internet of Things) devices.

Classic Bluetooth and Bluetooth Low Energy (BLE) are two different versions of the Bluetooth wireless communication technology. Here are the key distinctions between them:

- **Power Consumption:** One of the main differences between Classic Bluetooth and BLE is power consumption. Classic Bluetooth is designed for continuous data streaming and can consume more power, making it suitable for devices with larger batteries. On the other hand, BLE is optimized for low power consumption, allowing devices to operate on small batteries for extended periods.

- **Data Transfer Rate:** Classic Bluetooth offers higher data transfer rates, typically up to 3 Mbps (megabits per second), which is suitable for applications like audio streaming and file transfers. BLE, on the other hand, has a lower data transfer rate, typically up to 1 Mbps, which is sufficient for applications with smaller data payloads like sensor data or simple control commands.

- **Range:** Classic Bluetooth generally has a longer range compared to BLE. Classic Bluetooth can typically reach up to 100 meters, depending on the implementation and environmental factors. BLE, on the other hand, has a shorter range, typically up to 100 meters, but it can be lower in practice due to its power optimization techniques.

- **Use Cases:** Classic Bluetooth is commonly used for applications that require continuous data streaming, such as audio streaming between a smartphone and wireless headphones or transferring files between devices. BLE is well-suited for applications that prioritize power efficiency and intermittent data transfer, such as wearable devices, health and fitness trackers, smart home devices, and IoT applications.

- **Complexity:** Classic Bluetooth has a more complex protocol stack compared to BLE. BLE was designed with simplicity in mind, focusing on core functionalities and minimizing overhead to achieve low power consumption. This simplicity allows for easier implementation and lower development costs for BLE-enabled devices.

1.2 Why using BLE not Wi-Fi for communication

There are several reasons why Bluetooth Low Energy (BLE) technology is often preferred over Wi-Fi for certain applications, such as a car key system. Here are some reasons for choosing BLE over Wi-Fi:

- **Power Consumption:** BLE is specifically designed for low-power applications, making it more energy-efficient compared to Wi-Fi. This is crucial for battery-powered devices like car keys, as it allows for longer battery life and reduces the need for frequent recharging or battery replacement.

- **Range:** BLE has a shorter range compared to Wi-Fi, which can be an advantage in certain scenarios. For a car key system, a shorter range helps to ensure that the key is only within proximity of the vehicle, enhancing security by reducing the risk of unauthorized access from a distance.

- **Compatibility:** BLE is widely supported across various devices, including smartphones, tablets, and wearable devices. This compatibility allows for seamless integration with mobile apps, making it easier for users to interact with the car key system using their smartphones.

- **Quick Connection Establishment:** BLE has a faster connection establishment time compared to Wi-Fi. This quick connection setup is beneficial for time-sensitive applications like unlocking a car door, where users expect near-instantaneous response.

- **Security:** BLE incorporates several security features, such as encryption and authentication, to ensure secure communication between devices. While Wi-Fi also offers security protocols, BLE's lower power and shorter range characteristics can be advantageous for preventing unauthorized access and reducing the risk of attacks.

- **Cost:** BLE technology is generally more cost-effective compared to Wi-Fi, making it a viable choice for applications where cost efficiency is a consideration. This can be especially important for mass production of car key systems, where cost optimization is crucial.

1.3 BLE Communication Modes

BLE (Bluetooth Low Energy) communication modes refer to different types of connections and data transfer methods supported by the Bluetooth Low Energy technology. These different communication modes in BLE provide flexibility and versatility for various types of applications, ranging from simple data transmission to more complex interactions between devices. Here are the common BLE communication modes:

- **Advertising mode:** In this mode, a BLE device broadcasts packets of data, known as advertising packets, at regular intervals. These packets contain information about the device,

such as its name and available services. Advertising mode is primarily used for device discovery and initial connection establishment.

- **Connection mode:** Once a BLE device receives an advertising packet from another device, it can establish a connection by sending a connection request. This mode allows for bidirectional communication between two connected devices, known as the central device (e.g., smartphone) and the peripheral device (e.g., sensor). The connection mode enables data exchange using various protocols, such as the Generic Attribute Profile (GATT).

- **Central mode:** In this mode, a BLE device acts as the central device that initiates connections with peripheral devices. The central device controls the connection and can request data from peripheral devices or send commands to them.

- **Peripheral mode:** In contrast to the central mode, a BLE device operates in peripheral mode when it advertises itself and waits for incoming connection requests from central devices. Peripheral devices typically provide data or services that can be accessed by central devices.

- **Broadcaster mode:** This mode is a variant of advertising mode where the BLE device continuously broadcasts data packets without establishing connections. It is commonly used for simple one-way data transmission, such as transmitting sensor data to multiple receiving devices simultaneously.

- **Observer mode:** Similar to broadcaster mode, observer mode is a variant of connection mode where a device scans for advertising packets from other devices without establishing connections. It allows a device to passively monitor the advertising data from nearby devices.

1.4 BLE Communication Operations

BLE communication operations refer to the actions and procedures involved in establishing and maintaining communication between BLE devices. These communication operations form the basis of establishing and maintaining communication between BLE devices. They allow for efficient data exchange and interaction between central and peripheral devices in various applications, such as IoT devices, health and fitness trackers, and smart home systems.

- **Device discovery:** BLE devices use advertising packets to broadcast their presence and available services. In this operation, devices scan for advertising packets to discover nearby BLE devices. The scanning device listens for these packets and collects information such as device name, services, and signal strength.

- **Connection establishment:** Once a device discovers a desired BLE device through advertising, it can initiate a connection. The device sends a connection request to the target device, which can either accept or reject the request. If accepted, a connection is established between the two devices.

- **Service discovery:** After establishing a connection, the central device can perform service discovery to determine the available services provided by the peripheral device. This involves

sending queries to the peripheral device to obtain a list of supported services, characteristics, and their corresponding UUIDs (Universally Unique Identifiers).

- **Characteristic read/write:** Once the services and characteristics are discovered, the central device can read or write data to specific characteristics of the peripheral device. This allows for bidirectional data exchange between the devices. The central device can send read requests to retrieve data from a characteristic or write requests to update data in a characteristic.

- **Notification and indication:** BLE supports notification and indication mechanisms to enable asynchronous data transfer from the peripheral device to the central device. The peripheral device can send notifications or indications to the central device when the value of a specific characteristic changes. The central device can register for these notifications or indications and receive data updates without actively polling for changes.

- **Connection termination:** When the communication session between devices is complete or no longer required, the connection can be terminated. Either device can initiate the disconnection process by sending a disconnection request.

2. AES Algorithm for Encryption

When it comes to choosing a secure encryption algorithm for exchanging data between an ESP32 and an Android app over Bluetooth Low Energy (BLE), there are several options available. One commonly used and well-regarded encryption algorithm is the Advanced Encryption Standard (AES). AES is widely adopted and considered secure for most practical applications for offering several advantages:

- **Security:** AES is a widely recognized and trusted encryption algorithm that provides a high level of security. It has been extensively studied, tested, and evaluated by experts in the field of cryptography. AES has a proven track record of protecting sensitive information in various applications.

- **Symmetric Encryption:** AES is a symmetric encryption algorithm, meaning the same key is used for both encryption and decryption. This makes it suitable for scenarios where the ESP32 and the Android app need to securely exchange information. By using AES, you can ensure that the authentication key is securely communicated between the devices.

- **Efficiency:** AES is designed to be efficient in terms of both computational resources and memory usage. It offers a good balance between security and performance. AES encryption and decryption operations can be performed quickly and efficiently on microcontrollers like the ESP32, making it suitable for resource-constrained devices.

- **Standardization and Interoperability:** AES is an industry-standard encryption algorithm adopted by organizations and governments worldwide. It is widely supported across different platforms, programming languages, and devices. By using AES, you ensure interoperability between the ESP32 and the Android app, as both can use the same encryption and decryption algorithm.

- **Flexibility:** AES supports different key sizes, including 128-bit, 192-bit, and 256-bit. You can choose the appropriate key size based on our security requirements. AES also supports different modes of operation, such as CBC (Cipher Block Chaining) and CTR (Counter), which offer additional flexibility in encrypting and decrypting data.

- **Publicly Available Implementation:** AES is a publicly available encryption algorithm, and there are well-documented and widely used implementations available in various programming languages. This makes it easier to find libraries and resources for AES implementation, simplifying the development process for our project.

It is important to note that while AES is a secure encryption algorithm, proper implementation and key management are crucial to maintaining the security of our system. We carefully handle key storage, key exchange, and ensure appropriate authentication mechanisms to protect against potential security vulnerabilities.

3. Data and Memory Management System

3.1 SPIFFS File System for Key and Vector Storage

SPIFFS (Serial Peripheral Interface Flash File System) is a file management system designed for small embedded systems that use SPI flash memory for data storage. It is commonly used in microcontroller-based platforms such as Arduino, ESP8266, and ESP32.

SPIFFS provides a file-based interface that allows reading, writing, and managing files stored in SPI flash memory. It operates directly on the flash memory, providing a file system abstraction layer on top of it. This allows developers to treat the flash memory as a traditional file system, enabling easy storage and retrieval of data.

Some key features of SPIFFS include:

- **Lightweight:** SPIFFS is designed to be lightweight and efficient, making it suitable for resource-constrained embedded systems.

- **Wear-leveling:** SPIFFS includes wear-leveling algorithms to distribute write operations evenly across the flash memory, prolonging its lifespan.

- **File organization:** SPIFFS organizes data into files and directories, similar to a traditional file system. It supports standard file operations such as opening, closing, reading, and writing files.

- **Metadata:** SPIFFS stores file metadata, including file size, modification timestamp, and access permissions.

- **POSIX-like API:** SPIFFS provides a simple API that resembles the POSIX file system API, making it familiar to developers who have experience with standard file systems.

SPIFFS is typically used in scenarios where the available RAM is limited, and traditional file systems like FAT32 may not be suitable. It is well-suited for storing configuration data, sensor readings, log files, and other small data files in embedded systems.. It has been deprecated by the creators of the ESP8266 and ESP32 platforms.

3.2 Comparing SPIFFS To Storing Data In EEPROM

there are several benefits to using SPIFFS:

- **Larger storage capacity**: SPIFFS allows you to use SPI flash memory, which typically offers much larger storage capacity compared to EEPROM. EEPROMs commonly found in microcontrollers often have limited storage capacity, typically in the range of a few kilobytes to a few tens of kilobytes. In contrast, SPI flash memory can provide several megabytes or even gigabytes of storage space.
- **File system abstraction**: SPIFFS provides a file system abstraction layer, allowing you to organize data into files and directories. This makes it easier to manage and access different data files compared to raw EEPROM storage, where you would have to implement our own data organization scheme.
- **Ease of use**: SPIFFS provides a familiar file system interface with standard file operations such as opening, closing, reading, and writing files. This makes it easier to integrate and work with existing code that expects file system functionality.
- **Wear-leveling**: SPIFFS includes wear-leveling algorithms that distribute write operations across the flash memory, helping to extend its lifespan. EEPROM, on the other hand, has limited write endurance, meaning it can only sustain a certain number of write cycles before it may start to fail.
- **Faster write speeds**: SPIFFS generally offers faster write speeds compared to EEPROM, as writing to SPI flash memory is typically faster than writing to EEPROM. This can be beneficial for applications that require frequent or large data writes.

System Implementation

1. Description of Hardware Components Used

1.1 esp32

The ESP32 is a versatile microcontroller with dual-core processing, built-in Wi-Fi and Bluetooth, ample memory and storage, GPIO pins for device interfacing, ADC for analog signal measurement, security features, and support for various development environments. It is power-efficient and commonly used for IoT applications.

1.2 push button

When pushed make us capable to generate and store another key.

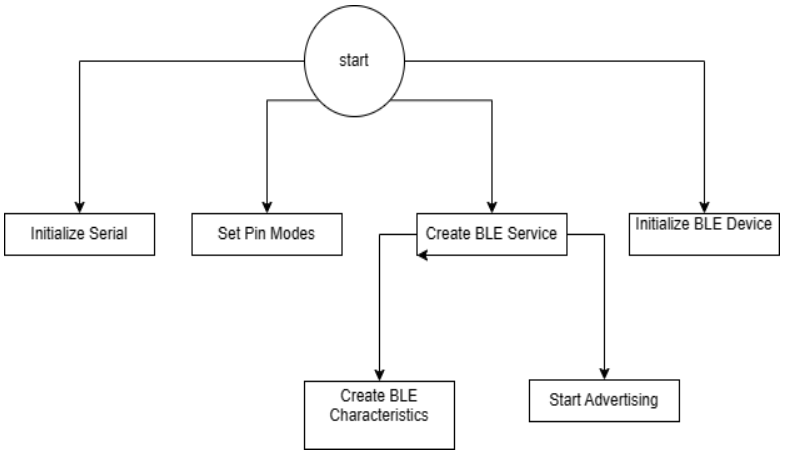
1.3 power door lock

To perform the controller action.

2. flowchart

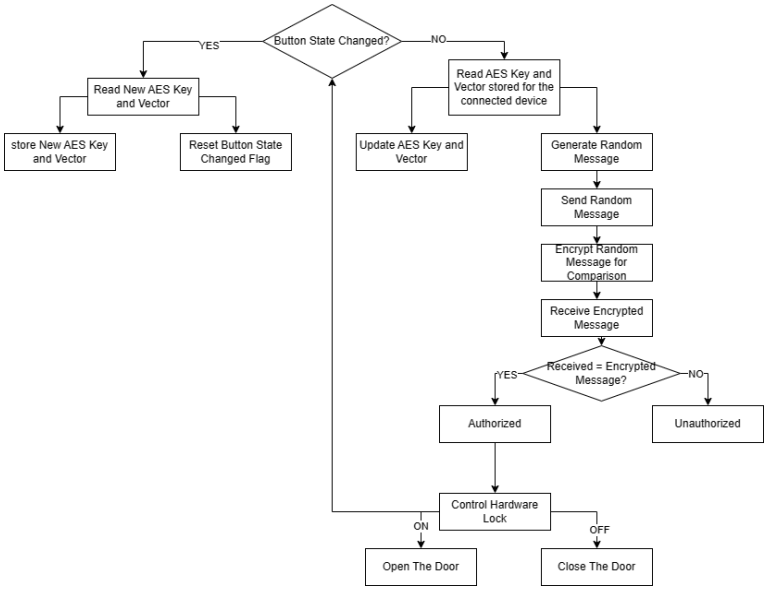
2.1 Setup

- Start the program.
- Initialize the serial communication.
- Set the pin modes.
- Initialize the BLE device.
- Create the BLE service.
- Create the BLE characteristics.
- Start advertising the BLE service.



2.2 Loop

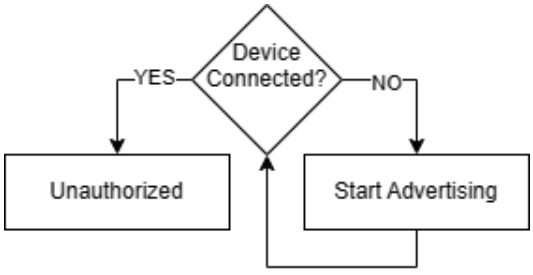
- Check if the button state has changed.
  - If yes, read the new AES key and vector.
    - Store the new AES key and vector.
    - Reset the button state changed flag.
  - If not, read the AES key and vector stored for the connected device.
    - Update the AES key and vector.
    - Generate a random message.
    - Send the random message.
    - Encrypt the random message for comparison.
    - Receive the encrypted message.
    - Check if the received message matches the encrypted message.
      - If yes, the device is authorized.
        - Control the hardware lock.
          - If the action is "ON", open the door.
          - If the action is "OFF", close the door.
      - If not, the device is unauthorized.



NOTE THAT the button state is handled with ISR. Using an Interrupt Service Routine (ISR) for button state handling offers benefits such as immediate responsiveness, reduced CPU load, precise timing, and simplified code structure. It ensures quick detection of button state changes, minimizes CPU usage, provides accurate timing information, and improves code organization.

2.3 BLE Device Connection

- Start advertising.
- Check if a device is connected.
  - If yes, the device is unauthorized.
    - Go to the Loop section.
  - If not, go back to start advertising.



NOTE THAT callbacks are used for connection and disconnection and other events in BLE as they provide an event-driven approach, simplifies event handling, enables asynchronous operations, and seamlessly integrates with BLE libraries. It ensures structured and efficient event responses, resource management, and UI updates.

Overview of Software Tools and Libraries Employed

1. IDE: PlatformIO

PlatformIO is an open-source IDE for embedded systems development. It supports various microcontrollers, provides a unified platform for firmware development, offers code editing, library management, debugging, testing, and integrates with popular code editors. It simplifies the development process and supports cross-platform development.

PlatformIO is designed to work seamlessly with Visual Studio Code (VSCode). It provides a plugin for VSCode called "PlatformIO IDE," which integrates all the functionality of PlatformIO into the VSCode environment. By installing the PlatformIO IDE extension in VSCode, you can leverage the features of PlatformIO while enjoying the benefits of a powerful and customizable code editor.

2. Libraries

2.1 **ArduinoBLE**

This library is part of the Arduino framework and provides an easy-to-use interface for BLE programming on the ESP32 using the Arduino IDE or PlatformIO. It offers a simplified API for creating BLE peripherals and interacting with BLE devices.

2.2 SPIFFS

Thebigpotatoe/Effortless-SPIFFS is a library specifically developed for ESP8266 and ESP32 microcontrollers. It aims to simplify the integration and usage of SPIFFS, the SPI Flash File System. With a user-friendly API, it provides streamlined functions for managing the file system, including file creation, reading, writing, deletion, and file system utilities. The version `^2.3.0` implies compatibility with version 2.3.0 and any subsequent compatible versions. This library offers an effortless and efficient approach to work with the file system stored in the microcontroller's flash memory, enabling easier file management in our projects.

2.3 **AESLib**

The AESLib library for ESP32 is a versatile cryptographic library that facilitates AES (Advanced Encryption Standard) encryption and decryption operations. It offers support for different key sizes, including 128, 192, and 256 bits, allowing you to choose the level of security required for our application. By integrating the AESLib library into our ESP32 projects, you gain the ability to protect sensitive data during transmission or storage. Whether you are developing secure communication systems, IoT applications, or cryptographic protocols, the AESLib library provides a reliable and straightforward solution for implementing AES encryption and decryption on the ESP32 microcontroller platform. Its intuitive API and robust functionality make it a valuable tool for ensuring data security and confidentiality in a wide range of applications.

Conclusion

In conclusion, the project aims to establish secure communication and device authentication between an ESP32 device and an Android app using Bluetooth Low Energy (BLE), while incorporating the SPIFFS (SPI Flash File System) as a file management system.

By incorporating encryption algorithms like AES (Advanced Encryption Standard) and implementing device authentication mechanisms, we can ensure the confidentiality, integrity, and security of the data exchanged between the devices.

