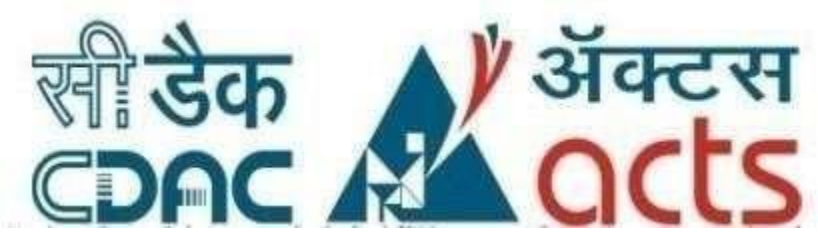
Project Report
On
CANShield: Collision Avoidance Protocol



Submitted
In partial fulfilment
For the award of the Degree of

PG-Diploma in Embedded Systems and Design (PG-DESD)

C-DAC, ACTS (Pune)

Guided By:

Mr Bhupender Pratap Singh

Submitted By:

Karan Solanki *240340130023*
Lalit Deshpande *240340130028*
Onkar Chougale *240340130031*
Prasad Kulkarni *240340130035*

Centre for Development of Advanced Computing(C-DAC), ACTS

(Pune- 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Mr Bhupender Pratap Singh.** C-DAC ACTS, Pune for her constant guidance and helpful suggestion for preparing this project CANShield: Collision Avoidance protocol**.** We express our deep gratitude towards him for inspiration, personal involvement, constructive criticism that he provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Mrs. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mrs. Risha P R (Program Head)** and **Mrs. Srujana Bhamidi** (Course Coordinator, PG-DESD) for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Karan Solanki 240340130023

Lalit Deshpande 240340130028

Onkar Chougale 240340130031

Prasad Kulkarni 240340130035

ABSTRACT

Safety has always been a paramount concern in automotive systems, increasingly so with the advent of Advanced Driver Assistance Systems (ADAS) that enhance vehicular safety by integrating a wide range of sensors and automated controls. These systems, coupled with the growing trend of Vehicle-to-Everything (V2X) communication, have made automotive electronic systems more interconnected and complex. However, this increased connectivity also brings about significant security challenges, as these systems become potential targets for various types of cyberattacks. The Controller Area Network (CAN) protocol, which is widely used in automotive systems for in-vehicle communication, lacks inherent security features, making it vulnerable to such threats.

In this project, we focus on developing a Vehicle Collision Detection system using the CAN protocol, specifically targeting the enhancement of vehicular safety. The system utilizes two STM32F407VGT6 microcontroller boards: one for collecting distance data from ultrasonic sensors mounted on the vehicle, and another for processing this data to provide real-time feedback to the driver. The first microcontroller board captures the distance between the vehicle and obstacles in its path and transmits this data to the second board via the CAN bus. The second board then processes the received data to display the distance on an LCD screen and control a buzzer that provides auditory warnings based on the proximity of obstacles.

The system's design is underpinned by the need for real-time operation and robustness, which is achieved by integrating FreeRTOS to manage concurrent tasks efficiently. This project not only demonstrates the effectiveness of using the CAN protocol in safety-critical automotive applications but also highlights the importance of addressing security vulnerabilities inherent in such systems. The modular architecture of the system ensures that it can be adapted for further enhancements, including the integration of more sophisticated ADAS features and advanced security measures to safeguard against potential cyber threats.

Table of Contents

1. Introduction

The "CAN Shield Car Collision Avoidance System" is designed to enhance vehicle safety by preventing collisions through real-time distance monitoring and alert mechanisms. The system is built around two STM32F407VGT6 microcontrollers, which communicate via the Controller Area Network (CAN) protocol, a robust vehicle bus standard known for its reliability in automotive and industrial applications.

In this system, ultrasonic sensors are used to measure the distance between the vehicle and potential obstacles in both the front and rear. This distance data is transmitted between the microcontrollers using CAN transceivers (MCP2551). The receiving microcontroller processes this data and displays the distances on a 16x4 LCD screen connected via the I2C bus. Additionally, a buzzer is employed to alert the driver when the vehicle comes dangerously close to an obstacle, providing a crucial auditory warning to avoid collisions.

The project also leverages FreeRTOS, a real-time operating system, on the receiving microcontroller to manage multiple tasks efficiently. FreeRTOS ensures that the system responds quickly to changes in distance data, updating the display and triggering the buzzer in real-time.

This project aims to demonstrate the effective use of CAN communication in a collision avoidance system, showcasing the integration of microcontrollers, sensors, and real-time software in creating a practical automotive safety solution.

1.1 Problem Statement

In modern vehicles, safety is paramount, and collision avoidance systems are critical in reducing accidents. However, implementing an effective and reliable system that can monitor and respond to potential collisions in real-time remains a challenge. This project addresses this issue by developing a vehicle collision detection system using CAN protocol, which offers robust communication in automotive applications.

1.2 Objective and Specification

The primary objective of the "CAN Shield Car Collision Avoidance System" is to develop a reliable and real-time system that enhances vehicle safety by preventing collisions. The system aims to achieve this by continuously monitoring the distance between the vehicle and potential obstacles in both the front and rear, and providing immediate visual and auditory alerts to the driver. The system should be robust, easy to integrate into existing vehicle electronics, and capable of operating under various driving conditions.

- **Microcontroller:**

  Two STM32F407VGT6 microcontrollers for processing sensor data and managing communication.

- **Communication Protocol:**

  CAN Bus: Utilizes MCP2551 transceivers to establish communication between the two STM32 microcontrollers.
  Baud Rate: Configured to ensure reliable and timely data transmission.