The use of AES encryption, in combination with SPIFFS, provides a robust security framework. SPIFFS allows the ESP32 to securely store sensitive information, such as encryption keys or authentication data, in a file system that resides in the ESP32's flash memory. This prevents unauthorized access to critical data and enhances the overall security of the system.

With SPIFFS, the ESP32 can securely manage the storage and retrieval of encrypted data, configuration files, or other necessary information. This file management system ensures the availability and persistence of the data required for the encryption and authentication processes.

Furthermore, by incorporating device authentication, we establish trust between the ESP32 and the Android app. The exchange and verification of unique identifiers or secret keys allow only authenticated devices to interact with the ESP32. This adds an additional layer of security and prevents unauthorized devices from accessing or manipulating sensitive data.

Virtual Assistant

Abstract

A virtual assistant for cars is a software-based application that uses natural language processing, machine learning and deep learning algorithms to interact with passengers and drivers of vehicles. The purpose of this virtual assistant is to provide a seamless and personalized driving experience by assisting with various tasks such as

Vehicle control, vehicle maintenance, navigation, entertainment, communication, and more.

The virtual assistant can be activated through voice commands, it can understand and execute the command without any spoken acknowledgement, making it easy for drivers and passengers to interact with it while on the go.

Generally, The introduction of a virtual assistant for cars holds great promise in improving the safety and convenience of driving. By offering a hands free and tailored experience this technology enables drivers to remain attentive on the road while still remaining connected to their digital activities.

Introduction

Virtual assistants have become very common nowadays. With the widespread adoption of smartphones and smart speakers, virtual assistants like Siri, Alexa, Google Assistant, and others have become increasingly popular. These assistants use natural language processing and machine learning algorithms to understand and respond to user requests, perform tasks, and provide information. Our simple virtual assistant will contribute in controlling cars like opening the car doors and closing it, control the car lights, etc. it will activate after hearing its wakeup word and will play a chime sound confirming that the virtual assistant has been activated. At the heart of our virtual assistant are three key models.

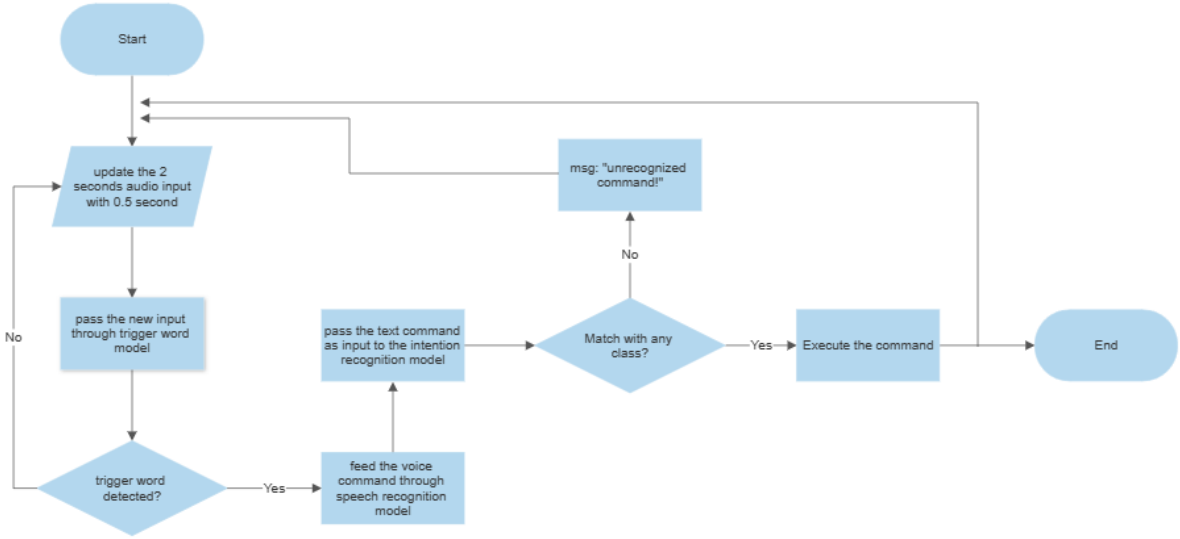
- **The first model (Trigger word detection)** to detect the wakeup word

- **The second model (Speech recognition model)** converts voice commands into text commands
- **The third model (Intention recognition)** classifies the text to one of the tasks the virtual assistant can do

Requirements

- **Software**
  - Python +3.8
  - Pytorch 1.12.1
  - Windows os
- **Hardware**
  - Microphone
  - Pc with a powerful GPU to run the model

Workflow



The first model “Trigger word detection model” is producing inferences at a frequency of every 500 ms. It takes its audio chunk with that length. When the activation word is heard or given a flag (true) it activates, and the user hears a chiming sound then the model will take us to the second stage. The second stage “Speech recognition model” is activated by the wakeup/trigger word model and it takes the audio signals as an input and returns text as an output. This model converts the voice command to a text command so we can classify it with the third model. The third stage takes the output of the second stage as a text then it classifies it to the corresponding class. and its output refers to a specific task that is assigned a class (number).

Stages

1. Trigger word detection model

Every voice assistant has a name like Amazon Alexa, Google Home, Apple Siri, and Baidu DuerOS to wake up upon hearing a particular word and our model is called Rem and its wake-up word is “wake up Rem”. The trigger word model is supposed to be the gate for the virtual assistant as it detects the wake-up word (which is “wake up Rem”) in our case. We need this to be as light as possible as it will be working the most among the other models. Also, it should have a high accuracy as it’s trained to detect only one word. Another

advantage for using it is that the virtual assistant won't do any task without being activated by the user.

1. Data Synthesis

First of all, we need to have a dataset to train a model. Since we want the model to detect our custom wake-up word, we should build this dataset.

1. Building the dataset

My friends and I recorded ourselves saying the positive wake-up word and some negative random words. We also recorded some background noises from different places.

In the raw_data directory, you will find 3 more directories (positives - negatives - backgrounds) containing .wav audio files which are the records after being edited. Typically, the positive and negative directories contain records in less than 2 seconds length and the background records are just 2 seconds long. We'll be using these audio files to synthesize our dataset which will be used later to train our model.

2. Synthesizing the dataset

In this section, we will present how we made 4000 training samples out of the records we mentioned above. First, why do we need to synthesize the dataset, can't we just record it? Yes, recording the dataset looks easier but It will be a difficult task to label it accurately. In addition to that, the dataset collection would be slow if we recorded all the samples.

The process we are using to synthesize the data involves the following steps:

- Picking a random 2 second background clip.
- 50% chance to insert a random positive clip in the background clip.
- Less than 50% chance to insert a random negative clip in the background clip.
- Labeling the data after the positive word is said.
- Take the spectrogram of every sample and we will talk about that later.
- Repeating that for 4000 times.

That way we can have many training samples that aren't similar at all. Furthermore, now we know exactly when the positive word is said, and it will be easy to label the data. The way we are going to label the data will be presented later. To implement the process above we are going to walk through the functions we made. But before we do that you have to know that the package we are using ("pydub") samples audio clip with timesteps of 1 ms:

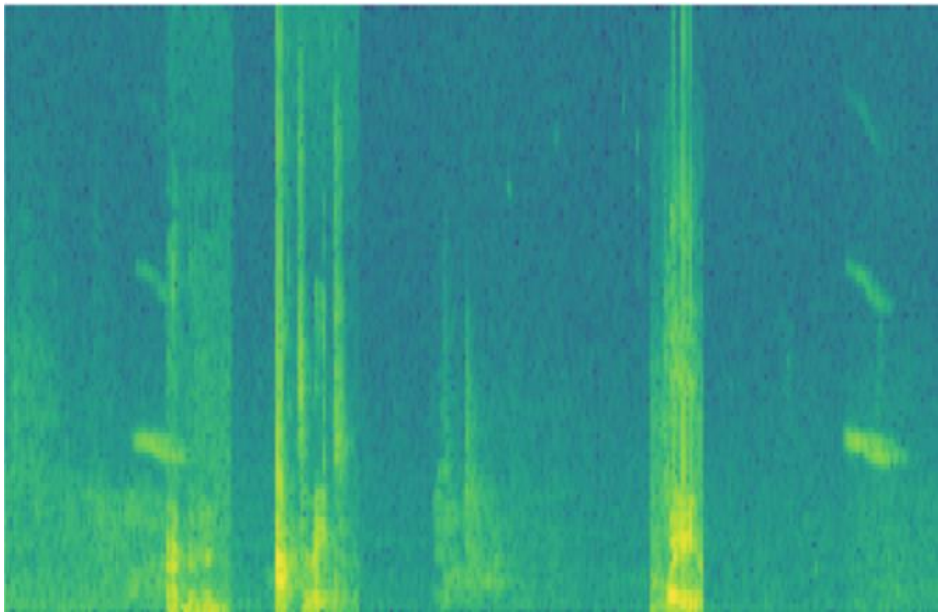
- get_random_time_segment: this returns a random time segment (0:1950 ms).

- is_overlapping: checks if the time segment is overlapping with any time segment that was added to a single sample before.
- insert_audio_clip: inserts an audio segment at a random time in our background audio using the 2 functions above.
- insert_ones : inserts 1s right after the positive word is said (fills 50 timesteps with 1s).
- create_training_example : uses all the functions above to create a single sample returning the spectrogram of this sample as a numpy array and the label list.
- make_training_set : generates the number of samples we need (4000 in our case) collects them in a single numpy matrix then saves them into a .npy file.

3. Converting the audio samples to spectrograms

If you noticed the function that was creating the training samples was returning a numpy matrix representing the spectrogram of that audio clip. But what is the spectrogram? and why are we using it?

First let's discuss how the computer represents the audio recordings. A microphone records little variation in air pressure over time, and it is these little variations in air pressure that our ears also perceive as sound. The audio we are using is sampled at 48KHZ (48000 value for every frequency every second) which means the training sample clip will give 98k number for each frequency and that's too much for a model input. That's why we compute the spectrogram for the inputs before it's fed into the model. The spectrogram tells us how the energy of different frequencies in an audio signal changes over time.



For a 2 seconds audio clip sampled at 48KHZ the spectrogram function we are using will return a numpy array in size (1198,101)

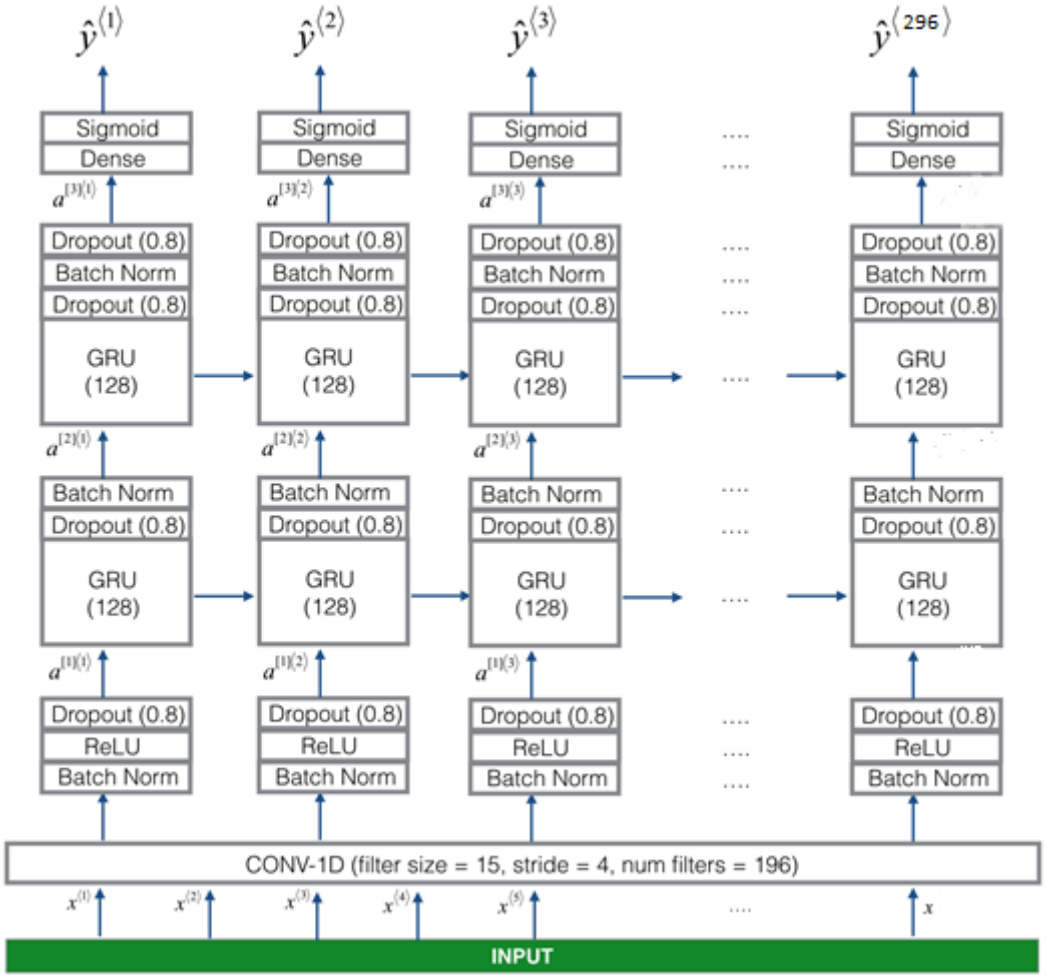
Now we can define our key values that we are going to use in this model :

Tx = 1198 (represents the spectrogram output and one dimension of the model input).

N_freq = 101 (the number of frequencies computed by the spectrogram function and another dimension for the model input).

Ty = 296 (the model output length).

2. Model architecture



The 1D convolutional step inputs 1198 timesteps of the spectrogram (2 seconds), outputs a 296 step output. This will work as a feature extractor and will also speed up the model as it will reduce the number of hidden cells of the GRUs above. The GRU reads the output of the conv layer after being normalized (from left to right). The 2nd GRU layer is followed by a linear network and a sigmoid layer to classify the output in range from 0 to 1.

3. Real-Time predictions

Like all virtual assistants, it should respond quickly with minimal time delay. So we need to provide a continuous audio stream that enables the model to generate predictions in real-time. The audio steam will update the model's input with a 0.5 sec

chunk. This will allow the virtual assistant to detect the wake-up word and interact with the user in under 0.5 seconds.

2. Speech recognition model

For the speech recognition phase of our project, we utilized the SpeechRecognition package, a Python library that provides support for several speech recognition engines. This package allowed us to easily integrate speech-to-text functionality into our project, enabling it to transcribe spoken language into text. We used the famous open ai speech recognition model Whisper, a Transformer based encoder-decoder pretrained model. This model is also referred to as a sequence-to-sequence model. It was trained on 680k hours of labeled speech data annotated using large-scale weak supervision.

The whisper offline api function is passed an audio recording and the language to recognise returning back the transcribed text.

3. Intention recognition model

Although the virtual assistant can recognize text commands, it still requires a way to determine the user's intention and take appropriate actions. This is where an intention recognition model comes into play. Intent recognition, also commonly referred to as intent classification, uses machine learning and natural language processing to associate text data and expression to a given intent. This model will take a text command (speech recognition output) as the input and will output a one-hot encoded matrix referring to the command predicted class. For a task like that we will be fine-tuning a pretrained NLP model called “BERT” (Bidirectional Encoder Representations from Transformers). So we will be using the Hugging Face library to access this model and fine tune it. The library also contains the BERT tokenizer that we will be presenting in the next section.

A. Data preparation

Just for the development we made a small training set with 89 labeled samples. As we are fine tuning BERT the text input should be passed through a tokenizer, BERT tokenizer is a specific type of tokenizer used in NLP to preprocess text input data for use with BERT. This tokenizer is designed to split input text into individual tokens, which are then converted into numerical representations that can be input into the BERT model. The tokenizer uses a WordPiece algorithm, which breaks words down into smaller subwords and generates a vocabulary of subwords that can be used to represent any word in the input text. It also performs other tasks such as adding special tokens to the input sequence, padding sequences to a fixed length, and generating attention masks to indicate which tokens should be attended to during the model's attention mechanism.

Tokens	[CLS]	don	'	t	like	it	!	[SEP]	[PAD]	[PAD]
Token IDs	101	2123	1005	1056	2066	2099	999	102	0	0
Token Type IDs	0	0	0	0	0	0	0	0	0	0
Attention Mask	1	1	1	1	1	1	1	1	0	0

B. Model architecture

```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
├─BertModel: 1-1                        --
│    └─BertEmbeddings: 2-1              --
│    │    └─Embedding: 3-1              22,268,928
│    │    └─Embedding: 3-2              393,216
│    │    └─Embedding: 3-3              1,536
│    │    └─LayerNorm: 3-4              1,536
│    │    └─Dropout: 3-5                --
│    └─BertEncoder: 2-2                 --
│    │    └─ModuleList: 3-6             85,054,464
│    └─BertPooler: 2-3                  --
│    │    └─Linear: 3-7                 590,592
│    │    └─Tanh: 3-8                   --
├─Dropout: 1-2                          --
├─Linear: 1-3                           6,152
├─ReLU: 1-4                             --
==========================================================================
Total params: 108,316,424
Trainable params: 108,316,424
Non-trainable params: 0
==========================================================================
```

As we see the layers represented in the figure above. The 3 embedding layers are (Token embeddings layer - segment embeddings layer - positional embeddings layer) respectively followed by the 12 transformer encoders then the output of the encoder layers is fed into a pooling layer, which aggregates the contextualized embeddings produced by the encoder layers into a fixed-length vector representation. The pooled output vector will be used as an input to our fine-tuning linear layer.

Conclusion

In this project, we have proposed a Virtual assistant that consists of 3 models. First the trigger word detection model where we synthesized the data ourselves and trained it to detect our custom trigger word. Second the speech recognition model where we used the open ai whisper pretrained model. Lastly, the intention classifier/recognition model where we fine-tuned the BERT pretrained model.

Car control system

A car control system refers to the collection of components and technologies that are responsible for controlling various aspects of a vehicle's operation. These control systems play a crucial role in ensuring the proper functioning and safety of the vehicle.

The car control system in this project consists of three subsystems: moving, sound system, and monitoring the car.

Car movement system

One of the most important aspects of car control systems is the control of vehicle movement. This includes systems that help in maintaining stability and control during acceleration, braking, and turning. These systems utilize various sensors and actuators to monitor and adjust parameters such as wheel speed, steering angle, and vehicle pitch and roll angles.

Many applications involving the use of changing regulated speed motors for various loads, require the usage of Direct Current motors. DC motors are available in a variety of power and speed configurations. The simple approaches for controlling them, such as pulse width modulation, and the excellent performance of DC motors will result in an increase in the quantity and variety of applications in the future. Wireless communication, on the other hand, is widely used in communication and information exchange, particularly with the widespread usage of cell phones. Hence car movement in this project is done using a joystick controller which allows for wireless and precise control of steering as well as speed of the car. Moreover, the driving system of the DC motor consists of NodeMCU with ESP8266 wifi chip, and L298N H-bridge motor driver. This driving system is used to drive two DC gear motors which are used to control the motion of the car where each dc motor controls two wheels, one for front-wheels which provide steering control, and the other is for the rear-wheels which is responsible for speed control.

We had a few ideas on how to control our car, but using a PlayStation 4 gamepad to do so appealed to us the most. The connection of PS4 gamepad depends on Bluetooth connection so we had many options,

- **Connect gamepad directly to Microcontroller (arduino)**

  This option has some problems:

  -It would be hard to connect gamepad directly to arduino without bluetooth shield (which will not allow us to connect motors).

  -It would be hard to calibrate gamepad using arduino (or C driver).

- **Use an intermediate connection between gamepad and microcontroller(like laptop or raspberry bi)**

  -This option will allow us to calibrate the gamepad easily and connect the microcontroller to the laptop.

  -However, this method is problematic because the Bluetooth connection between the laptop and microcontroller (via Bluetooth module) is poor, occasionally breaks down for no clear reason, and has a limited range (only 4 meters).

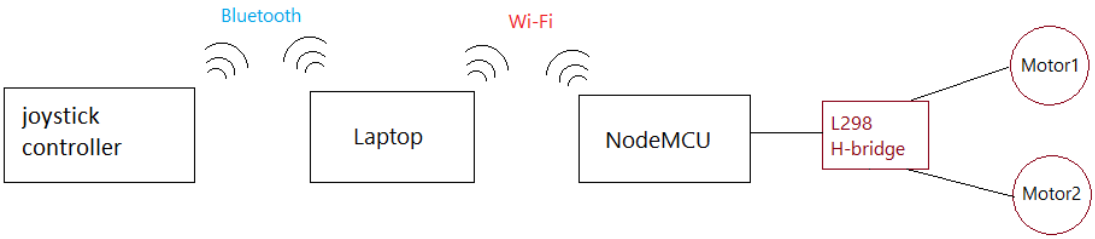
- **Use an intermediate connection but connect microcontroller (Nodemcu) using WIFI**

This option solves most of Bluetooth's problems (i.e sudden disconnect and short range) and it also adds some features that will be mentioned later.

Components

- PS4 gamepad
- NodeMCU-V3 ESP8266
- MicroUSB
- L298N motor driver
- 6V DC motor (x2)
- Robot car prototype
- Intermediate(laptop or pi)
- 12V battery
- DC-DC step-down converter

System Block Diagram



Gamepad Calibration

Steps followed:

- Connect the gamepad to laptop
- Calibrate the gamepad
- Create control message
- Send the control message to Websocket server

1. **Connect gamepad**
   Gamepad's Bluetooth connection makes it incredibly simple to connect to a computer, and the connection between the computer and gamepad is excellent and has a great range because the gamepad was designed to control games quickly and from a large distance.

---

2. **Calibrate the gamepad**
   In this phase we used a library in Python named **pygame.**

- First thing first we initialize the gamepad in the file

```
pygame.joystick.init()
```

- Then we put all connected gamepads IDs in an array then choose which gamepad we are going to use and use it as object

```
joysticks = [pygame.joystick.Joystick(x) for x in range(pygame.joystick.get_count())]
pygame.init()
JOYSTIC = pygame.joystick.Joystick(0)
```

- Then we tried to get all button IDs using method “pygame.event.get()” as we have several events (I will only mention used events)

– pygame.JOYBUTTONDOWN ==========> detects button press state
– pygame.JOYBUTTONUP =============> detects button release state
– pygame.JOYAXISMOTION ===========> detects joysticks motion
– pygame.JOYDEVICEREMOVED ======> detects gamepad disconnection

When we try to get button IDs we use:

```
    for event in pygame.event.get():
        if event.type == pygame.JOYBUTTONDOWN:
            print(event)
```

And the results is shown in image 1-1

```
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 3})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 1})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 0})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 2})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 10})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 9})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 10})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 0})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 2})>
<Event(1539-JoyButtonDown {'joy': 0, 'instance_id': 0, 'button': 13})>
```

(image 1-1)

As we can see we have the button IDs of our gamepad (id = 0) which can be used to do action and the result that we have 16 buttons and their IDs are like image 1-2 shows

(image 1-2)

The same way is followed when getting Joystick ID and value ranges

```
if event.type == pygame.JOYAXISMOTION:
```

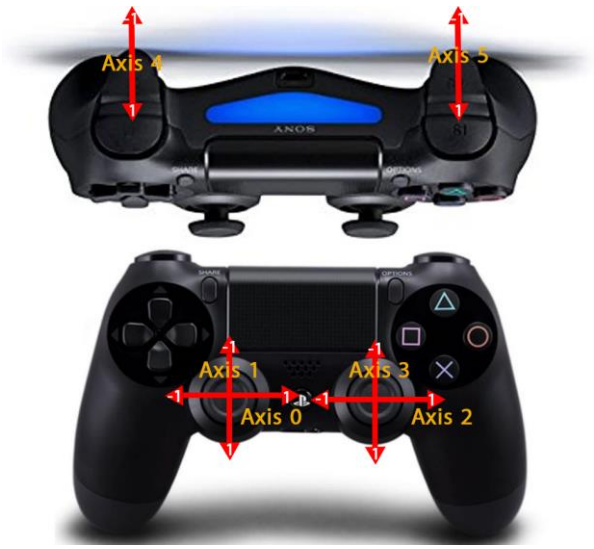
```
            print(event)
```

And the result is shown in (image 1-3)

```
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 2, 'value': 1.0})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 2, 'value': 0.003906369212927641})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 2, 'value': -1.000030518509476})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 2, 'value': 0.003906369212927641})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 0, 'value': 0.003906369212927641})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 0, 'value': 1.0})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 0, 'value': 0.003906369212927641})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 0, 'value': -1.000030518509476})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 0, 'value': 0.003906369212927641})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 5, 'value': -1.000030518509476})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 5, 'value': 1.0})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 5, 'value': -1.000030518509476})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 4, 'value': -1.000030518509476})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 4, 'value': 1.0})>
<Event(1536-JoyAxisMotion {'joy': 0, 'instance_id': 0, 'axis': 4, 'value': -1.000030518509476})>
```

(image 1-3)

We have 6 joy axis motions and range of values of each is between -1 and 1



(image 1-4)

---

3. **Create control message**

   After knowing events and IDs it is time to determine what objects control our car and create control message

   As followed in most of car control games we decided to:

   - Use R2- joy (axis 5) to move car forward and give it speed values between 0 and 5
   - Use L2- joy (axis 4) to move car backward and give it speed values between 0 and 5
   - Use L3- joy (axis 0) to move car Right and left and give it motor speed values between 0 and 5 for each direction
   - Use R1-button (button 10) to stop all motors
   - Use O-button (button 1) and throw-button (button 3) to control lights (optional)

- Use PS-button or event pygame.JOYDEVICEREMOVED to terminate calibration program

Note: - In joy-axis we mapped values range from (-1 – 1) to (0 – 5)
- If calibration program terminates the car will stop

Then the Control frame is created as following:

Stop	Forward	Backward	Right	Left

Example of Frame creation:

```
send_request = "0"+ str(int(speed)) + str(int(back_spd)) + str(int(Right)) + "0"
```

- Stop values can be 0 to allow motion, and 1 to stop car and it's set to 1 in three states
  1. R1-button is pressed
  2. PS-button is pressed
  3. Gamepad disconnected

No two opposite values are set together

For example in (image 1-5) the first frame indicates that car is stopped and second frame the car moves forward with max speed (speed 5)

```
R1_Stop_10000
R2_05000
R2_05000
R1_Stop_10000
AR_00000
AR_00000
AR_00050
AL_00005
L2_00500
L2_00500
R1_Stop_10000
R2_05000
R2_05000
AR_05050
AR_05050
AR_05050
AL_05005
AL_05004
R1_Stop_10000
```

(image 1-5)

For any additional feature we want to control (e.g lighting) we just activate its button and assign it a place in the control frame.
Each value is casted to string and the created frame gets sent to the server to broadcast to the clients.

Writing calibration code covered all conditions manually to avoid errors possible and it's divided into three stages with three events inside a "while true loop"

- event.type == pygame.JOYBUTTONDOWN: to take needed buttons readings
- event.type == pygame.JOYAXISMOTION :to take needed axis reading
- event.type == pygame.JOYDEVICEREMOVED: to sense if the gamepad disconnected

4. **Send the control message to Websockets server**

After creating control message it's time to send to websockets server so we used library **`websockets`** in python which is synchronous in sending messages at every change in gamepad which makes a problem of time delay as it takes time to connect, send and disconnect from server which may stuck the rest of the program and may causes an accident to the car.

So what is the solution?

Solution is to use **`asyncio`** library which makes it easy to send very quickly which is better at response and goo in safety

This how to connect to our server

```
#connect to websocket server
async def connect(msg):
   url="ws://localhost:8000"

   async with websockets.connect(url) as websocket:
      await websocket.send(msg)
```

This shows how to send async message to server

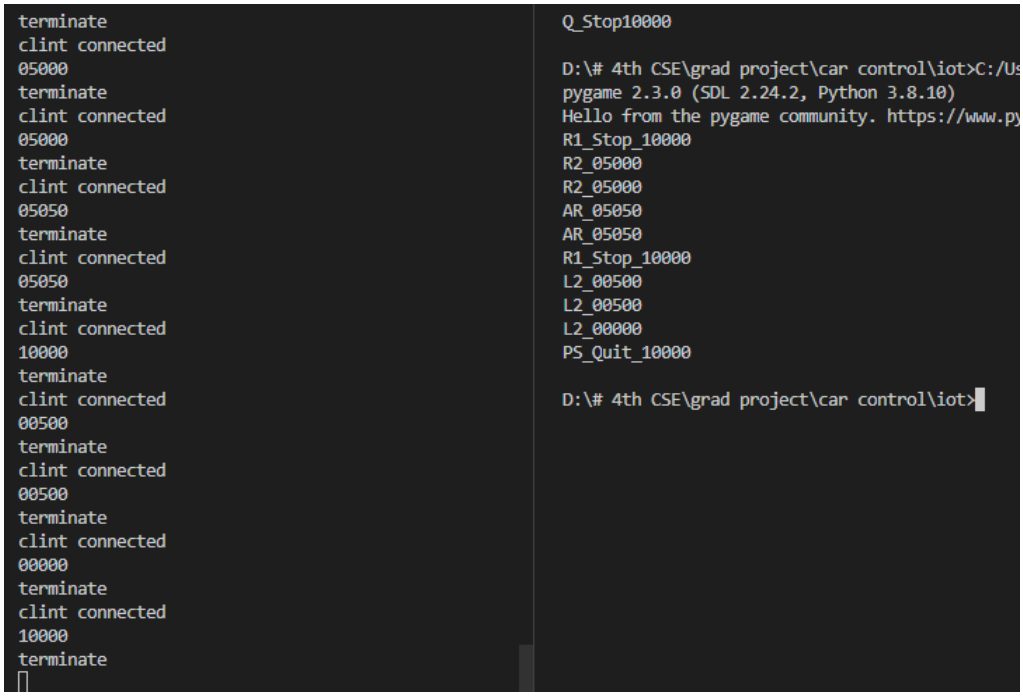
```
#Send data
               asyncio.get_event_loop().run_until_complete(connect(send_request))
```

Messages received by server shown in image 1-6

```
clint connected
02000
terminate
clint connected
02000
terminate
clint connected
02000
terminate
clint connected
02000
terminate
clint connected
02000
terminate
clint connected
02000
terminate
```

(image 1-6)

Data transmission between celebration program and server site:



(image 1-7)

Server-side Implementation

The server-side acts as a bridge between the hardware and software components, enabling seamless communication and control. NodeJs with WebSocket is used to build this server. NodeJs is a powerful and versatile JavaScript runtime environment that allows developers to build scalable and high-performing applications. Unlike traditional web development NodeJs enables the execution of JavaScript code on the server-side. This opens up a world of possibilities, as developers can now create entire web applications, APIs, and even real-time communication systems using JavaScript as the primary language. NodeJs utilizes an event-driven, non-blocking I/O model, which makes it highly efficient and capable of handling a large number of concurrent requests. WebSocket is a communication protocol that provides full-duplex communication channels over a single TCP connection. It enables real-time, bidirectional communication between a client and a server.

In this part of the chapter, we will use NodeJs with WebSocket to build a server to establish a connection between the Gamepad and the NodeMCU in order for the car to be controlled seamlessly.

Process

Initially a port number is assigned to the server which clients use to connect to the server.

To check that the server is running, we use app.listen() which listens for connection and when a client connects it prints a message to the server console.

```
const express = require("express")
const app = express()

const PORT = 8000

app.use(express.json());

const server = app.listen(PORT, function () {
    console.log("server running");
});

app.get("/", function (req, res) {
    console.log(req.body.request)
    res.sendFile(__dirname + "\\index.html");

});
```

To connect NodeMCU and the controller to the server we use WebSocket.
First, we import WebSocket as a server,
Then assign the “server” in a new WebSocket server “wss”.
Define a variable “value” which will hold the encoded frame sent by the gamepad.
WebSocket offers a group of events that are helpful for communication between client and server, the ones used here are:

- wss.on(“connection”) : when a connection is established it prints a message on the server console and sends a message to the client.
- ws.on(“message”) : when a message is received from a client, it converts it to string then it prints the message on console and resends it to all connected clients using broadcast().
- ws.on(“close”) : when a client is disconnected a message is printed on the console.

```
const SocketServer = require("ws").Server;
const wss = new SocketServer({ server });

value = "0";
wss.on("connection", function (ws) {
    console.log("clint connected")
    ws.send(value);

    ws.on("message", function (msg) {
        value = msg + "";
        console.log(value);
        broadcast(value);
    });

    ws.on("close", function () {
        console.log("terminate");
    });
});
```

broadcast() is a function that sends a message or data to multiple connected clients simultaneously. However, it is not a built-in function so we define it as follows:

```
function broadcast(msg) {
    wss.clients.forEach(function (client) {
        if (client.readyState === client.OPEN) {
            client.send(msg);

        }
    });
```

It checks if each client is connected then sends the message or data using client.send().

Motors Control

Motor control involves the management and regulation of motor operations across diverse applications. In the context of this project, motors are employed in car movement. The control of car movement is achieved through the utilization of a NodeMCU, which is coupled with two 6V DC motors and an L298N motor driver.

DC Motor

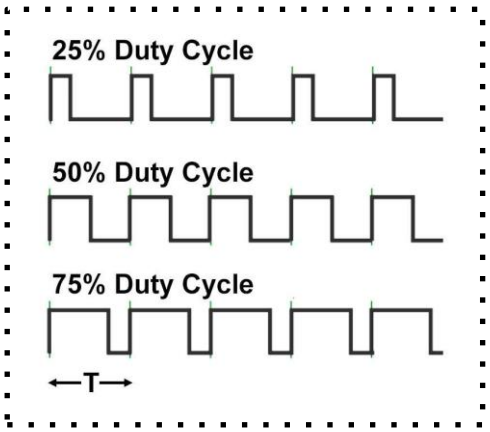
A DC motor has two aspects to control: speed and direction. To change motor direction, the direction of current flow through the motor is changed. To vary motor speed the voltage across the motor terminals must be varied. In order to control the speed and direction of the DC motor we use PWM and H-bridge .

PWM

Pulse width modulation (PWM) is the most common way to control a DC motor's rotational speed. By lengthening or shortening the time period when the control signal has the logic value 1 (t_{ON}), the

motor's rpm can be changed. As a result, the motor will run at its maximum speed, when t_{ON} is maximal, and t_{OFF} is zero at a duty-cycle of 100%. The motor will spin at half its rated speed at a 50% duty cycle, therefore $t_{ON} = t_{OFF}$. If the duty cycle is zero percent, the motor will be turned off, causing t_{OFF} to be at its maximum and t_{ON} to be at zero percent. The duty-cycle is noted by d and can be calculated through the relation:

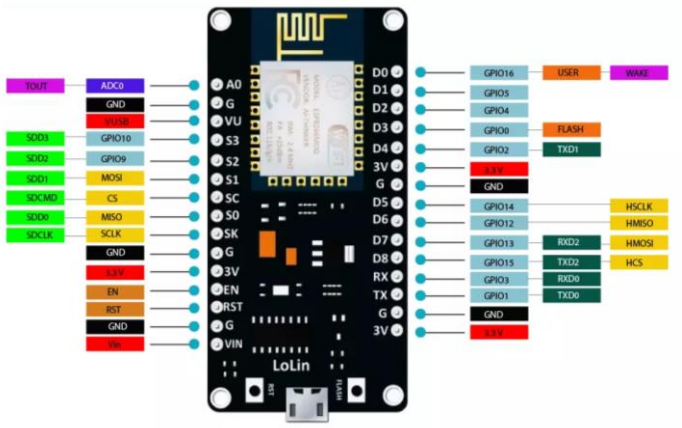
$$d = \frac{t_{ON}}{t_{ON} + t_{OFF}} * 100\%$$



(image 1-8)

NodeMCU PWM

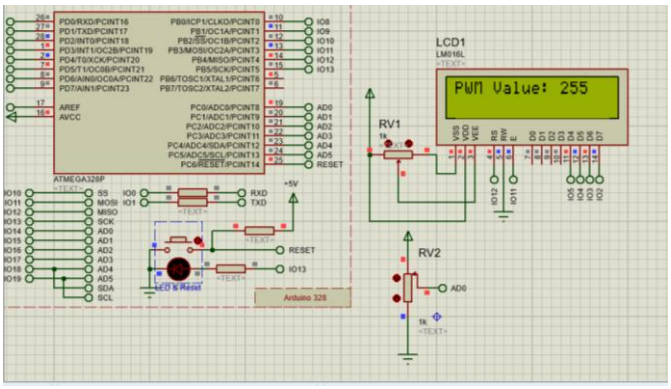
ESP8266 supports PWM on all general-purpose input-output pins GPIO0-GPIO16 with 10-bit resolution. However, for best optimization we work in 8-bit resolution hence the duty cycle will vary from 0-255. So this value can be in the range 0-255 where 0 denotes no PWM and 255 denotes a duty cycle of 100%.



(image 1-9)

The PWM frequency on NodeMCU can typically be adjusted from around 1 Hz up to several kilohertz. The duty cycle, expressed as a percentage, determines the proportion of time the signal is high (on) compared to the total period.