- **Sensors:**

  Ultrasonic Sensors (HC-SR04): Measure the distance between the vehicle and obstacles at the front and rear.

- **Display:**

  16x4 LCD: Connected via the I2C bus to the receiving microcontroller, used to display real-time distance data.

- **Alert System:**

  Buzzer: Activates when the distance to an obstacle falls below a predefined threshold, providing an audible warning to the driver.

- **Software:**

  FreeRTOS: Real-time operating system used on the receiving STM32 microcontroller for efficient task management.
  Task Management:
  - CAN Task: Handles receiving and processing of distance data.
  - Display Task: Manages updates to the 16x4 LCD.
  - Buzzer Task: Controls the activation of the buzzer based on distance thresholds.

- **Power Supply:**

  The system should be powered by the vehicle's electrical system, with appropriate voltage regulation for the microcontrollers and peripherals.

- **Environmental Resilience:**

  The system should be designed to function reliably in various environmental conditions, including temperature variations, vibrations, and electromagnetic interference.

- **Real-Time Performance:**

  The system must respond within milliseconds to changes in distance, ensuring timely alerts and accurate data display.

- **Integration:**

  The system should be designed to be easily integrated into existing vehicle systems with minimal modifications required.

2. Literature Review

The increasing complexity and interconnectivity of modern vehicles have led to the integration of various systems that enhance both safety and driving experience. Among these, the Controller Area Network (CAN) protocol stands out as a critical component in automotive electronics, facilitating communication between various invehicle subsystems. However, the rise of Advanced Driver Assistance Systems (ADAS) and Vehicle-toEverything (V2X) communication has brought new challenges, particularly concerning the security and reliability of these interconnected systems.

1. Controller Area Network (CAN) Protocol:

The CAN protocol, developed by Bosch in the 1980s, has become the standard communication protocol for invehicle networks due to its robustness and real-time capabilities. It allows various microcontrollers and devices within a vehicle to communicate with each other without the need for a host computer. Despite its widespread use, the CAN protocol has notable security limitations. The protocol does not inherently support authentication, encryption, or integrity checking, making it susceptible to various attacks, including message spoofing, replay attacks, and Denial of Service (DoS) attacks. Studies by researchers like Hoppe et al. have demonstrated the potential for security breaches, showing that an attacker could manipulate CAN messages to interfere with critical vehicle functions like airbag deployment or engine control .

2. Collision Avoidance Systems:

Collision avoidance systems, a subset of ADAS, have been widely researched and developed to enhance vehicle safety. These systems typically use a combination of sensors (e.g., ultrasonic, radar, LiDAR) to detect obstacles and provide warnings to the driver. The integration of these systems with the CAN protocol allows for real-time communication between sensors and the vehicle’s central processing units. However, as vehicles become more autonomous, the reliability and security of these communication channels become increasingly critical. A compromised CAN network could lead to false sensor readings or delayed warnings, potentially resulting in accidents rather than preventing them .

3. FreeRTOS in Embedded Systems:

FreeRTOS, an open-source real-time operating system, is widely used in embedded systems for managing tasks in real-time applications. Its lightweight nature and support for multitasking make it suitable for automotive

applications, where multiple subsystems must operate concurrently with precise timing. In the context of vehicle collision detection systems, FreeRTOS can be used to manage tasks such as sensor data acquisition, CAN message processing, and actuator control, ensuring that each task is executed within its required time frame. The use of FreeRTOS in automotive systems is well-documented, with numerous studies highlighting its ability to improve system responsiveness and reliability .

4. Security Concerns in Automotive Networks:

As vehicles become more connected, the threat landscape expands, making cybersecurity a critical concern for automotive networks. The lack of security features in the CAN protocol has prompted researchers and industry professionals to explore various methods to secure in-vehicle communications. Proposed solutions include the use of encryption, message authentication codes (MACs), and intrusion detection systems (IDS) to protect against unauthorized access and tampering. However, these solutions often introduce additional complexity and can impact the real-time performance of the CAN network, highlighting the trade-off between security and system performance .

References:

Hoppe, T., et al. "Security threats to automotive CAN networks – Practical examples and selected short-term countermeasures." Proceedings of the International Conference on Computer Safety, Reliability, and Security, 2008.

Eichler, S. "A security architecture concept for vehicular network nodes." Proceedings of the IEEE 62nd Vehicular Technology Conference (VTC-2005-Fall), 2005.

Barbalace, A., et al. "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application." IEEE Transactions on Industrial Informatics, 2017.

Szilagyi, P., Koopman, P. "Low cost multicast authentication via validity voting in time-triggered embedded control networks." Proceedings of the 9th IEEE International Workshop on Factory Communication Systems, 20

3. Methodology

- **Methodology**

The methodology section outlines the systematic approach used in the design, development, and implementation of the "CAN Shield Car Collision Avoidance System." This project integrates hardware and software components, including microcontrollers, sensors, communication protocols, and a real-time operating system, to create a reliable and responsive collision avoidance system.

1. System Design and Architecture”

1.1 Hardware Design:

- Microcontrollers:
- Two STM32F407VGT6 microcontrollers were selected for their processing power, peripheral interfaces, and compatibility with CAN communication.
- One microcontroller is designated as the transmitter to send distance data from sensors, and the other as the receiver to process the data, display it on an LCD, and control a buzzer.
- CAN Transceivers:
- MCP2551 transceivers are used to enable communication between the two STM32 microcontrollers over the CAN bus. These transceivers provide the necessary electrical interface and fault protection for the CAN protocol.
- Ultrasonic Sensors:
- HC-SR04 ultrasonic sensors are employed to measure the distance to obstacles. The sensors are placed at the front and rear of the vehicle to detect obstacles in both directions.
- 16x4 LCD:
- A 16x4 LCD display is connected via the I2C bus to the receiving microcontroller. The LCD is used to show real-time distance data.
- Buzzer:
- A buzzer is integrated into the system to provide an auditory warning when the vehicle is too close to an obstacle. The buzzer is controlled based on the distance data received.