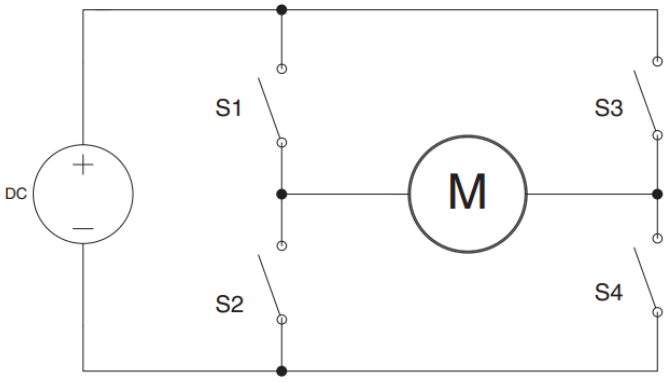
As seen in the simulation the PWM varies with the potentiometer from 0 to 255.



(image 1-10)

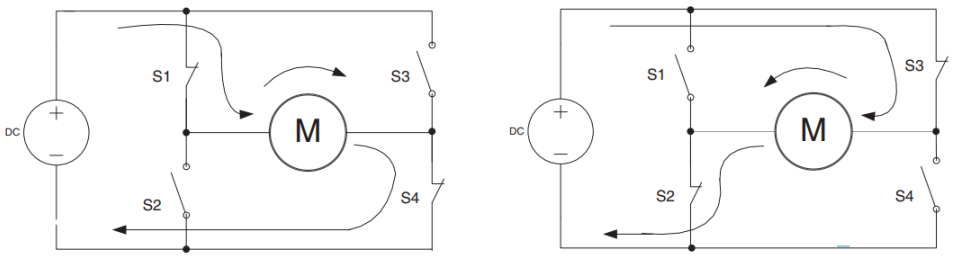
H-bridge

PWM signals are typically produced by microcontrollers. Microcontrollers serve as the logic control or decision component and DC motors serve as the acting component in an electric drive system. The acting component runs at high voltages and power rates while the control component requires 5V voltage level and low power. The two components of the system must therefore have an interface. H-bridge is an amplifier that frequently serves as a galvanic separation between the two components of the electric drive system which make up such an interface. (image 1-11) depicts the block diagram of the H-bridge-based DC motor control system.



(image 1-11)

Four diagonally operated electrical switches make up the bridge above. (image 1-12 a) shows what happens when the switches S1 and S4 are turned on: a positive voltage is given to the DC motor, which rotates clockwise. When the switches S2 and S3 are turned on (see image 1-12 b), the voltage polarity is reversed, allowing the motor to rotate counterclockwise.



(image 1-12)

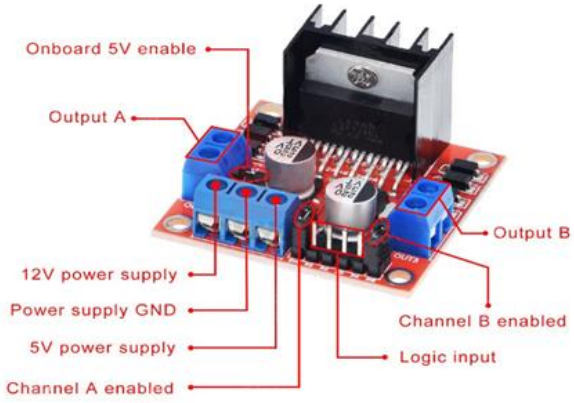
- **L298N Dual-Hbridge**

  IN1, IN2 : MOTOR 1 DIRECTION

  IN3, IN4 : MOTOR 2 DIRECTION

  ENA : MOTOR1 SPEED

  ENB : MOTOR2 SPEED



- **4-channel Logic level converter**

  The control pins of L298N (IN, EN) operate at 0-5V which is not provided by NodeMCU as it has 0-3.3V output voltage. To deal with this we utilized a 4-Channel Logic level converter, which maps between the two ranges 0-3.3V and 0-5V. Logic level converter is powered by the 5V L298N supplies.

- **DC-DC step-down converter**

Motors in the car movement system require 6V to revolve at their maximum rate. However, the battery provides 12V which would damage the motors. The solution was connecting the battery to a DC-DC step-down converter and mapping it to 6V, which will be sent to the L298N to control both motors.

Process

The program has two main goals:

1. Connect NodeMCU to the server.
2. Control the motors.

1. Connect NodeMCU to the server:

To write for a NodeMCU we include ESP8266WiFi library, and to connect NodeMCU to the server as a client we include WebSocketsClient library.

```
#include <ESP8266WiFi.h>
#include <WebSocketsClient.h>
```

Then, we define the wifi name and password which we use to set up the Wi-Fi connection.

```
const char *ssid = "Dark";
const char *pass = "123456789";
```

Next, we define the IP address and port number which we use to start the connection with the server.

```
#define SERVER   "192.168.137.1"
#define PORT     8000
#define URL      "/"
```

In case the Wi-Fi is not connected yet, a dot is printed every .5s on the serial monitor.

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

When a connection is established, the Wi-Fi name and the IP address are printed on the serial monitor.

```
.
Dark
192.168.137.162
```

Initiate the connection between the client and the server using the predefined SERVER, PORT, and URL.

```
wsc.begin(SERVER, PORT, URL);
```

Upon receiving an event go to the “websocketEvent” function.

```
wsc.onEvent(websocketEvent);
```

The “websocketEvent” function is a function with void return type and three parameters:
“type”: holds the event name,
“data”: is an array of characters holding the content of the received message,
“length”: holds the number of characters in “data”.

```
void websocketEvent(WStype_t type, uint8_t *data, size_t length){
```

This function checks on “type” to assign actions to each event, here there are three main events:

- WStype_CONNECTED: when the client has connected successfully to the server.

It prints “Connected to server” on serial and sends a message to the server.

- WStype_DISCONNECTED: when the client has disconnected from the server.

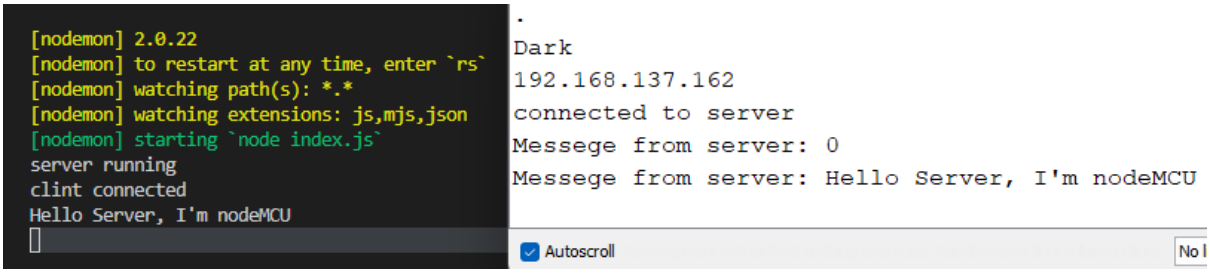
It prints “Disconnected” on the serial monitor.

- WStype_TEXT: when the client receives a message from the server, which is received in the “data” parameter.

It prints the message on the serial monitor.

```
case(WStype_TEXT):
  Serial.printf("Messege from server: %s\n",data);
  break;
```

Testing the connection



(image 1-14)

Control the motors

The two motors are responsible for speed and direction of wheels, as the first motor controls the rear wheels speed and direction which is responsible for car speed and direction of movement forward or backward. The second motor,on the other hand, is in control of the front wheels speed and direction which affect the direction of turning right or left and the speed of turning.

First, we define the NodeMCU pins to its corresponding pins in L298N for both motors.

To control both motors speed and direction of rotation we created a function named "`rotate`" with void return value and three parameters:
"motor" which holds the motor name,
"speed" which holds the speed as a value between 0 and 255,
and "dir" which holds the direction of rotation either CW or CCW.

Define the function as follows:

```
void rotate(int motor, int speed, int dir){
  if(motor == 1){
    if(dir == 1){
       digitalWrite(IN1, HIGH);
       digitalWrite(IN2, LOW);
       analogWrite(ENA, speed);
      }else if(dir == 2){
       digitalWrite(IN1, LOW);
       digitalWrite(IN2, HIGH);
       analogWrite(ENA, speed);
        }
```

The code receives the gamepad's sent values in the "data" parameter as an eight-byte frame; only five of them are used here. Each byte represents a motor state so they are assigned to variables as follows:

```
case(WStype_TEXT):
  Serial.printf("Messege from server: %s\n",data);
  char stop,forward,backward,right,left;
  stop = data[0];
  forward = data[1];
  backward = data[2];
  right = data[3];
  left = data[4];
```

Byte no.	Name	Value	Motors state
1	stop	1	no rotation
2	forward	1-5	Motor1 rotates CW
3	backward	1-5	Motor1 rotates CCW
4	right	1-5	Motor2 rotates CW
5	left	1-5	Motor2 rotates CCW

The speed is converted from the range 1-5 to PWM values 0-255 by multiplying the range 1-5 by 51.

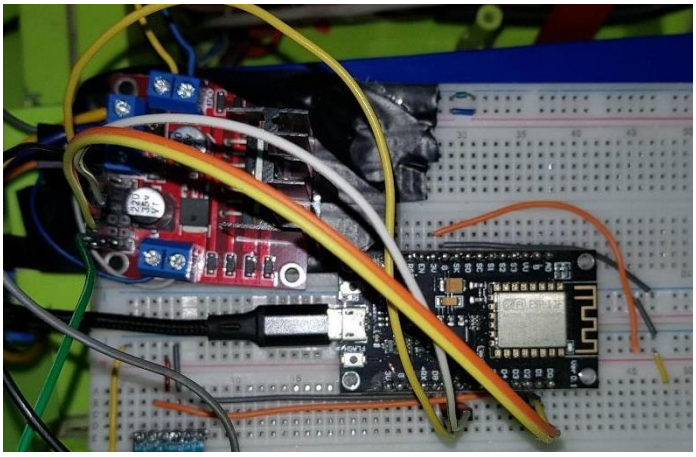
```
case '0':
    if(forward != '0') rotate(motor1, 51 * (forward - '0') , CW);
    else if (back != '0') rotate(motor1, 51 * (back - '0') , CCW);

    if(right != '0') rotate(motor2, 51 * (right - '0') , CW);
    else if (left != '0') rotate(motor2, 51 * (left - '0') , CCW);
```

Steps of implementation

1. Power on Nodemcu.

2.Connect the NodeMCU with L298n driver and motors.

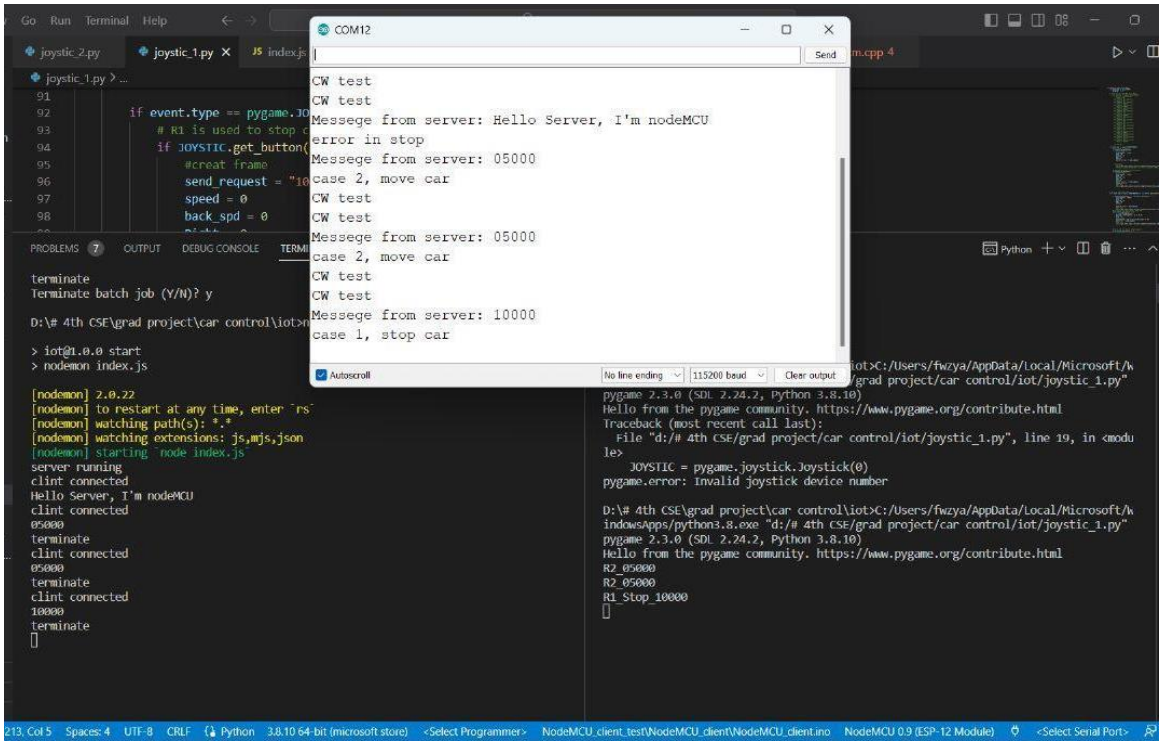
NodeMCU	L298N
D6	IN1
D7	IN2
D8	ENA
D2	IN3
D3	IN4
D4	ENB



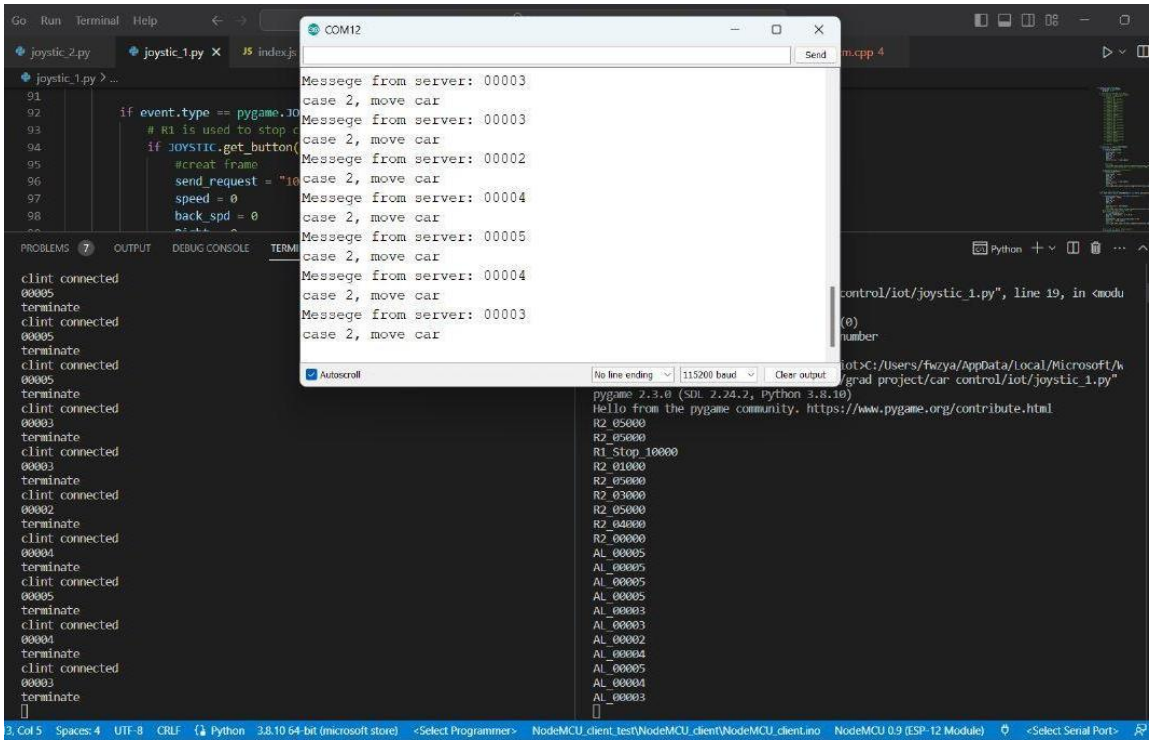
(image 1-15)

3.Connect the gamepad to laptop via Bluetooth

4.Run the server and connect it with the Gamepad and NodeMCU.



(image 1-16)


COM12
Send
Messege from server: 00003
case 2, move car
Messege from server: 00003
case 2, move car
Messege from server: 00002
case 2, move car
Messege from server: 00004
case 2, move car
Messege from server: 00005
case 2, move car
Messege from server: 00004
case 2, move car
Messege from server: 00003
case 2, move car
Autoscroll
No line ending
115200 baud
Clear output
clint connected
00005
terminate
clint connected
00005
terminate
clint connected
00005
terminate
clint connected
00003
terminate
clint connected
00003
terminate
clint connected
00002
terminate
clint connected
00004
terminate
clint connected
00005
terminate
clint connected
00004
terminate
clint connected
00003
terminate
pygame 2.3.0 (SDL 2.24.2, Python 3.8.10)
Hello from the pygame community. https://www.pygame.org/contribute.html
R2_05000
R2_05000
R1_Stop_10000
R2_01000
R2_05000
R2_03000
R2_05000
R2_04000
R2_00000
AL_00005
AL_00005
AL_00005
AL_00003
AL_00003
AL_00002
AL_00004
AL_00005
AL_00004
AL_00003

(image 1-17)

References:

- pygame.joystick — pygame v2.5.0 documentation
- https://websockets.readthedocs.io/en/stable/
- https://docs.python.org/3/library/asyncio.html
- How To Build WebSocket Server And Client in NodeJS – PieSocket Blog
- https://handsontec.com/dataspecs/module/esp8266-V13.pdf
- http://www.handsontec.com/dataspecs/L298N%20Motor%20Driver.pdf

Car Sound System

Introduction

The car sound system has multiple features and one important feature was chosen to be implemented is an **embedded MP3 system**

Cars often have embedded MP3 capability in the sound system to provide drivers and passengers with the convenience of listening to their favorite music while on the go.

Benefits of having embedded MP3 in the car system:

1. *Music enjoyment*: MP3 is a widely used audio format that allows for high-quality music playback. Having an embedded MP3 player in a car allows users to listen to their own music collection or streaming services, enhancing their driving experience.

2. *Convenience*: With an embedded MP3 player, drivers don't need to rely on external devices like smartphones or portable music players to listen to music. They can directly access and control their music library through the car's audio system.

3. *Integration*: An embedded MP3 player is seamlessly integrated into the car's audio system, allowing for easy control and navigation through the car's interface. This integration often includes features like steering wheel controls, voice commands, and touchscreen interfaces.

4. *Storage*: Cars with embedded MP3 players often have built-in storage capacity to store a large number of music files. This eliminates the need for external storage devices and allows users to have their entire music collection readily available in the car.

5. *Connectivity*: Embedded MP3 players in cars may also offer connectivity options such as USB ports, auxiliary inputs, Bluetooth, or Wi-Fi. These features enable users to connect their devices, stream music wirelessly, or play music from various sources.

To create the MP3 sound system for the car we used an Arduino Mega with the method of PCM (Pulse Code Modulation) to process the audio signals in the Arduino IDE including the TMRpcm library. The exact method and implementation will be discussed in detail. So first of all, what is PCM?

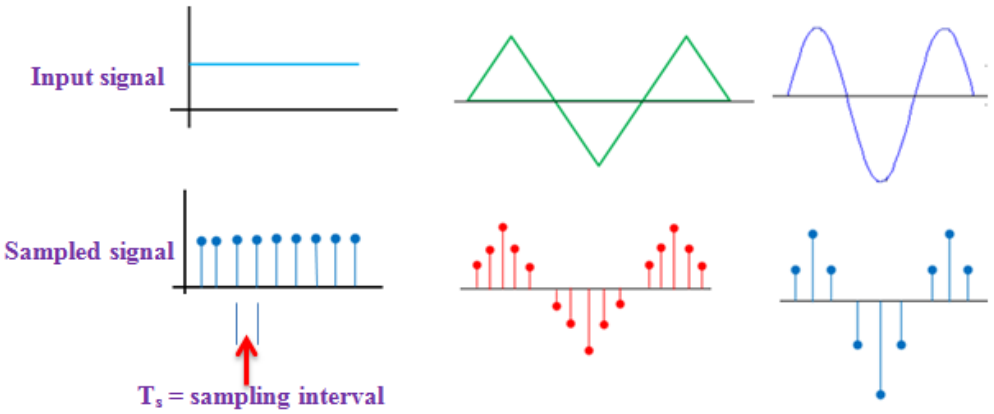
PCM (Pulse Code Modulation)

Pulse Code Modulation is a method used to digitally represent analog signals generally and in our case analog audio signals. It is the standard form of digital audio in digital audio applications and a standard technique for converting analog audio signals into digital format that can be processed and transmitted digitally.

The theory of PCM is summed up in the following steps:

1. Sampling

The first step in PCM is to sample the analog audio signal at a regular intervals. This involves measuring the amplitude of the audio signal at regular intervals. This involves measuring the amplitude of the audio signal at discrete points in time. the rate at which these samples are taken is known as the *sampling rate*, typically measured in samples per second or Hertz (Hz). Common sample rates include 44.1kHz (CD quality) and 48kHz (DVD quality. Example of a sampled signal in the opposite figure.

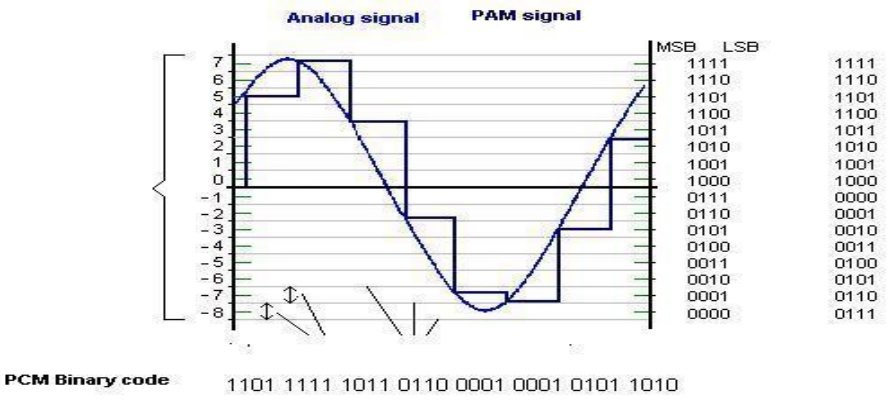


2. Quantization

Once the audio signal is sampled, the next step is to quantize the samples. Quantization involves assigning a digital value to each sample, representing its sample, representing its amplitude. This is done by dividing the range of possible amplitudes into a finite number of levels The number of levels is determined by the bit depth, which specifies the number of bits used to represent each sample. Common bit depths include 8-bit, 16-bit, and 24-bit. The higher the bit depth, the more accurately the analog signal can be represented digitally.

3. Encoding

After quantization, the digital values representing the samples are typically encoded using binary code. In PCM, a common encoding scheme is two's complement. In two's complement, the most significant bit (MSB) represents the sign of the sample (positive or negative), and the remaining bits represent the magnitude. This encoding allows both positive and negative values to be represented, effectively representing the full range of the analog signal.



4. Transmission and storage

The encoded PCM samples can be transmitted or stored digitally. They can be stored in various formats, such as WAV (which was used in our case) or FLAC files, or transmitted over digital communication channels, such as USB or Bluetooth. The receiver or playback device can then decode and reconstruct the original analog audio signal from the PCM samples.

5. Digital-to-Analog Conversion (DAC):

To convert the digital PCM samples back into an analog audio signal, a Digital-to-Analog Converter (DAC) is used. The DAC converts the binary representation of each sample into an analog voltage or current that can be amplified and sent to a speaker or headphones. The quality of the DAC used can affect the accuracy and fidelity of the reconstructed analog signal.

By using PCM, analog audio signals can be accurately represented in a digital format, enabling various digital audio processing techniques such as playback, mixing, and filtering. PCM is widely used in applications such as audio recording, playback devices, telecommunication systems, and more.

PCM in the system

Since the Arduino does not have DAC (Digital to analog converter) this step is skipped and the audio signals are digital but analog like using the PWM Arduino pins.

The Design

Used Hardware Components

- *Arduino Mega 2560 Rev3:*

The microcontroller in the system which deals with the audio signal processing and integration of all elements.

- *TDA2030 Power Amplifier Module:*

It is a simple and compact designed chip used for the audio power amplification. It is cost-effective and versatile as it can provide o/p power ranging from few watts up to 20 watts with a wide supply voltage range. It is designed to provide low distortion audio amplification maintaining the fidelity and quality of the audio signal.

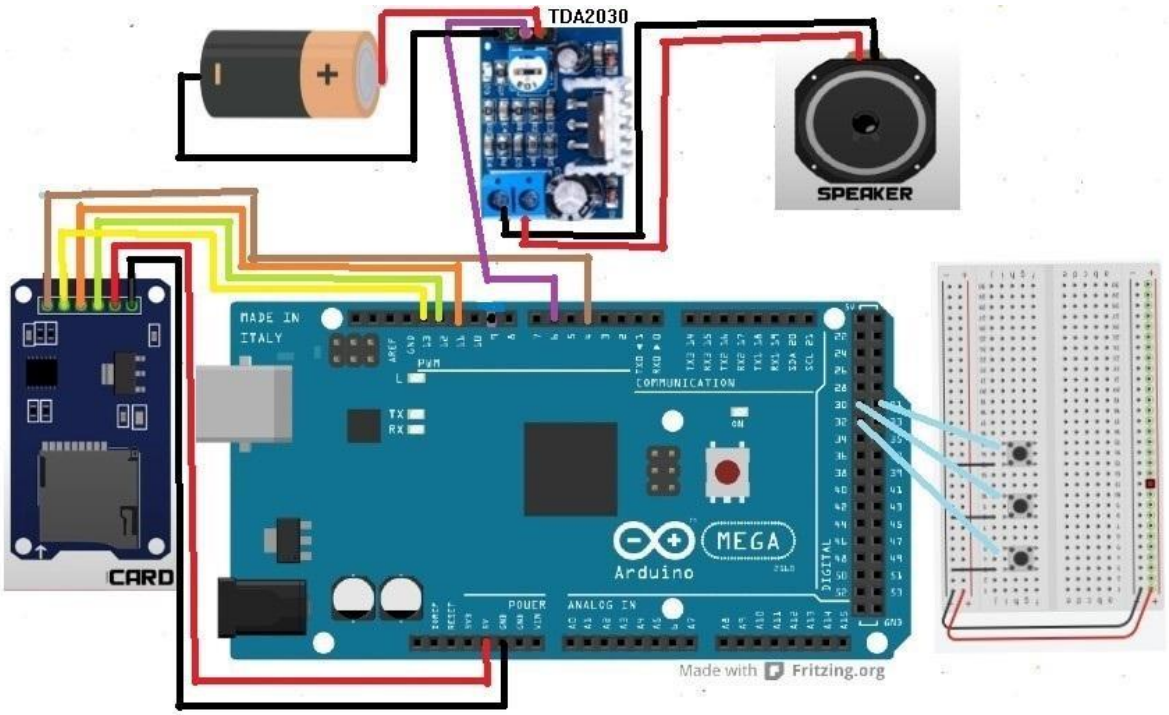
- *SD card*:

The main system storage. It offers large storage capacities, allowing it to store a vast collection of audio files. It is easy and convenient in installment into the system and updating audio files. It gives quick access and Seamless Playback ensuring a smooth and uninterrupted playback. It has high durability and reliability as SD cards are designed to withstand challenging conditions, including vibrations and temperature fluctuations. It has solid-state construction, making it more durable and resistant to physical damage compared to other mechanical storage options.

- Micro SD card Reader Module : Interface for the SD card to the system
- The speaker embedded in the car body

- Push Buttons
- Jumper wires
- Power from the car battery

System Circuit Diagram and connections



Considerations regarding the system

- One push button used to play or stop the track. Another one is used to play the following track and the third is used to play the previous track.
- The SD card must be formatted before injecting the audio files into it
- The audio files is converted and formatted with this tool http://bit.ly/2ZoXgTJ is in the form .wav with the following format:
  Bit resolution: 8 Bit
  Audio frequency: 16Khz
  Audio channels : mono
  PCM format : U8
- The TDA2030 module is connected to the car battery to give better audio amplification

Software writing

The software objective is to program the hardware to perform the following MP3 functionalities:

- Play music when the MP3 launches
- Stop the playing track when the button ButtonOff is pushed down.
- Play the next track when the button Nextplay is pushed down.
- Play the previous track when the button Prevplay is pushed down.

The implementation

- we start by including the needed libraries

```
/*****************************/
#include <SD.h>       // need to include the SD library
#include <TMRpcm.h>   //  Lib to play wav file
#include <SPI.h>     // to use the serial bus
```

Note that the library <TMRpcm.h> is the library that allows the Arduino to deal with the audio files played through Pulse code modulation and the library deals with audio files in form of wav.

```
tmrpcm.play(cdow.wav);
delay(25000);
```

- In the setup, an audio is played as a countdown for the startup of the system and then a delay with the time of the audio is applied

- Implementing the NowPlaying function:

```
146
147  void NowPlaying(int16_t songint) {
148      songplayed = 1;
149      static char *Song;
150      if (songint == 1) {
151          Song = "1bep.wav";
152      } else if (songint == 2) {
153          Song = "2aza.wav";
154      } else if (songint == 3) {
155          Song = "3lna.wav";
156      } else if (songint == 4) {
157          Song = "4mlk.wav";
158      } else if (songint == 5) {
159          Song = "5wsa.wav";
160      }
161
162      tmrpcm.play(Song);
163      // Serial.print("Now Playing: ");
164      // Serial.println(Song);
165      // delay(10000);
166
167      /************** publish now playing to ROS **************/
168
169      snprintf(title_buffer, sizeof(title_buffer), Song);
170      now_playing.publish(&song_title);
171  }
```

The function is responsible for deciding which song is playing. It takes the songint argument. Each song corresponds with a number from 1 to 5 desciding the song based on the number, then actually playing this song; in line 162 using the function 'play' in the TMRpcm library to play the song.

The main program algorithm

- We control the playback of songs based on button presses. It increments or decrements the 'playback' variable to select the next or previous song, and plays the selected song using the 'NowPlaying' function. If the 'ButtonOff' pin is pressed, it stops the playback. It follows the following algorithm:

1. Reads the state of 'ButtonOff', 'Nextplay', and 'Prevplay' pins using 'digitalRead()' function and assigns the values to 'OffState', 'NextState', and 'PrevState' variables.

2. Checks if the 'OffState' is not LOW (meaning the button connected to 'ButtonOff' pin is not pressed).

3. If 'NextState' is LOW (meaning the button connected to 'Nextplay' pin is pressed), increments the 'playback' variable by 1. Resets the 'songplayed' variable to 0. If 'playback' reaches 6, it is reset to 1. Then, it delays for 150 milliseconds.

4. If 'PrevState' is LOW (meaning the button connected to 'Prevplay' pin is pressed), decrements the 'playback' variable by 1. Resets the 'songplayed' variable to 0. If 'playback' reaches 0, it is reset to 5. Then, it delays for 150 milliseconds.

5. Checks if 'songplayed' is false (meaning a new song is not currently playing).

6. If the above conditions are met, calls the 'NowPlaying()' function with 'playback' as an argument to play the corresponding song.

7. If 'OffState' is LOW (meaning the button connected to 'ButtonOff' pin is pressed), stops the playback using 'tmrpcm.stopPlayback()'. Resets the 'songplayed' variable to 0. Delays for 500 milliseconds.

Conclusion

The car sound system can offer many options to be implemented and a lot more features in the system can be implemented in future work but overall for our current work; the inclusion of embedded MP3 capability in cars enhances the driving experience by providing drivers and passengers with a convenient and enjoyable way to listen to their favorite music while on the road which enhances the overall driving experience.

Speedometer

Introduction

Speed is an important characteristic of cars, it represents the ability of a vehicle to cover a given distance within a specific period.

Speed influence various aspects of our daily lives. It affects the efficiency of transportation, emergency response, and even personal comfort. For example, faster cars can reduce commuting times, facilitate rapid medical interventions, and enhance individual and societal productivity and mobility.

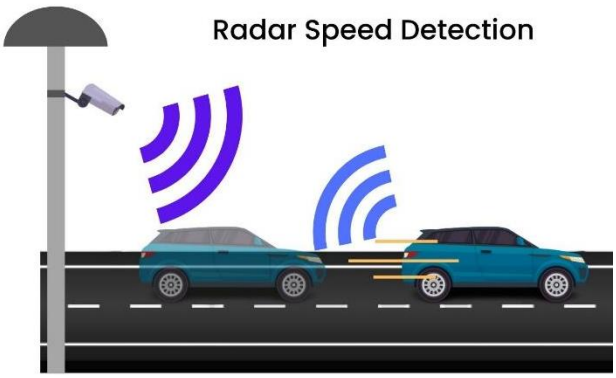
While speed can provide pleasure and excitement, exceeding traffic limits and conditions can result in serious consequences such as accidents, injuries, and death.

Speeding was a factor in 29% of all traffic fatalities in 2021, killing over 33 people per day.

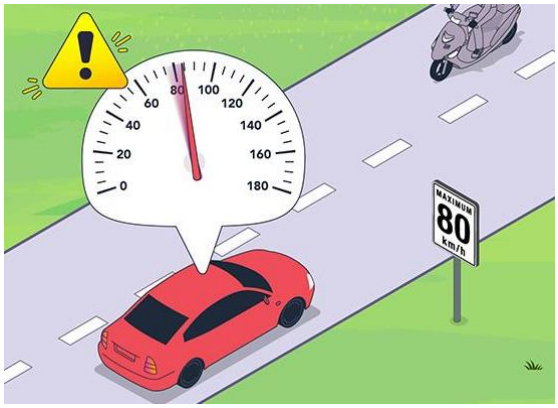
So, it's necessary to measure the speed of cars for several reasons:

- Safety: One of the most important reasons for measuring car speed is to maintain road safety. Speeding is a major factor to road accidents, as higher speeds reduce the driver's ability to react to unexpected situations and increase the stopping distance required. By measuring car speed, authorities can enforce speed limits and take necessary actions to prevent accidents and save lives.

- Law Enforcement: Speed limits are put in place to ensure the safety of all road users. Measuring car speed allows law enforcement agencies to monitor compliance with these speed limits. Radar guns, speed cameras, and automated systems can all be used to detect speeding violations. Speed limit enforcement helps to maintain order on the roads and discourages reckless driving.



- Traffic Management: Measuring vehicle speeds is critical for efficient traffic management. By monitoring the flow of vehicles and determining their speed, transportation authorities can analyze traffic patterns, identify congested areas, and make informed decisions about infrastructure improvements. This information is essential for optimizing traffic flow, reducing congestion, and improving overall transportation efficiency.

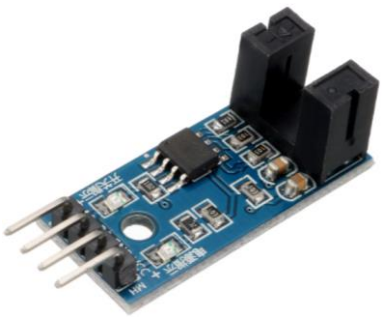


- Infrastructure Planning: Accurate data on car speeds helps in planning and designing road infrastructure. By understanding the typical speed patterns on different types of roads, transportation planners can determine appropriate speed limits, design road signs, and implement safety measures like curves, lane widths, and signage placements. This helps in creating roadways that are safer and more efficient for drivers.

- Research and Statistics: Researchers can study the relationship between speed and accidents, analyze the impact of speed limits on road safety, and identify trends or patterns related to car speed. These findings contribute to evidence-based policy-making and the development of effective road safety strategies.

In our project, we used Photoelectric speed Encoder Sensor to calculate speed of the rear gear.

A photoelectric speed sensor (also known as a photogate) is a device used to measure the speed of rotating objects passing through it such as motors, wheels, or shafts. It consists of a light emitting device (LED), a photodiode or phototransistor receiver, and a slotted disc.



Specifications

- LM393 comparator chip
- Infrared sensor with 5 mm wide slot

- Operating Voltage: 3.3 V to 5 V
- Encoder current consumption: 1.4 mA
- Digital Output: 0 and 1
- Operating temperature: 0 ºC ~ 70 ºC
- Mounting holes for easy installation
- PCB Size: 32 x 14 mm

Features

1. Clean and Stable Digital Signal Output: The sensor produces a clean and stable digital signal output, minimizing signal noise or interference. This feature ensures accurate speed measurement and consistent performance.

2. LM393 Comparator Chip: The sensor utilizes the LM393 comparator chip, which provides a driving ability of 15 mA. This allows the sensor to interface effectively with other devices and systems, ensuring reliable operation.

3. High Speed Detection Capability: The photoelectric speed encoder sensor is capable of detecting and measuring high-speed rotations accurately. It can handle a wide range of rotational speeds, making it versatile for different applications.

4. Compact and Durable Design: The sensor is designed with a compact form factor, making it easy to install and integrate into various systems. It is also built to withstand harsh operating conditions, ensuring durability and longevity.

5. Easy Integration: The sensor is designed for easy integration into existing systems. It can be seamlessly integrated with motors, wheels, shafts, or other components, making it convenient to incorporate into different applications.

6. Compatibility: The sensor is compatible with different types of rotating objects and materials. It can work with a wide range of materials, surfaces, and environments, allowing for flexibility in its usage.