1.2 Software Design:

- FreeRTOS Integration:-

The receiving microcontroller runs FreeRTOS to manage multiple tasks, ensuring real-time performance and responsiveness.

- Key tasks include CAN message handling, LCD display updates, and buzzer control.
- CAN Protocol Configuration:
- The CAN bus is configured for communication between the two microcontrollers. Key parameters such as baud rate, message ID filtering, and data frame structure are carefully defined to ensure reliable and timely communication.

-Sensor Data Acquisition:

- The transmitter microcontroller collects distance data from the ultrasonic sensors, converts it into a digital format, and transmits it over the CAN bus.
- Data Processing and Display:
- The receiver microcontroller processes the received distance data, updates the LCD display via I2C, and triggers the buzzer if the distance falls below a critical threshold.

2. Implementation

2.1 Hardware Implementation:
- Circuit Design:
- The microcontrollers, CAN transceivers, ultrasonic sensors, LCD, and buzzer are connected according to the designed schematic. Proper attention is given to power supply regulation and signal integrity.
- PCB Design:
- A custom PCB is designed to house all components, ensuring compactness and reliability in a vehicle environment. - Assembly and Testing
- The components are soldered onto the PCB, and initial connectivity tests are performed to ensure correct assembly.

2.2 Software Implementation:
- Firmware Development:
- The firmware for both the transmitter and receiver microcontrollers is developed using STM32CubeIDE. The code is structured into different modules, such as CAN communication, sensor data processing, and task management under FreeRTOS.
- FreeRTOS Task Configuration:
- Tasks are created and prioritized based on the system requirements. The CAN task is given high priority to ensure timely data reception, while the display and buzzer tasks are managed with lower priority.
- CAN Communication Setup:
- CAN communication is initialized with proper baud rate settings and message filtering. The message format is standardized for ease of interpretation at the receiving end.

3. Testing and Validation

3.1 Unit Testing:
- Individual components, such as the ultrasonic sensors, CAN communication, LCD display, and buzzer, are tested separately to verify their functionality.
- FreeRTOS task scheduling and inter-task communication are tested to ensure that the system responds within the required real-time constraints.

3.2 Integration Testing:
- The system is tested as a whole, with all components integrated. Testing scenarios include various obstacle distances to ensure accurate sensor readings, correct CAN message transmission, appropriate LCD updates, and timely buzzer activation.
- The system's response to multiple obstacles and rapid changes in distance is evaluated to validate its performance in dynamic conditions.

3.3 Environmental Testing:
- The system is subjected to different environmental conditions, such as temperature variations, vibrations, and electromagnetic interference, to ensure robustness and reliability in real-world vehicle scenarios.

4. Optimization and Refinement

    4.1 Performance Tuning:
    - The CAN bus configuration and FreeRTOS task scheduling are optimized for the best performance. This includes fine-tuning the baud rate, adjusting task priorities, and optimizing code execution paths.
    - Sensor calibration is performed to improve the accuracy of distance measurements.

    4.2 Bug Fixes and Enhancements:
    - Any identified bugs during testing are fixed, and enhancements are made to improve system reliability and user experience, such as adding debounce logic for the ultrasonic sensors or optimizing the LCD refresh rate.

5. Documentation and Reporting

    5.1 System Documentation:
    - Detailed documentation is prepared, covering hardware schematics, software architecture, and user manuals. This documentation serves as a reference for future maintenance and upgrades.

    5.2 Project Report:
    - A comprehensive project report is compiled, including the introduction, literature review, problem statement, objectives, methodology, results, and conclusions. The report is structured to provide a clear understanding of the project development process and its outcomes.

    This methodology ensures a systematic and thorough approach to developing the CAN Shield Car Collision Avoidance System, from initial design through implementation and testing, leading to a functional and reliable automotive safety solution**.**