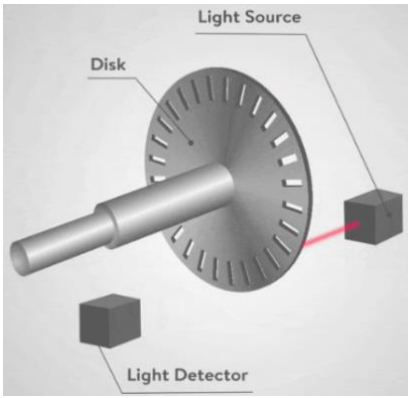
7. Real-time Speed Monitoring: The sensor provides real-time speed monitoring, enabling immediate feedback on the speed of rotating objects. This real-time information is valuable for control systems, automation processes, and quality assurance.

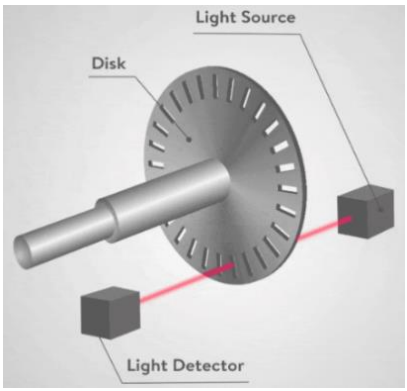
8. Cost-effective Solution: The photoelectric speed encoder sensor offers a cost-effective solution for speed measurement applications. It provides accurate speed measurement capabilities without requiring complex or expensive setups.

How it works:

The transmitter emits a beam of light, usually infrared or laser, across the path where the disk will pass. The emitted light forms a beam or a series of beams that are directed towards the receiver.

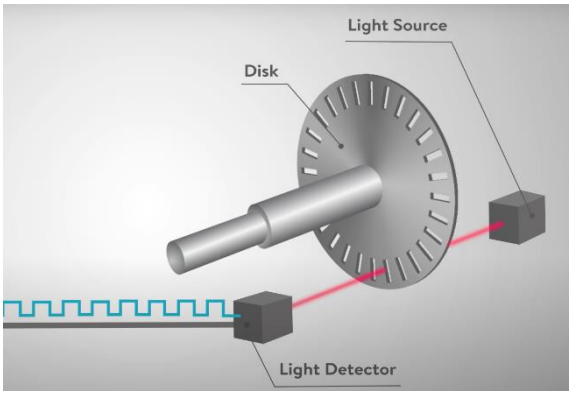
The receiver is positioned opposite the transmitter and receives the light beams. It typically contains a light-sensitive component, such as a photodiode or phototransistor, that detects the presence or absence of the light beams.





As the disk rotates, the holes on its surface pass between the light source and the light detector. When a slot passes through, the light detector receives the light, and when a solid portion of the wheel passes through, the light is blocked.

The light detector converts the received light into an electrical signal. When a slot passes through, the detector registers a high signal level, and when a solid portion passes through, it registers a low signal level.

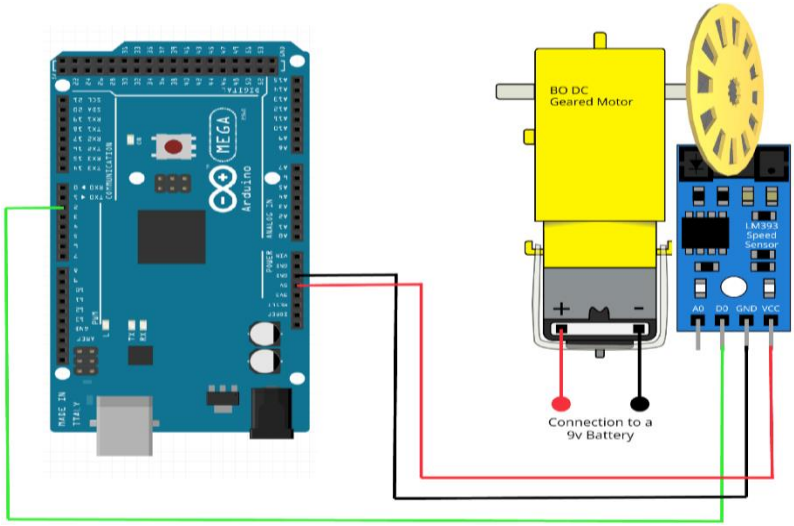


The electrical signal is processed by the sensor's circuitry to determine

the time between high and low signal levels. By measuring the time it takes for a known number of holes to pass through the sensor, the sensor calculates the rotational speed of the disk. The speed is expressed in revolutions per minute (RPM) or pulses per second.

RPM = Frequency / No. of Pulses Per Revolution * 60;

Circuit Diagram



Sensor is connected to Pin 2 in ARDUINO MEGA.

Code

This code implements a method to measure the speed of a rotating disk using a sensor or encoder that generates pulses. The time between pulses is used to calculate the RPM and averaging is applied for stability.

Variable definitions to be used in code.

```
#include <Arduino.h>

const byte PulsesPerRevolution = 2;
const unsigned long ZeroTimeout = 100000;
const byte numReadings = 2;

volatile unsigned long LastTimeWeMeasured;
volatile unsigned long PeriodBetweenPulses = ZeroTimeout + 1000;
volatile unsigned long PeriodAverage = ZeroTimeout + 1000;
unsigned long FrequencyRaw;
unsigned long FrequencyReal;
unsigned long RPM = 0;
int PulseCounter = 1;
unsigned long PeriodSum;

unsigned long LastTimeCycleMeasure = LastTimeWeMeasured;
unsigned long CurrentMicros = micros();
int AmountOfReadings = 1;
int ZeroDebouncingExtra;
unsigned long readings[numReadings];
unsigned long readIndex;
unsigned long total;
unsigned long average;
```

```
void pollSpeed() {
    LastTimeCycleMeasure = LastTimeWeMeasured;
    CurrentMicros = micros();
    if (CurrentMicros < LastTimeCycleMeasure) {
        LastTimeCycleMeasure = CurrentMicros;
    }

    FrequencyRaw = 10000000000 / PeriodAverage;

    if (PeriodBetweenPulses > ZeroTimeout - ZeroDebouncingExtra || CurrentMicros - LastTimeCycleMeasure > ZeroTimeout - ZeroDebouncingExtra) {
        FrequencyRaw = 0;  // ((++RateCounter) % (F_CPUSet frequency as 0.
        ZeroDebouncingExtra = 2000;
    } else {
        ZeroDebouncingExtra = 0;
    }
    FrequencyReal = FrequencyRaw / 10000;

    RPM = FrequencyRaw / PulsesPerRevolution * 60;
    RPM = RPM / 10000;
    total = total - readings[readIndex];
    readings[readIndex] = RPM;
    total = total + readings[readIndex];
    readIndex = readIndex + 1;

    if (readIndex >= numReadings) {
        readIndex = 0;
    }
    average = total / numReadings;

    // Serial.print("\tRPM: ");
    // Serial.println(RPM);
}
```

pollSpeed() function calculates the RPM of a rotating disk based on the time between pulses. It also maintains an array of RPM readings and calculates the average RPM over a specified number of readings.

It assigns the value of LastTimeWeMeasured to LastTimeCycleMeasure (It stores the previous time the pulse was measured), it retrieves the current time in microseconds using the micros() function and assigns it to CurrentMicros.

Then checks if the current time (CurrentMicros) is less than the last time (LastTimeCycleMeasure), If it is true, it means the microcontroller's timer has rolled over, and we need to update LastTimeCycleMeasure to the current time to prevent incorrect calculations.

Then calculates the raw frequency by dividing a large constant value by the average time between pulses, it gives the frequency in pulses per second.

Then checks if the time between pulses is greater than the allowed timeout minus the debouncing extra time. It also checks if the time since the last pulse is greater than the allowed timeout minus the debouncing extra time. If either condition is true, it means there is a timeout or a debouncing issue, ,the frequency is set to 0 and the ZeroDebouncingExtra variable is updated.

If the conditions are false, ZeroDebouncingExtra is set to 0, indicating that there is no debouncing issue.

Then calculates the revolutions per minute (RPM) by dividing the raw frequency (FrequencyRaw) by the number of pulses per revolution (PulsesPerRevolution). Multiplying by 60 converts the frequency to RPM.

Then subtracts the value at the current readIndex from the total. It prepares to update the readings array with the new RPM value and ensure that the sum reflects the latest set of RPM measurements.

Then updates the value at the current readIndex in the readings array with the calculated RPM.

Then adds the updated value at the current readIndex to the total and increments the readIndex by 1 to move to the next position in the readings array.

Then checks if readIndex has reached or exceeded the maximum number of readings (numReadings). If true, it means we have reached the end of the array and need to reset readIndex back to 0 to start overwriting the oldest readings.

Then calculates the average RPM by dividing the total sum of RPM readings (total) by the number of readings (numReadings).

```
58  void pulseEvent() {
59      PeriodBetweenPulses = micros() - LastTimeWeMeasured;
60      LastTimeWeMeasured = micros();
61      if (PulseCounter >= AmountOfReadings) {
62          PeriodAverage = PeriodSum / AmountOfReadings;
63          PulseCounter = 1;
64          PeriodSum = PeriodBetweenPulses;
65
66          int RemapedAmountOfReadings = map(PeriodBetweenPulses, 40000, 5000, 1, 10);
67          RemapedAmountOfReadings = constrain(RemapedAmountOfReadings, 1, 10);
68          AmountOfReadings = RemapedAmountOfReadings;
69      } else {
70          PulseCounter++;
71          PeriodSum = PeriodSum + PeriodBetweenPulses;
72      }
73  }
```

The pulseEvent() function is an interrupt service routine (ISR) triggered when a pulse is detected. It calculates the time period between pulses, and adjusts the number of readings and averaged time period.

It calculates the time period between consecutive pulses by subtracting the previous measured time (LastTimeWeMeasured) from the current time (micros()).

Then updates LastTimeWeMeasured with the current time (micros()).

Then checks if the PulseCounter has reached or exceeded the desired number of readings (AmountOfReadings). If true, it means enough pulses have been counted to calculate the averaged time period and adjust the number of readings.

Then calculates the averaged time period between pulses by dividing the sum of time periods (PeriodSum) by the number of readings (AmountOfReadings), resets the PulseCounter to 1(start of a new set of readings).

Then initializes PeriodSum with the current time period between pulses (PeriodBetweenPulses) since it is the first reading in a new set.

Then maps the PeriodBetweenPulses value from a range of 40000 to 5000 to a new range of 1 to 10. The mapping function allows the number of readings to be adjusted dynamically based on the time period between pulses.

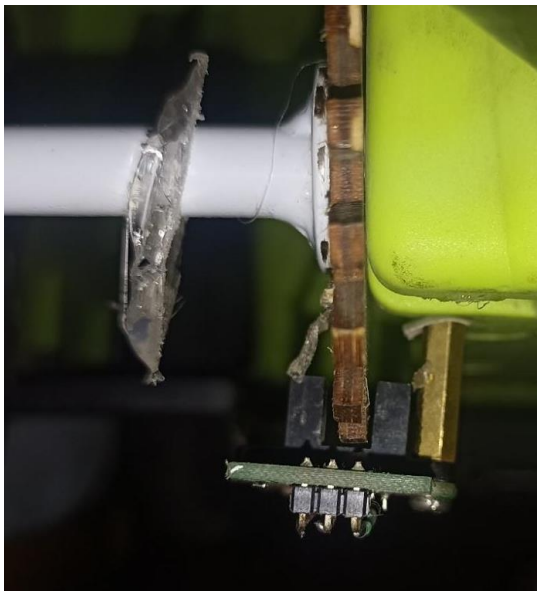
The purpose of this remapping is to dynamically adjust the number of readings used for averaging based on the measured time period between pulses.

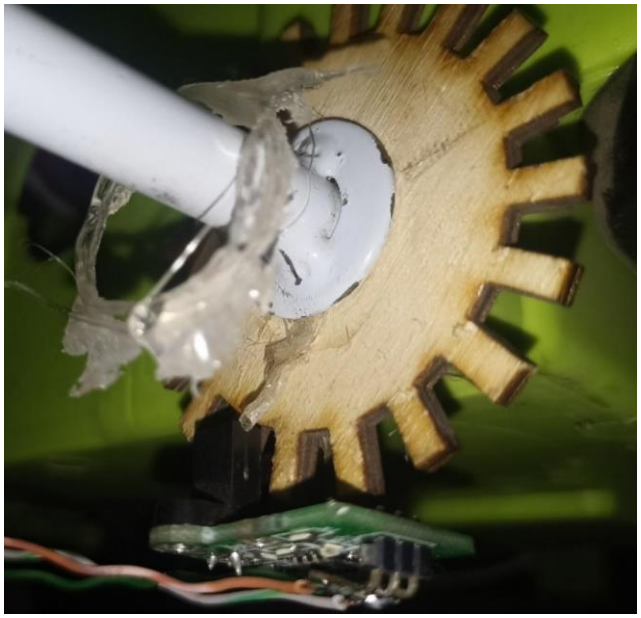
Then ensures that the mapped RemapedAmountOfReadings value is constrained within the range of 1 to 10. It prevents the number of readings from going below 1 or exceeding 10.

Then updates the AmountOfReadings variable with the adjusted value obtained from the mapping and constraining process.

If false, increments counter by 1 and add the current time period to PeriodSum.

Practical Photos





Conclusion

Speed is a fundamental concept that plays a crucial role in the automotive industry. It impacts performance, safety, efficiency, and regulatory aspects of vehicles. Understanding and managing speed effectively are essential for both drivers and automotive engineers to achieve optimal performance and safety on the roads.

Module integration & Inter-node communication

Introduction

As Advanced Driver Assistance Systems (ADAS) become increasingly complex, effective communication between system components is crucial. ROS (Robot Operating System) has emerged as a popular choice for inter-node communication in distributed robotic systems. With its libraries, tools, and standardized message-passing mechanisms, ROS enables seamless data exchange and synchronization between nodes, facilitating the implementation of robust distributed functionalities. Through this exploration, we seek to highlight the significance of ROS as a foundational software framework for inter-node communication in distributed robotic systems, exploring its relevance and applicability in the context of ADAS, and detail its application in our project.

Design Requirements

The design of an ADAS software framework states the following requirements:

A. Sensor Integration

The ADAS software framework should be capable of integrating and interfacing with various sensors such as cameras, lidars, radars, and ultrasonic sensors, enabling data fusion from multiple sensors.

B. Data Processing

The ADAS software framework should efficiently process sensor data in real-time to perform tasks like object detection, tracking, and path planning, meeting the computational requirements of ADAS algorithms.

C. Perception Algorithms

The ADAS software framework should support the development and integration of perception algorithms, including object detection, lane detection, traffic sign recognition, and pedestrian detection.

D. Communication and Networking

The ADAS software framework should enable seamless communication and networking between different components, such as sensors, actuators, and decision-making modules, facilitating coordination.

E. Simulations and Testing

The ADAS software framework should provide tools to create virtual environments for testing and validating ADAS components, reducing the cost and risks associated with real-world testing.

F. Modularity and Reusability

The ADAS software framework should follow a modular approach, allowing the creation of independent software modules that can be reused and combined efficiently to build ADAS systems.

G. Ecosystem and Community

The ADAS software framework should benefit from an active community and ecosystem of developers, researchers, and companies, enabling access to various externally maintained ADAS-related packages, libraries, and tools.

These requirements represent general considerations for designing an ADAS development environment, independent of any specific framework or platform.

Distributed Computing Frameworks – ROS

From the stated design requirements, a distributed computing framework emerges as an effective solution for creating an ADAS software framework. One noteworthy framework in this context is ROS. ROS, or Robot Operating System, is an open-source framework supporting the development of complex, modular systems in a distributed computing environment. It provides a collection of tools, and conventions that enable the creation and management of software for robots. Despite its name, ROS is not an operating system, but rather an abstraction layer that runs on top of an existing Linux operating system. The performance critical parts of the framework are written in C++, and applications operating on top of the framework may be written in C++, Python, Lisp, NodeJS, Dart, and other languages.

Key Features

Message Passing: ROS facilitates communication between different software components of a robot using a publish-subscribe messaging system. Nodes, which are modular software units, can be deployed on separate computers or microcontrollers, and publish messages to topics or subscribe to topics to exchange information with each other in a network.

Package Management: ROS uses a package-based architecture, allowing developers to organize their code and dependencies into shareable, extensible units. This modular approach promotes code reuse and simplifies software development.

Hardware Abstraction: ROS provides hardware abstraction by offering a standard interface to interact with different devices and sensors. This enables developers to write hardware-agnostic code that can work with various system configurations.

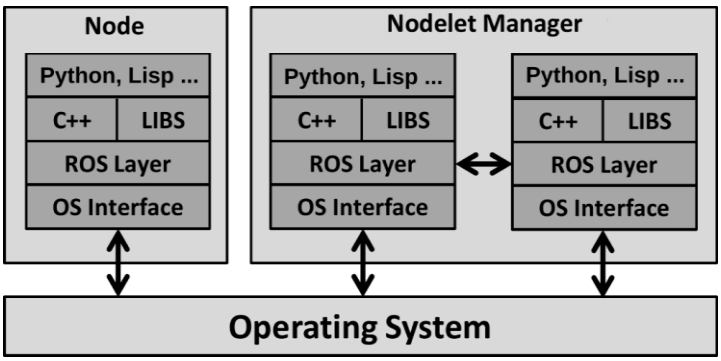
Tooling and Ecosystem: ROS offers a rich set of tools and libraries to support various aspects of robot development. These include visualization tools, simulation environments, debugging utilities, motion planning libraries, and more.

Community: ROS has a vibrant and active community of developers who contribute to the ecosystem by developing and sharing packages, tutorials, and resources. This collaborative nature fosters knowledge exchange and accelerates the development process.

System Architecture

The ROS framework is a multi-server environment allowing software applications to act as one software system by communicating across server boundaries. One server in the system is dedicated as ROSMaster, making it responsible for registering applications (nodes) as well as running the central parameter storage, and the message broker server. Satellite servers (slaves) are connected to the master server over a computer network via TCP or UDP protocols, these servers can be used to load-balance the system, or can each have a different functionality according to its specialized hardware configuration.

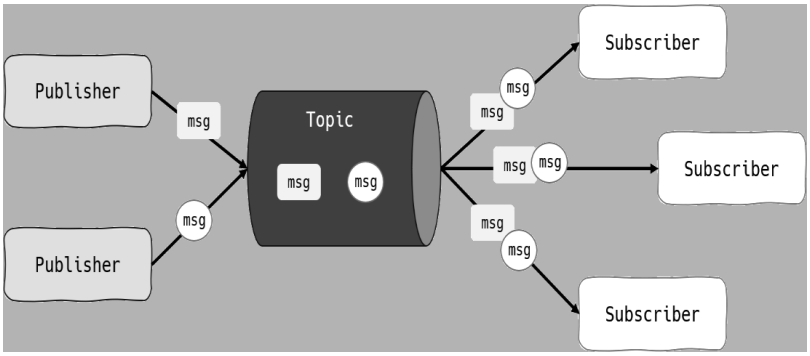
Applications in ROS are called nodes or nodelets, the distinction between them is that each node is mapped to a process, while nodelets are architected to work within a single process, called a nodelet manager, for purposes of low overhead resource sharing. The ROS application architecture is visualized by the opposite figure.