3.1.Block Diagram

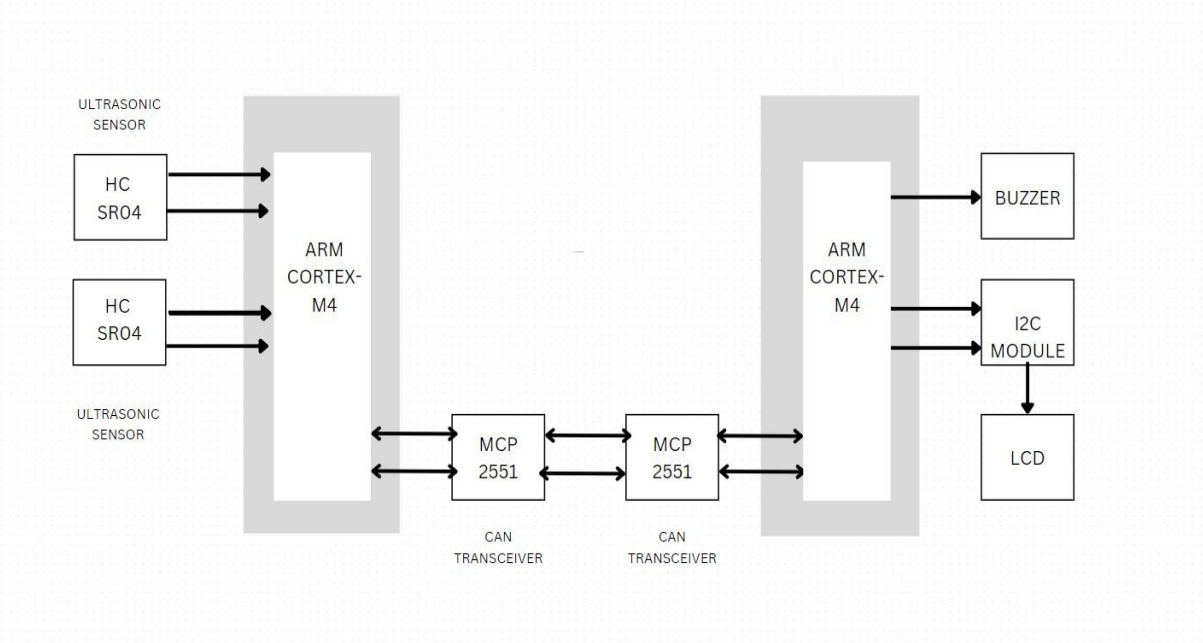


Figure : Block Diagram for proposed System

Proposed System

4.1 STM32F407VGT6 Pin Configuration

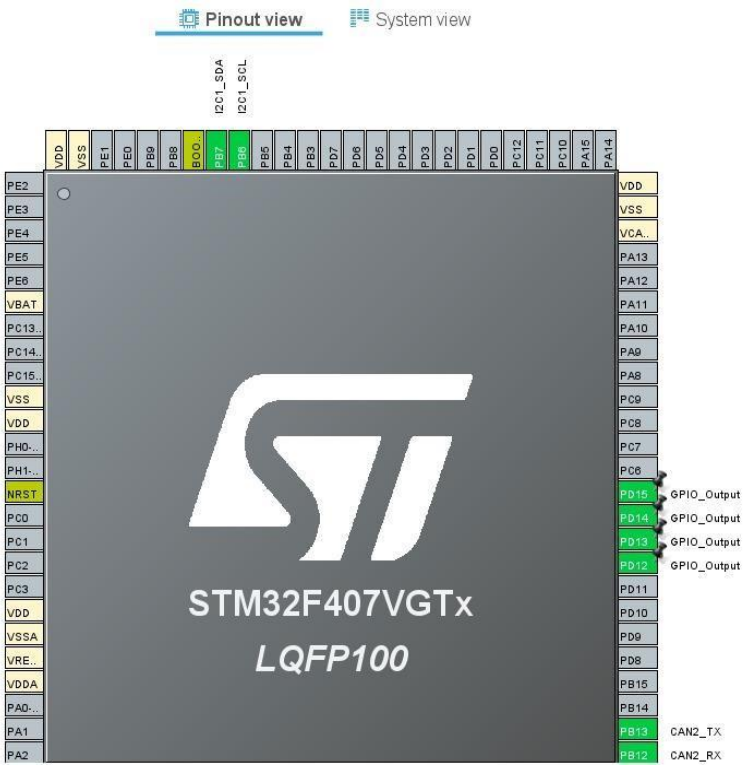


Figure:PinConfiguration of STM32F407VGT6

4.2 Clock Configuration

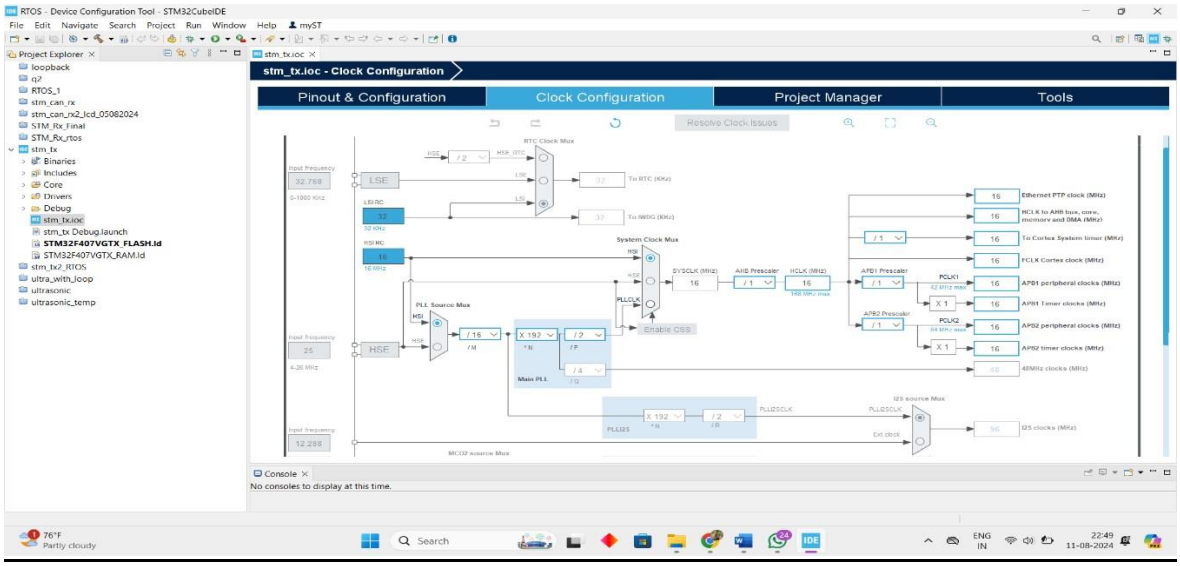


Figure 3: Clock Configuration

.3 Hardware and Components

4.3.1 STM32F407VGT6

The STM32F407VGt6 microcontroller is a high-performance ARM Cortex-M4-based microcontroller unit (MCU) manufactured by STMicroelectronics. It offers a wide range of features and peripherals suitable for various embedded applications, including industrial control systems, consumer electronics, and IoT devices.

- The STM32F407VGt6 MCU serves as the central processing unit for the system, handling data acquisition, processing, and communication tasks.
- It features a powerful ARM Cortex-M4 core running at up to 168 MHz, providing sufficient processing power for real-time sensor data processing and control algorithms.
- Peripherals and Interfaces:
- The STM32F407VGt6 MCU is equipped with a rich set of peripherals and interfaces, including multiple UART, SPI, I2C, and CAN interfaces.
- These interfaces facilitate communication with external sensors, modules, and devices, enabling seamless integration into the system architecture.
- Data Acquisition: The STM32F407VGt6 MCU interfaces with various sensors, such as temperature sensors, speed sensors, NO2 sensors, and others, to collect real- time data.
- It employs dedicated ADC (Analog-to-Digital Converter) channels to digitize analog sensor readings, ensuring accurate and reliable data acquisition.
- The MCU supports various communication protocols, such as UART, SPI, and I2C, for serial communication with peripheral devices.
- It facilitates communication with external modules, including the Bluetooth HC-05 module, ESP32 module, LCD display, and others, enabling seamless data transmission and control.

- Firmware development for the STM32F407VGt6 MCU is carried out using integrated development environments (IDEs) such as STM32CubeIDE or Keil µVision.

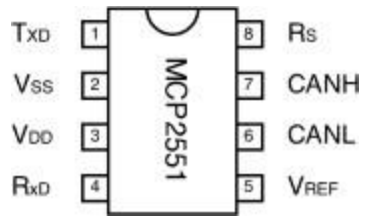
4.3.2 HC – SR04 Ultrasonic Sensor

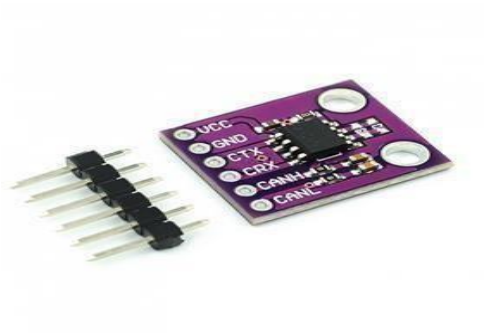
- These sensors generate high frequency sound waves and calculate the time interval between sending the signal and receiving the echo to determine the distance to an object.

- Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules include ultrasonic transmitters, receiver, and control circuit. The basic principle of work:
- Using IO trigger for at least 10us high level signal,
- The Module automatically sends eight 40 kHz and detects whether there is apulse signal back.
- F the signal comes back, through high level, time of high output IO duration is the time from sending ultrasonic to returning.
- Test distance = (high level time × velocity of sound (340M/S) / 2.

4.3.3 MCP2551(CAN TRANSCEIVER)

- Supports 1 Mb/s operation.
- Implements ISO-11898 standard physical layer requirements.
- Suitable for 12V and 24V systems.
- Externally controlled slope for reduced RFI emissions.
- Detection of ground fault (permanent dominant) on TXD input
- Power-on reset and voltage brown-out protection.
- An unpowered node or brown-out event will not disturb the CAN bus.
- Low current standby operation
- Protection against high-voltage transients • Automatic thermal shutdown protection
- Up to 112 nodes can be connected.
- High noise immunity due to differential bus implementation
- Temperature ranges: - Industrial (I): -40°C to +85°C - Extended (E): -40°C to +125°C





4.3.4 LCD 16x2 Display

A 16x4 LCD display is a type of liquid crystal display that can show information in a grid of 16 columns and 4 rows. This kind of display is commonly used in various electronic projects and devices due to its versatility and relatively low cost.

- **Key Features:**

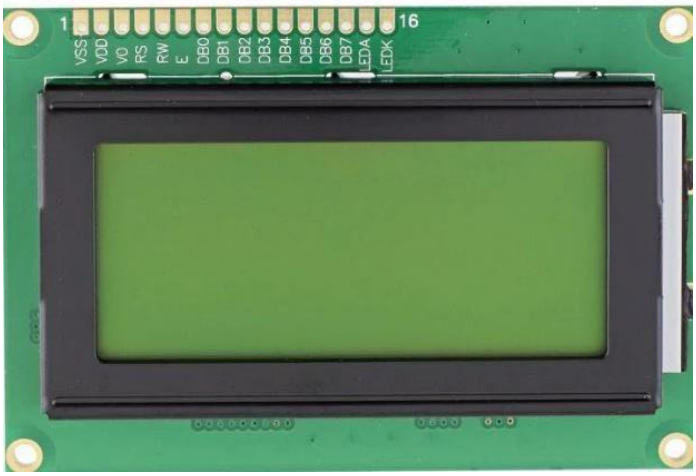
Display Size: 16 columns and 4 rows of characters. Each character is usually 5x8 pixels in size.

Character Set: Displays standard alphanumeric characters, along with some special symbols.

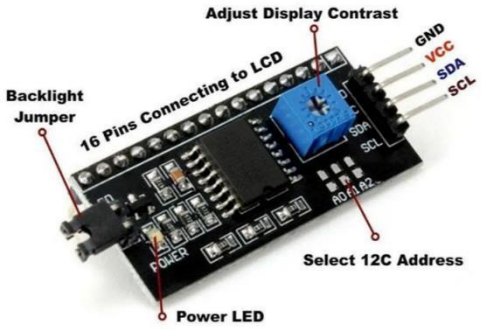
Backlight: Some models come with a backlight to make the display easier to read in low light

conditions.

Interface: Typically uses an HD44780 controller or compatible, which is standard for many LCD modules. This controller allows you to control the display with a variety of commands.