Asynchronous messaging is implemented inside ROS following the publish-subscribe pattern as shown in the figure below. Therefore, a node generating information defines a publisher and registers this publisher with the

ROSMaster, while a node requiring information instantiates and registers a subscriber. The registered publishers and subscribers are associated with each other by unique topic names.

The information exchanged through ROS is declared in a ROS-specific format (ROS msg), that supports primitive data types like booleans, strings, integers, and floats, specialized types such as Image, Compressed_Image, and Path, as well as type composition, where users build custom-types



from the provided standard types. To support message exchange across programming languages and CPU architectures, the declared messages are compiled into native classes (binary format) that gets serialized and deserialized automatically by ROS during transmission in inter-node communication. This overhead is omitted in the intra-process communication of nodelets.

Performance

ROS by itself is not a real-time capable system due to missing priority-enforced message passing, and node execution time guarantees. To get a real-time capable ROS environment, the claimed strategy is to deploy low-level embedded circuits that meet the real-time constraints and integrate them with ROS.

Testability

The modular messaging structure of ROS allows for seamless interchange of incoming data sources, this allows nodes to be easily tested with generated (mocked), pre-recorded, or simulated data. ROS also integrates with a full robot system simulation framework powered by a 3D graphics engine with full support for dynamic and kinematic physics. ROS also provides integrated infrastructure for sending diagnostic messages to a central diagnostics server that displays the system health in a traffic light color scheme to immediately determine if data integrity is at risk.

Problem Statement

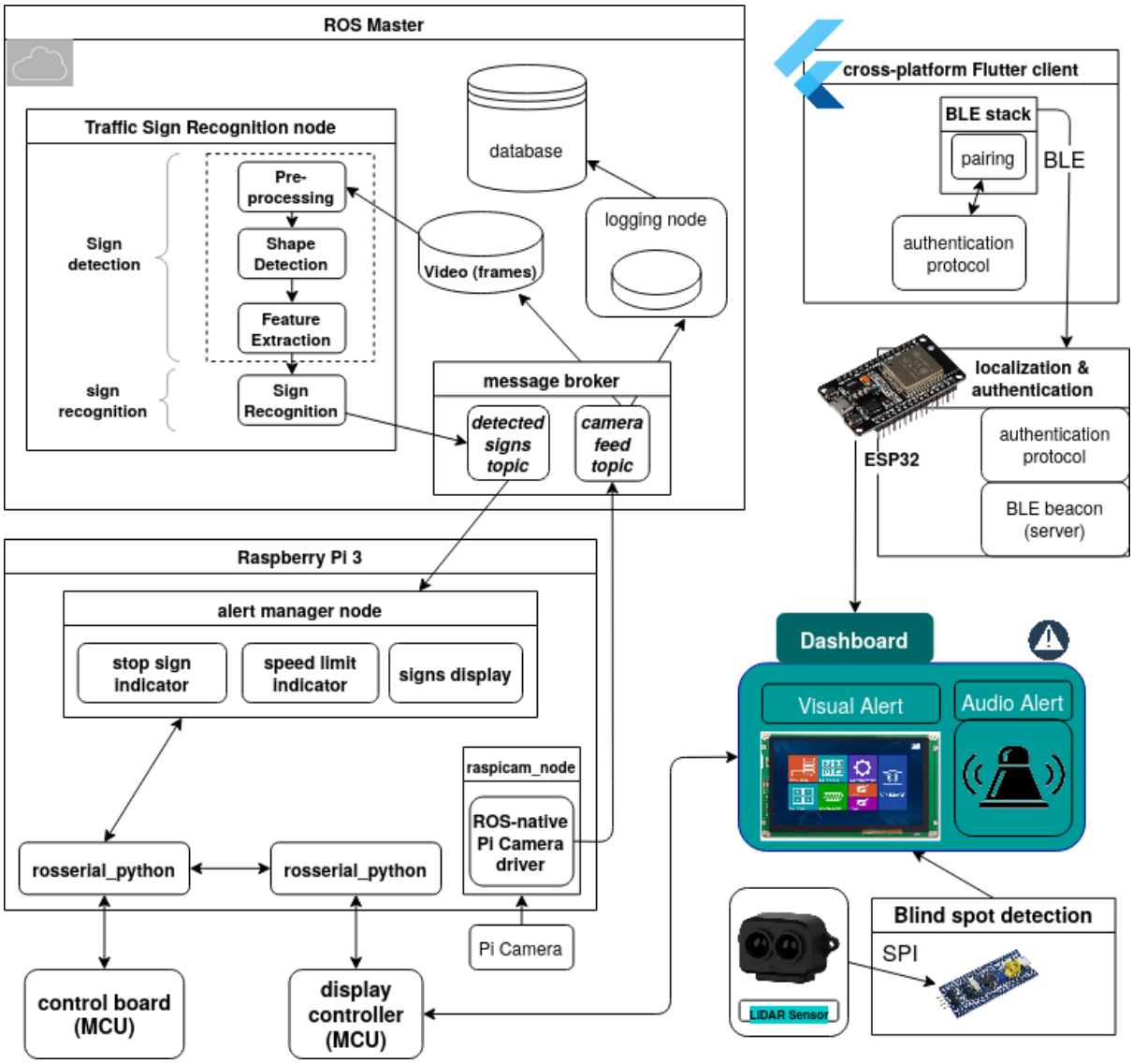
Our project consists of multiple software modules each of which is deployed to a separate compute node, some of which requires asynchronous communication with other processes in order to operate correctly, such as:

- Traffic sign detection AI: requires a powerful computer separate from the vehicle
- Integrated in-vehicle camera
- Vehicle control board
- System display module

Proposed Architecture

Given the project at hand, we decided on employing two computers and two microcontrollers in implementing our ADAS system, where one computer acts as the ROS master server and has the AI node on it, while the other is a Raspberry Pi 3 integrated into the vehicle and controls other embedded peripherals using ROS technologies. The microcontrollers are connected to the distributed system using rosserial, which is a ROS module that acts as a master server that communicates with a device over UART and creates nodes and topics, as well as subscribe to and publish messages on behalf of the microcontroller, then updating its state as if it was directly inside the ROS network. For the camera, we used raspicam_node, which is a ROS community project known for being the fastest Raspberry Pi Camera solution for ROS due to being a full camera driver designed specifically as a ROS node.

Architecture Diagram



Deployment methodology

Running and exploring ROS (on the master server)

Since ROS Foxy (ROS2) dropped support for rosserial 8-bit microcontrollers altogether, we picked ROS Noetic, which is the ROS distribution released immediately before Foxy, for our project. Noetic is still a widely used and well supported version of ROS, since the ROS maintainers community provides long-term support (LTS) for it, meaning it continues to receive bug fixes and security updates ensuring it remains stable and reliable. Since the ROS ecosystem is deeply integrated into Linux OSes, each distribution of ROS has very specific requirements of the Linux system it's installed on. ROS Noetic is primarily designed and officially supported for Ubuntu 20.04 LTS, with software dependency on GCC-9 as well as both Python 2 and 3. These requirements are too specific for a regular general purpose Linux computer in 2023, which posed the problem of being an impractical Linux platform for a modern development environment. We tested the practicality of two possible solutions for our use case, which are virtualization and containerization.

Containerization is a method of packaging an application, along with its dependencies and configuration files into a standard unit known as a container. This container provides an isolated lightweight environment for the application to run consistently across different environments. By encapsulating the application and its dependencies, containerization enables easier deployment, and management of software complex applications. Containerization requires a container runtime which is a thin wrapper around the host system's kernel that operates the container and manages its permissions, resources, namespace and processes.

Virtualization is another approach for creating isolated environments for software applications. It involves the abstraction of the underlying hardware resources, allowing multiple virtual machines (VMs) to run concurrently on a single physical machine. Each VM operates as a self-contained entity, complete with its own full operating system, applications, and dependencies. Virtualization provides flexibility in running different operating systems and software versions on the same hardware, enabling compatibility across more diverse environments. The core concept behind virtualization is the hypervisor or virtual machine monitor (VMM), which is the software layer that manages the virtualization process, facilitating the creation, execution, and control of VMs, along with allocating system resources to each instance.

Our attempt at containerizing ROS along with our application was proven unfruitful due to the software accelerated networking of container runtimes which turned out inefficient in testing, especially in cases where the network was receiving a high influx of traffic targeting a device no longer resolvable on the network. So to combat this problem we employed the powerful virtualization capabilities of Linux systems to dedicate a hardware network interface card to an Ubuntu 20.04 virtual machine connecting it physically (on the wire) with the rest of the network. This approach leverages KVM (Linux kernel-based virtual machines) as a type 1 (baremetal) hypervisor for deploying an Ubuntu 20.04 instance on a Linux host. KVM is integrated directly into the Linux kernel, which allows it to take advantage of the kernel's capabilities, including process and memory management, scheduling, and device driver support to provide an efficient virtualization solution that offers near-native performance for virtualized workloads. Another tool we employed was libvirtd which is the API we picked for managing the virtual machine and its resources. Libvirtd enabled the physical passthrough of the NIC to the ROS VM.

ROS on the Pi

In our project, it was essential to have hardware-accelerated video support in order to drive the Raspberry Pi Camera efficiently and achieve optimal performance with raspicam_node, which is the

specialized package designed to interface with the Raspberry Pi Camera Module, allowing for the efficient capture of high-quality images and videos within the ROS framework. Hardware acceleration utilizes the GPU's specialized video processing capabilities in the Raspberry Pi to relieve the CPU of computationally demanding tasks, such as image encoding and decoding, which greatly enhances the efficiency of video-related operations, one of which is operating the camera module. Since the Linux kernel included in Ubuntu 20.04 for the Raspberry Pi 3 (which is based on the armv8 64-bit platform) doesn't support hardware acceleration, we had to use the officially supported Raspberry Pi OS edition corresponding to Ubuntu 20.04, which is Raspbian Buster. In order to run ROS Noetic on Raspbian Buster, a manual process involving building ROS Noetic from source using tools CMake and Catkin was required. This approach was necessary because prebuilt packages for ROS Noetic and its external dependencies, such as matplotlib and numpy, were not available in the Debian APT (Advanced Packaging Tool) and PyPI (Python Package Index) repositories specifically compiled for the armv7 architecture.

Building ROS Noetic from source involved several steps. First, the necessary dependencies, including Python packages, had to be installed and built manually. Then, once the dependencies were in place, the ROS Noetic source code was obtained using rosinstall_generator while community packages such as raspicam_node and rosserial were fetched by git, then the source code was built using Catkin, which is a popular cross-platform build system based on CMake. rosinstall_generator generated the necessary build scripts and cmake_lists based on the project's configuration. Then, Catkin was used to compile and link the ROS packages, creating the necessary executables and libraries. Building ROS Noetic from source allowed for a customized and tailored installation specifically for the Raspbian Buster environment. However, it required careful management of dependencies, configurations, and build processes to ensure a successful build. This manual process demanded familiarity with build systems. and the specific intricacies of Linux and ROS package management.

After building and running ROS Noetic on the Raspberry Pi 3 from source, the building process was documented and a bash script was written to fully automate the process in order to be used in other deployment quickly.

Embedded ROS?

rosserial_arduino is a package in ROS (Robot Operating System) that enables communication between a microcontroller running the Arduino framework and a ROS system. It allows developers to interface their Arduino-based hardware with the ROS ecosystem, enabling bidirectional communication and synchronization between Arduinos and other ROS nodes.

At its core, rosserial_arduino provides a lightweight communication protocol that enables the transmission of ROS messages between the Arduino and a computer running ROS. It achieves this by establishing a serial communication link between the Arduino board and the host computer, via USB or TCP.

The functionality of rosserial_arduino can be broken down into two main components.

Arduino Library

The rosserial_arduino library, which needs to be uploaded onto the Arduino board, provides the necessary functionalities to establish the serial connection and handle the communication with the ROS system. It handles the serialization and deserialization of ROS messages, allowing the Arduino to send and receive data in the ROS msg format.

The library also includes utilities for subscribing to and publishing ROS topics, calling ROS services, and managing parameters. It provides a familiar C++ API for interacting with a ROS system from the Arduino board, employing the same ergonomics of regular ROS nodes written in C++.

ROS Node

On the ROS side, rosseria_pythonl provides a ROS node that runs on the host computer and acts as a bridge between the Arduino and the ROS network. It establishes a serial communication connection with the Arduino board and handles passing ROS messages to and from the Arduino. The rosserial_python node publishes and subscribes to ROS topics, allowing data to be exchanged between the Arduino and other ROS nodes. It also handles ROS service calls and parameter interactions, enabling bidirectional communication and synchronization with the Arduino-based hardware.

Testing

During the development of the deployment, running the full AI model imposed a burden on the speed and efficiency of testing, so instead we wrote a mocked version of the traffic sign detection model that exactly matches the API of Ultralytics YOLO, which is the framework used to create and run the model. The mocking code can work interchangeably with the traffic sign detection node, where it runs as a ROS node, creates the detected signs topic, subscribes to the raspicam_node's topic and waits for messages of images. When the mocked node successfully receives an image, it generates a list of random length containing a number of randomly picked traffic sign classifications and sends the list over its topic, where other nodes consume it exactly how they consume messages from the real model. This mocking environment gave the ability to both test other nodes on the network without worrying about the functionality and correctness of the AI model as well as test the entire system on edge cases where the vehicle sees a lot of signs at once, or receives successive new detections too quickly. It also facilitated testing the system without needing to physically move traffic signs in and out the view of the camera, stopping it from slowing down the process. ROS's abstraction of communication made it easy to mock the presence of a module which in turn facilitated getting all other modules operating and communicating faster.

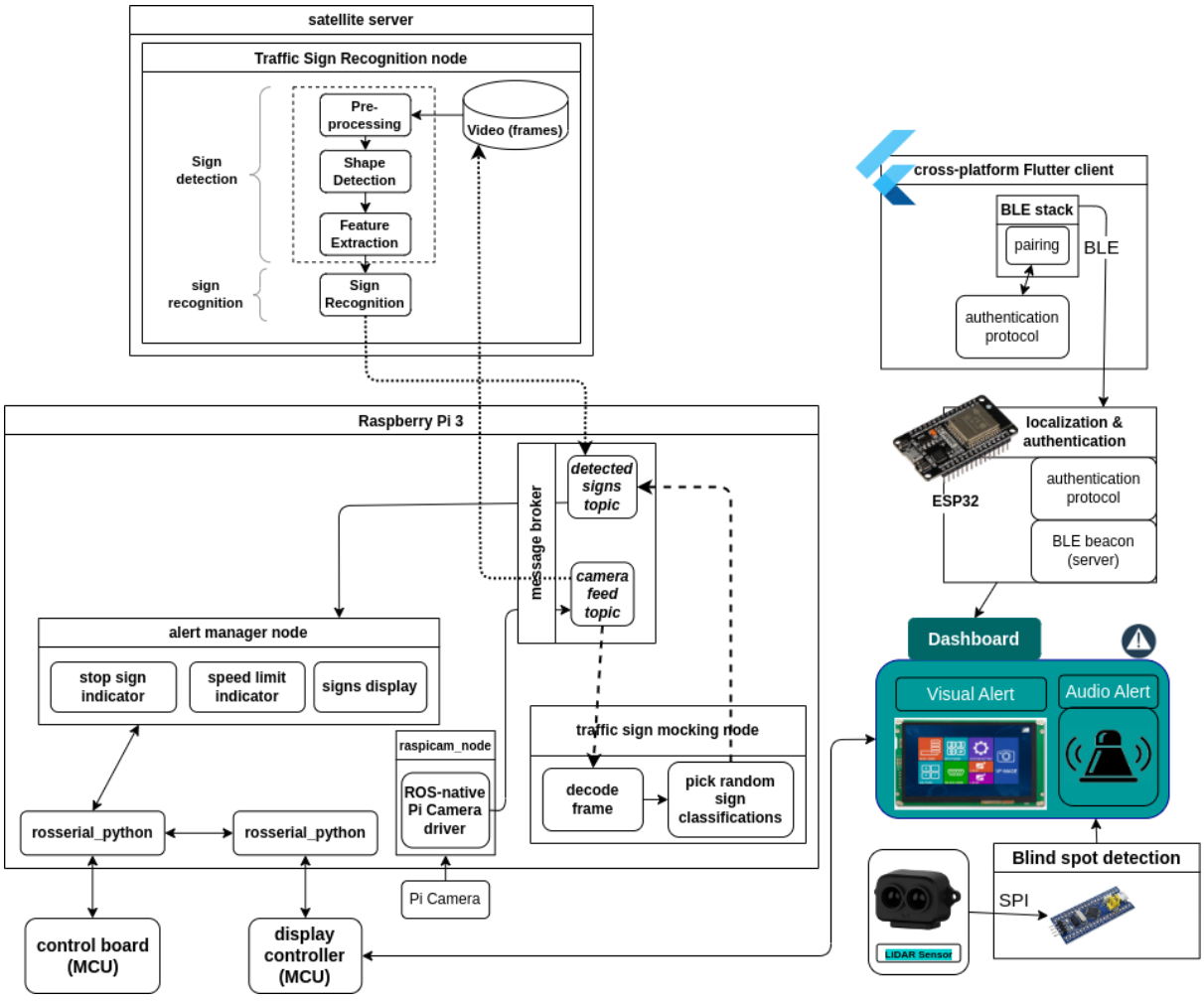
Operation methodology

In order to enable communication between the software components deployed on ROS, we wrote custom code specifically to insert the component outputs into messages following the ROS msg format. Due to the distributed nature of the system, manually starting up all the required nodes correctly and in order is a complex feat which requires programmatically launching the nodes and validating the configuration along the way. This is facilitated by another ros tool, which is roslaunch. roslaunch is a command-line program that consumes declarative configuration files that detail the launch process for a ROS system. These configuration files, typically with a .launch extension, provide a convenient way to specify which nodes should be started, their parameters, and any additional arguments or conditions for their execution. A launch file describes the desired state of ROS nodes, and roslaunch automatically resolves their operation dependencies, getting them up and running in the desired state. Other liveness checks were also written in bash to validate the availability of the system's nodes after launch, as well as utility functions to administer the nodes and quickly examine their outputs. Then after getting the full startup setup in place and making sure it's robust and fully functional, the Raspberry Pi 3 was configured to automatically launch its nodes on startup, since it's required to operate non-interactively upon getting powered on.

Fully integrated environment

We explored the possibility of taking a cloud native approach to deploying our project, which is why the server with the AI model was the one assigned as ROS master, where this server runs the main processes of the system and end-users connect to it externally. While this approach was employed successfully and worked well as a proof of concept, due to limitations of cost and internet reliability, we decided on changing the server assigned as master to the Raspberry Pi 3 to simplify the communication between the nodes, since most of them run on the Pi and won't route their messages through the network. This resulted in the Pi, along with the traffic sign mocking node, being a fully integrated development environment for testing and operating the system, as well as creating new nodes that consume the traffic signs detected.

Conclusion



The utilization of ROS as the communication system for deploying the ADAS, specifically for traffic sign detection and inter-microcontroller communication, has proven to be highly effective. ROS facilitated seamless data exchange and synchronization between the various software components, enabling efficient integration of sensors, data processing, perception algorithms, communication, and testing functionalities.

By leveraging ROS's message passing mechanisms, package management system, hardware abstraction, and extensive tooling and ecosystem, we were able to meet the design requirements of

the ADAS software framework. ROS provided a modular and reusable approach, allowing the creation of independent software modules that could be easily integrated and combined to build a robust ADAS system.

The active ROS community played a significant role in supporting the project, providing access to externally maintained packages, libraries, and tools. The availability of simulation environments and testing tools within the ROS ecosystem reduced the cost and risks associated with real-world testing, enhancing the overall development process.

Furthermore, the flexibility of ROS allowed for the deployment of the ADAS system on different hardware platforms. The Raspberry Pi, integrated with ROS, served as an efficient platform for controlling embedded peripherals, such as the Raspberry Pi Camera Module, and the dashboard and controller Arduinos.

Through the use of ROS, we successfully addressed the challenges of inter-node communication, testing, and deployment in our ADAS project. The modular and scalable nature of ROS, combined with its extensive capabilities and active community support, made it a suitable and effective solution for our proposed problem. ROS has demonstrated its significance as a foundational software framework for inter-node communication in distributed robotic systems, including ADAS applications. Its adoption has significantly contributed to the success and efficiency of our project, paving the way for future advancements in the field of ADAS development.

Car Display System

Introduction:

Its a short cut for Thin-Film Transistor display (TFT), its a type of Liquid Crystal DisplaY (LCD) technology which is commonly used in various electronics devices like Smartphones, Computer monitors, Television, etc. TFT displays several advantages over its predecessor LCD technologies, including improved image quality, faster response times, and better colour reproduction.

TFT displays have a significant role in industries such as consumer electronics,medical devices and automotive.

- In consumer electronics, TFT displays can be found in smartphones, tablets, laptops, and TVs, delivering sharp visuals and enabling interactive touch functionality.
- In medical devices, TFT displays are used in ultrasound machines, patient monitors, and medical imaging sys., providing accurate visualisation of critical data and diagnostic images.
- In the automotive industry, TFT displays are used in car infotainment sys., instrument clusters and head-up displays, providing drivers with fundamental info. And improving the driving experience.

Some of TFT advantages:

1. Thin-Film Transistor (TFT): its a type of transistors which used to control every individual pixel on LCD display, Each pixel has its own transistor, allowing for precise control of the pixels' brightness and its colour.

2. Colour Reproduction: TFT display has a very wide range of colour pallets which makes them suitable for the application where accurate colour representation is important, such as multimedia devices or even in professional monitors.

3. Viewing Angels: TFT display offers a wider viewing angle compared to its predecessor LCD technologies, ensuring remaining image quality consistent however it is viewed from other different angles.

4. Active Matrix: TFT display sometimes is referred to as an active matrix because every individual transistor provides an active control; which allows the TFT to have a faster refresh rate and better image quality compared to passive matrix displays.

5. Energy Efficiency: TFT display is genuinely more energy-efficient than other older LCD technologies, as the individual control in every pixel helps optimising power consumptions.

6. Response Time: TFT display has a super fast response times than its predecessor LCD technologies, which from its advantages reduce motion blur and also improves the overall visual experience, especially in content like gaming or video playback that needs a very high fast-paced.

7. Touchscreen Integration: TFT displays can be integrated with touchscreen technology, in which the user can interact through taps, swipes and scrolling; that is widely used in smartphones, tablets, some laptops, and other devices which require intuitive user interfaces.

8. Flexibility and Customization: TFT displays can be manufactured in a lot of different sizes, shapes, and resolutions to fit specific application requirements. This flexibility allows for customization and integration into different form factors and designs.

TFT displays have become a basic standard for millions of electronic devices due to their very improved image quality, super fast response times, and energy efficiency. They continue to evolve with advancements in technology, offering even better performance and features in modern devices and machines.

Importance of Monitoring in a car assistant system

1. Safety: Monitoring allows the system to detect and respond to potential safety hazards. For example, if the system monitors the driver's attention level and detects drowsiness or distraction, it can warn the driver or take control of the vehicle to prevent accidents.

2. Performance optimization: Monitoring helps in maximising the performance of the car assistant system. By constantly monitoring the system's operations, it can detect any deviations or malfunctions, ensuring that the system operates at its optimal capacity.

3. Real-time feedback: Monitoring provides real-time feedback to the driver or passengers regarding various parameters such as speed, fuel consumption, temperature, and battery charge. This feedback helps the driver make informed decisions and adjust their driving behaviour accordingly.

4. Predictive maintenance: By continuously monitoring various components of the car, the assistant system can detect any potential failures or issues before they cause a breakdown. This allows for timely maintenance or repair, reducing the risk of sudden breakdowns and increasing the overall reliability of the vehicle.

5. Data collection and analysis: Monitoring allows the car assistant system to collect and analyze data related to driving patterns, vehicle performance, and overall road conditions. This data can be used to optimize the system's algorithms and improve future versions of the car assistant system.

Overall, monitoring in a car assistant system is crucial as it ensures safety, performance optimization, real-time feedback, predictive maintenance, and data analysis, resulting in a better driving experience for the users.

TFT in assistant driving cars

TFT displays have a fundamental role in assistant driving cars by providing critical information and improving the overall driving experience.

In our module we used TFT display to display:

1. Sign.
2. Speed Limits.
3. Speed.
4. Sound System currently playing audio.

---

1. Sign

In our module we needed the sign to be detected from the street and displayed on the TFT display to make it easy for the driver to see the sign clearly and not to get distracted from monitoring it from the road, also in case the driver missed the sign on the road it will be displayed just in front of him/her on the TFT. our code was configured following the following algorithm:

- Specify a shape to display the Sign part in.
- Grid the TFT so that the sign is displayed on the top section of the display
- The sign label is displayed on the screen in the specific place preceded by the word “sign”
- Print the data that will be received from the camera then display it on the TFT

---

2. Speed Limit

The second thing that is displayed on the TFT is the speed limit that is detected from the road by the camera as a sign and then translated as a speed limit on the TFT so the driver makes sure not to go beyond the allowed speed limit.

This code creates a ROS node handle object. The node handle is used to interact with the ROS system and manage communication with other nodes.

It defines a constant “SPEEDLIMIT_POS” taking the value 10,110.

Then creates a ROS subscriber object named “speed_limit” that executes the previous function in the code whenever it received a message that type is “std_msgs:ulnt8” which is a message type in (std_msg) package which represents an unsigned 8-bit integer value from 0-255.

It formats the speed limit value (msg.data) into a character buffer within a range formed from 3 character including leading zeros if needed that's because “snprintf” is used so that’s ensures that the speed limit is displayed consistently with three digits, even if the original value is less than 100.

```
ros::NodeHandle nh;

#define SPEEDLIMIT_POS 10, 110
ros::Subscriber<std_msgs::UInt8> speed_limit(  // initialize
    "speed_limit",                             // on the topi
    [](const std_msgs::UInt8& msg) {           // with the fo

        tft.fillRect(GREEN_RECT);
        tft.drawRect(0, 80, 240, 80, WHITE);
        tft.setCursor(10, 90);
        tft.println("S_Limit:");


        char buffer[4];
        snprintf(buffer, sizeof(buffer), "%3d", msg.data);
        tft.setTextColor(WHITE);
        tft.setTextSize(2);
        tft.setCursor(SPEEDLIMIT_POS);
        tft.println(buffer);
    });

bool stop_visible = false;
uint32_t stop_counter = 0;
#define STOP_DISP_TIME(x) \
    stop_counter = (x * (F_CPU / 500))
```

It identifies a boolean variable ”stop_visible” and initialises it to “false” value. This variable is used to track the visibility of the stop sign on the display.

It identifies an unsigned 32-bit integer variable “stop_counter” and initialises it to Zero. This variable is used to keep track of the time for which the stop sign should be displayed.

Then it defines a macro “STOP_DISP_TIME(x)” that calculates the stop counter value based on the desired display time “x”. It multiplies the desired time by the clock frequency (F_CPU) divided by 500.

For all other code lines it's the same as the (Sign) code but with different values.

Those lines of code handle the display of the speed limit value and the management of the stop sign display on the TFT display. The speed limit value is displayed in a designated area on the display, and the stop sign can be shown for a specific duration based on the value set for “stop_counte”.

---

3. Speed

For the next feature shown in the TFT display is Speed, It displays the current speed that the driver going on with just to make sure not to go beyond the speed limit as the both of (Speed, Speed limit) is shown in the same TFT so it's gonna be easy for the driver to keep track in what speed he/she is allowed to go on with.

```
#define SPEED_POS 10, 190
ros::Subscriber<std_msgs::UInt8> current_speed(
    "current_speed",
    [](const std_msgs::UInt8& msg) {

        tft.fillRect(BLACK_RECT);
        tft.drawRect(0, 160, 240, 80, WHITE);
        tft.setCursor(10, 170);
        tft.println("Speed:");


        char buffer[4];
        snprintf(buffer, sizeof(buffer), "%3d", msg.data);
        tft.setTextColor(WHITE);
        tft.setTextSize(2);
        tft.setCursor(SPEED_POS);
        tft.println(buffer);
    });
```

This code defines a constant “SPEED_POS” with the value (10, 190).
Then creates a ROS subscriber object named “current_speed” that listens to the topic "current_speed" and executes the function whenever a message of type “std_msgs::UInt8” is received. The received current speed value is stored in the “msg.data” variable.

For all other code lines it's the same as the (Sign) code but with different values.

By executing the previous lines of code whenever a current speed message is received, the TFT display will be updated with the latest current speed, showing it on the screen.

---

4. Sound System State

The last feature shown in the TFT is sound currently played on the sound system so the driver can have an easy access to his/her playlist and can change it easily using the Sound system buttons therefore the audio name is gonna be changed on the TFT display screen.

```
#define SONG_POS 10, 270
ros::Subscriber<std_msgs::String> now_playing(
    "now_playing",
    [](const std_msgs::String& msg) {

        tft.fillRect(BLUE_RECT);
        tft.drawRect(0, 240, 240, 80, WHITE);
        tft.setCursor(10, 250);
        tft.println("Now Playing:");

        digitalWrite(LED_BUILTIN, LOW);
        tft.setTextColor(WHITE);  // Set Text P
        tft.setTextSize(2);
        tft.setCursor(SONG_POS);
        tft.println(msg.data);
    });
```

This code defines a constant “SONG_POS” with the value (10, 270).
Then creates a ROS subscriber object named “now_playing” that listens to the topic "now_playing" and executes the previous function whenever a message of type (std_msgs::String)is received on that topic. The received song name is stored in the “msg.data” variable.

By executing these lines of code whenever a "now_playing" message is received, the TFT display will be updated with the currently playing audio showing it on the screen.

The Arduino sketch is essential for receiving and processing messages sent by other nodes in a network made up of Raspberry Pi nodes.The data is subsequently processed in accordance with the preset logic or algorithms included in the sketch. This makes it possible for seamless coordination and communication between various network nodes, enabling effective data sharing and collaboration among the connected Raspberry Pi devices.

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

    nh.initNode();
    nh.subscribe(stop);
    nh.subscribe(speed_limit);
    nh.subscribe(other_signs);
    nh.subscribe(current_speed);
    nh.subscribe(now_playing);
}
```

pinMode(LED_BUILTIN, OUTPUT); sets the pin mode of the “LED_BUILTIN” pin to “OUTPUT”, indicating that it will be used to output digital signals.

pinMode(LED_BUILTIN, HIGH); sets the “LED_BUILTIN” pin to a “HIGH” state, turning on the LED connected to that pin, used to indicate that the system is powered on or active.

nh.initNode(); initialisation of the Raspberry Pi node, allowing the Arduino sketch to communicate with other nodes in the Raspberry Pi network.

nh.subscribe(stop); This line subscribes to the Raspberry Pi topic specified by the `stop` object. It listens for messages published on this topic and executes the corresponding callback function when a message is received.

nh.subscribe(speed_limit); This line subscribes to the Raspberry Pi topic specified by the `speed_limit` object. It listens for messages published on this topic and executes the corresponding callback function when a message is received.

nh.subscribe(other_signs); This line subscribes to the Raspberry Pi topic specified by the `other_signs` object. It listens for messages published on this topic and executes the corresponding callback function when a message is received.

nh.subscribe(current_speed); This line subscribes to the Raspberry Pi topic specified by the `current_speed` object. It listens for messages published on this topic and executes the corresponding callback function when a message is received.

nh.subscribe(now_playing); This line subscribes to the Raspberry Pi topic specified by the `now_playing` object. It listens for messages published on this topic and executes the corresponding callback function when a message is received.

The Arduino sketch can receive and process messages published by other nodes in the Raspberry Pi network. The callback functions associated with each subscription will be executed whenever a new message is received.

```
void loop() {
    nh.spinOnce();

    if (stop_visible) {
        if (--stop_counter <= 1) {  // when stop_counter reaches 0, delete the message
            tft.setCursor(STOP_POS);
            tft.println("        ");  //! replace this with a rectangle
            digitalWrite(LED_BUILTIN, LOW);
            stop_visible = false;
        } else {  // blink the LED
            if (stop_counter % 1000 == 0)
                digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
        }
    }
}
```

In this code “nh” refers to a ROS node handle. ‘spinOnce()’ is a function call that allows ROS to process any pending messages or callbacks.We check the value of the ‘stop_visible’ variable. To check

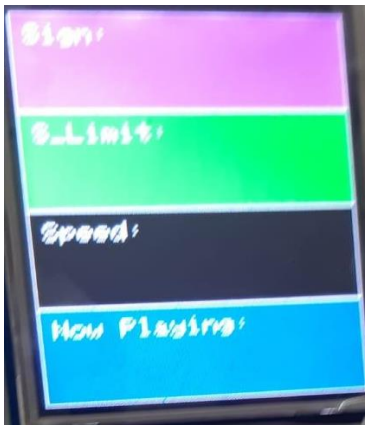
whether to execute the following or not:

–We decrement the value of ‘stop_counter’ by 1 and then check if the new value is less than or equal to 1. If it is true, the subsequent code block will be executed.:

It sets the cursor position on a TFT display to display some text then prints an empty string of spaces on the TFT display to overwrite the existing text with blank spaces. The number of spaces may vary.after that, it turns off the built-in LED. The ‘LED_BUILTIN’ refers to the pin number of the LED on the Arduino board. And Then it sets the `stop_visible` variable to false, indicating that the stop message is no longer visible.

`else {`: If the condition(‘stop_counter’ is greater than 1), this code block is executed:

-It checks if `stop_counter` is evenly divisible by 1000 then toggles the state of the built-in LED. It reads the current state of the LED pin using `digitalRead()`, negates it using the `!` (logical NOT) operator, and sets the LED state to the resulting value using `digitalWrite()`.



- When the car battery is turned on, the set up is configured so that the first thing that displays in the TFT is the following image (without any input data message)

Hardware connections

For the output

In order to make our module come true we firstly connected the TFT display to Arduino Uno.

TFT display connected to Arduino Uno.

After that, the Arduino Uno was connected to the Raspberry pi. So that the Arduino Uno can take the data from the Raspberry pi as an output data then displayed on the TFT display.

Arduino Uno connected to Raspberry pi

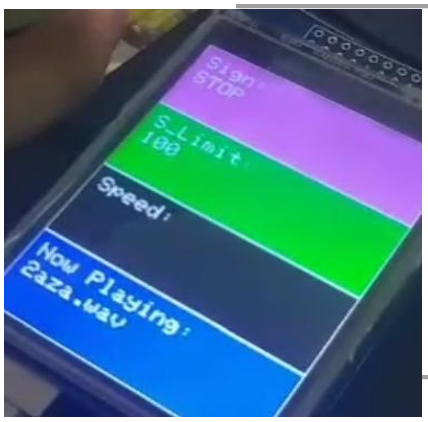
For the input

In order to have the input data/messages the input data is delivered to the Raspberry pi through:

- Arduino mega.
- Camera.

1. Arduino Mega:
   It is connected to the Car Sound System and Speed Encoder they send input data to Ros to be displayed on the TFT as “Speed” in case the input came from (Speed encoder) or as “Now Playing” in case the input is sent by (car sound system).

2. Camera:
   It's directly connected to Raspberry pi so when the camera detects any sign, it sends it to the Ros as an input data then Arduino Uno receives the data as an output so it can be translated and displayed on the TFT as “Sign”.
   In case the Sign that the camera gonna detect is in category Speeds limit the Ros will send it to the Arduino Uno as output message so the Arduino Uno can translate it to integer number then displays it on the TFT display as “Speed limits”.



After the TFT receives the input data messages from Arduino Mega (Car sound system) as “now Playing” and from the Camera as “Sign” and “Speed limit”

References

- STM32F10xxx/20xxx/21xxx/L1xxxx Cortex®-M3 programming manual
- STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced Arm®-based 32-bit MCUs - Reference manual
- ST7735S.pdf (focuslcds.com)
- SJ-PM-TF-Luna A01 Product Manual.pdf

- SJ-PM-TF-Luna-A03-Product-Manual.pdf (seeedstudio.com)
- Blind spot monitor - Wikipedia
- Blind-Spot Monitoring System using Lidar | SpringerLink
- https://www.akm.com/eu/en/products/rotation-angle-sensor/tutorial/optical-encoder/
- https://realpars.com/encoder/
- https://injuryfacts.nsc.org/motor-vehicle/motor-vehicle-safety-issues/speeding/
- https://www.news18.com/news/india/fast-and-fatal-over-speeding-kills-an-indian-every-6-minutes-injures-one-every-two-minutes-5892487.html
- pygame.joystick — pygame v2.5.0 documentation
- https://websockets.readthedocs.io/en/stable/
- https://docs.python.org/3/library/asyncio.html
- How To Build WebSocket Server And Client in NodeJS – PieSocket Blog
- https://handsontec.com/dataspecs/module/esp8266-V13.pdf
- http://www.handsontec.com/dataspecs/L298N%20Motor%20Driver.pdf
- Pehlivanoglu, Y. (2015). Arduino for Musicians. Oxford University Press
- https://en.wikipedia.org/wiki/Secure_Digital
- Proakis, J. G., & Salehi, M. (2007). Digital Communications. McGraw-Hill Education.
- Uba, M. (2017). Arduino Playground: Geeky Projects for the Curious Maker. No Starch Press.
- https://tronicspro.com/tda2030-stereo-amplifier-circuit/