4.3.5 I2C Module



Specifications:

- IC Chip: PCF8574
- Power Supply: 5V

- Serial I2C control of LCD display using PCF8574
- Backlight can be enabled or disabled via a jumper on the board.
- Contrast control via a potentiometer.
- Can have 8 modules on a single I2C bus (change address via solder jumpers) address, allowing
- Size: 41.6 x 19.2 mm

5 Software Description

5.1 STM32CubeIDE

STM32CubeIDE, an integrated development environment (IDE) crafted by STMicroelectronics, serves as a pivotal tool in the project's software development journey targeting STM32 microcontrollers. This environment provides a holistic platform with an array of tools and features aimed at expediting firmware development. Its seamless integration with the STM32CubeMX configuration tool streamlines the process of configuring STM32 peripherals, pin assignments, and middleware components. By visually configuring the MCU's parameters through STM32CubeMX, developers can swiftly generate initialization code, significantly reducing the workload associated with peripheral setup. Furthermore, STM32CubeIDE offers robust project management capabilities, facilitating efficient organization of source files, libraries, and resources within projects. With built-in debugging and testing functionalities, including support for hardware debugging using ST-LINK or JTAG/SWD debug probes, developers can effectively debug firmware code using features like breakpoints and watchpoints. Additionally, STM32CubeIDE comes bundled with the GNU Arm Embedded Toolchain, providing a robust compiler and toolchain optimized for ARM Cortex-M-based microcontrollers. This toolchain supports advanced compiler optimizations and debugging features, enhancing code efficiency and reliability. Integrated seamlessly with STM32Cube middleware components and software libraries, the IDE enables developers to easily incorporate middleware functionalities into their projects, further accelerating the development process. Through STM32CubeIDE's comprehensive suite of features, developers can efficiently develop, debug, and deploy firmware for STM32 microcontroller-based projects, including the envisioned wireless data transmission system.

5.1.1 STM32Cube Programmer

STM32Cube Programmer stands as an essential tool within the STM32 ecosystem, offering crucial functionalities for programming STM32 microcontrollers and configuring their embedded memories. This versatile tool streamlines the process of

flashing firmware onto STM32 devices and managing their memory configurations. By supporting various programming modes, including UART, USB, and CAN, STM32Cube Programmer accommodates diverse deployment scenarios and ensures compatibility with a wide range of STM32 microcontrollers. Its intuitive user

interface simplifies the task of selecting programming options and configuring device settings, enabling efficient and error-free programming operations. Moreover, STM32Cube Programmer integrates seamlessly with STM32CubeIDE and other development environments, providing a seamless workflow for firmware development and device programming. With support for batch programming and scripting capabilities, the tool enhances productivity and scalability, enabling developers to streamline production processes and automate repetitive tasks. Overall, STM32Cube Programmer plays a crucial role in the development lifecycle of STM32- based embedded systems, offering robust programming capabilities and streamlined device management functionalities.

5.2 SEGGER

SystemView is a real-time recording and visualization tool for embedded systems. It reveals the true runtime behavior of an application, going far deeper than the system insights provided by debuggers. This is particularly effective when developing and working with complex embedded systems comprising multiple threads and interrupts. SystemView can ensure a system performs as designed, track down inefficiencies, and find unintended interactions and resource conflicts.

SEGGER is a company that specializes in providing software and hardware solutions for embedded systems development. They offer a wide range of products and services tailored for embedded developers, including:

5.1.2 Embedded Software: SEGGER provides various software components such as real-time operating systems (RTOS), middleware (like file systems and communication stacks), and software development tools (like debuggers and flash programmers).

5.1.3 Embedded Tools: SEGGER offers a suite of development tools, including SEGGER Embedded Studio (an integrated development environment), J-Link debug probes (for debugging and programming microcontrollers), and Flasher (for programming flash memory on microcontrollers).

5.1.4 RTOS: SEGGER's embedded OS is a real-time operating system designed for embedded systems. It provides multitasking support, real-time response, and a small footprint, making it suitable for a wide range of embedded applications.

6 Communication Protocols

6.1 Controller Area Network

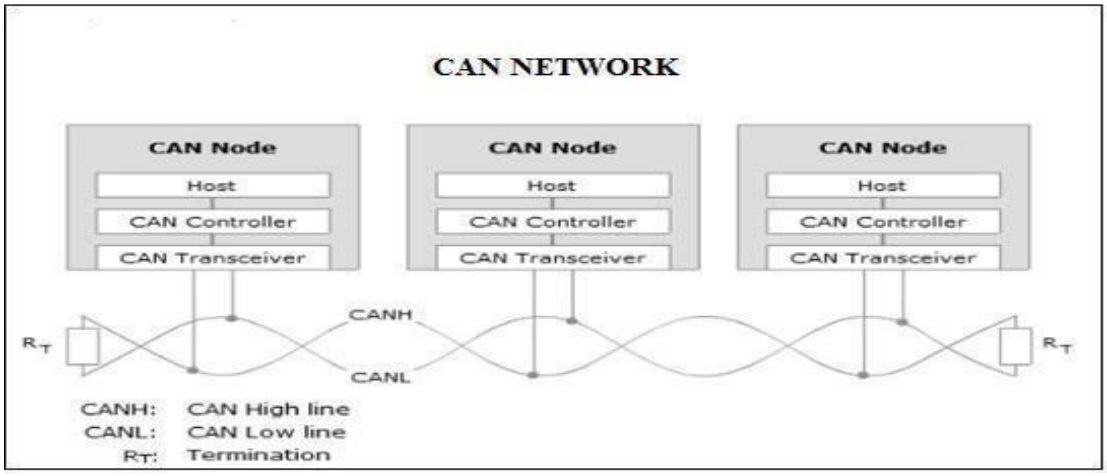
The Controller Area Network bus is an International Standardization Organization (ISO) defined serial communications bus originally developed for the automotive industry to replace the complex wiring harness with a two-wire bus. The Controller Area Network bus, or CAN bus, is a vehicle bus standard designed to allow devices and microcontrollers to communicate with each other within a vehicle without a host computer.

CAN is a reliable, real-time protocol that implements a multicast, data-push, publisher/subscriber model. CAN messages are short (data payloads are a maximum of 8 bytes, headersare 11 or 29 bits), so there is no centralized master or hub to be a single point of failure and it is flexible in size. Its real-time features include deterministic message delivery time and global prioritywith prioritized message IDs.

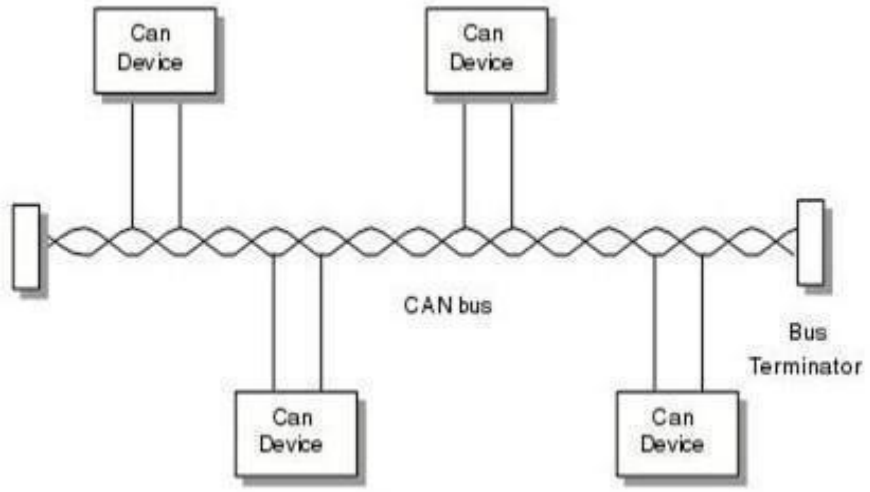
Bus arbitration is accomplished in CAN using bit dominance, a process where nodes begin to transmit their message headers on the bus, then drop out of the “competition” when a dominant bit is detected on the bus, indicating a message ID of higher priority being transmitted elsewhere. This means bus arbitration does not add overhead because once the bus is “won” the node simply continues sending its message. Because there is no time lost to collisions on a heavily loaded network.

- **CAN NETWORK**

  A CAN network consists of several CAN nodes which are linked via a physical transmission medium (CAN bus). In practice, the CAN network is usually based on a line topology with a linear bus to which a few electronic control units are each connected viaa CAN interface. The passive star topology may be used as an alternative. An unshielded twisted two-wire line is the physical transmission medium used most frequently in applications(Unshielded Twisted Pair — UTP), over which symmetrical signal transmission occurs. The maximum data rate is 1 Mbit/s. A maximum network extension of about 40 meters is allowed. At the ends of the CAN network, bus termination resistors contribute to preventing transient phenomena (reflections). ISO 11898 specifies the maximum number of CAN nodes as 32.



- **CAN BUS**



Physical signal transmission in a CAN network is based on transmission of differential voltages (differential signal transmission). This effectively eliminates the negative effects of interference voltages induced by motors, ignition systems and switch contacts. Consequently, the transmission medium (CAN bus) consists of two lines: CAN High and CAN Low.

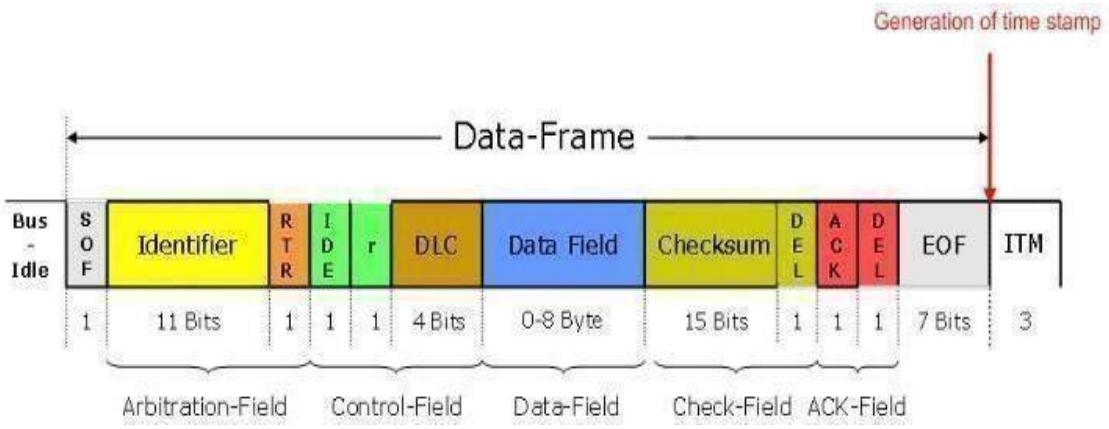
Twisting ofthe two lines reduces the magnetic field considerably. Therefore, in practice twisted pair conductors are generally used as the physical transmission medium. Due to finite signal propagation speed, the effects of transient phenomena (reflections) grow with increasing data rate and bus extension. Terminating the ends of the communication channel using termination resistors (simulation of the electrical properties of the transmission medium) prevents reflections in a high-speed CAN network.

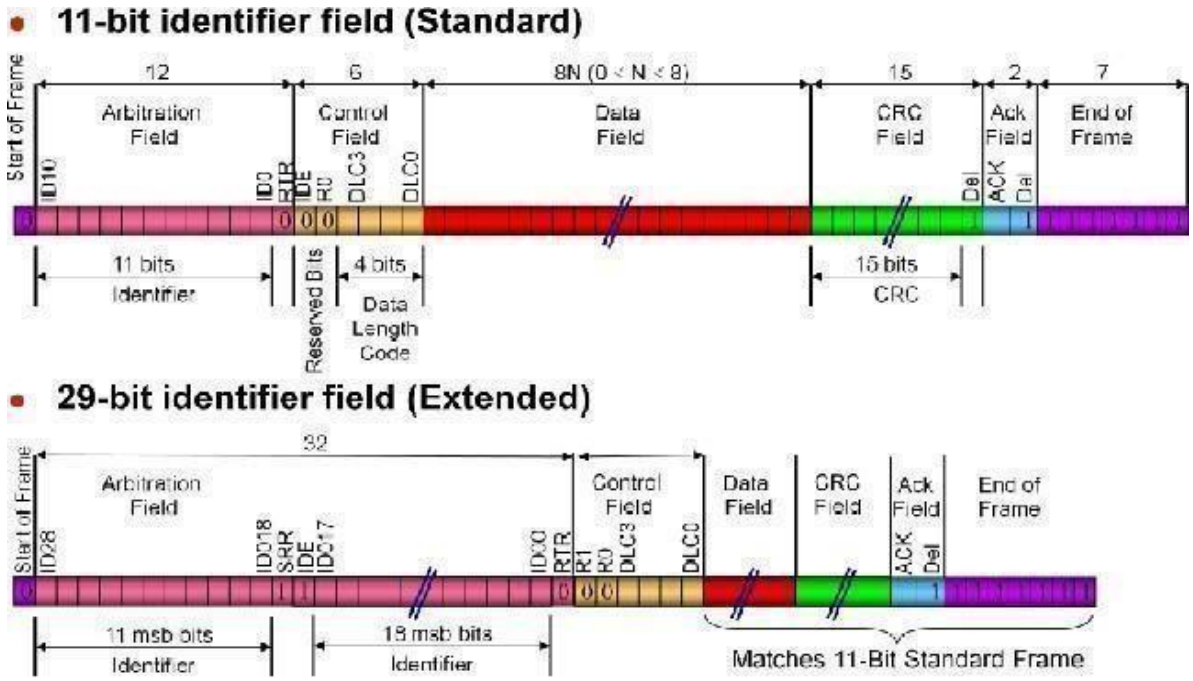
The key parameter for the bus termination resistor is the so-called characteristic impedance of the electrical line. This is 120 Ohm. In contrast to ISO 11898-2, ISO 11898-3 (low-speed CAN) does not specify any bus termination resistors due to the low maximum data rate of 125 kbit/s.

- **CAN FRAMES**
  1. Data Frame
  2. Overload Frame
  3. Remote Frame
  4. Error Frame

Data Frame:

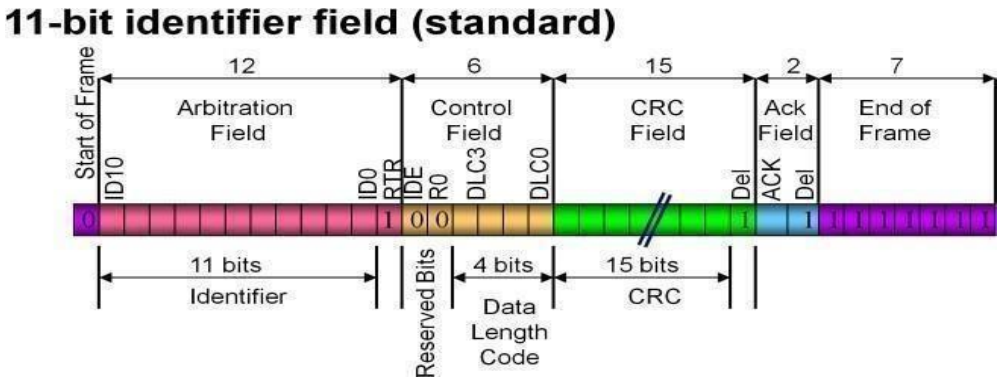
The data frame is the most common message type, and comprises the Arbitration Field, the Data Field, the CRC Field, and the Acknowledgment Field. The Arbitration Field contains an 11-bit identifier and the RTR bit, which is dominant for data frames. Next is the Data Field which contains zero to eight bytes of data, and the CRC Field which contains the 16-bit checksum used for error detection. Last is the Acknowledgment Field.





Remote Frame:

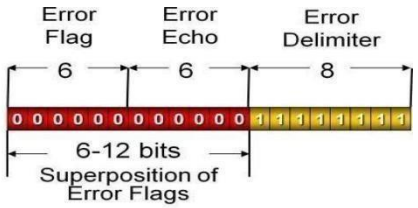
The intended purpose of the remote frame is to solicit the transmission of data from another node. The remote frame is similar to the data frame, with two important differences. First, this type of message is explicitly marked as a remote frame by a recessive RTR bit in the arbitration field, and secondly, there is no data.



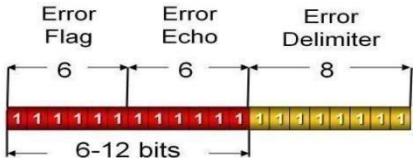
Error Frame:

The error frame is a special message that violates the formatting rules of a CAN message. It is transmitted when a node detects an error in a message and causes all other nodes in the network to send an error frame as well. The original transmitter then automatically retransmits the message. An elaborate system of error counters in the CAN controller ensures that a node cannot tie up a bus by repeatedly transmitting error frames.

- **Active Error Frame**

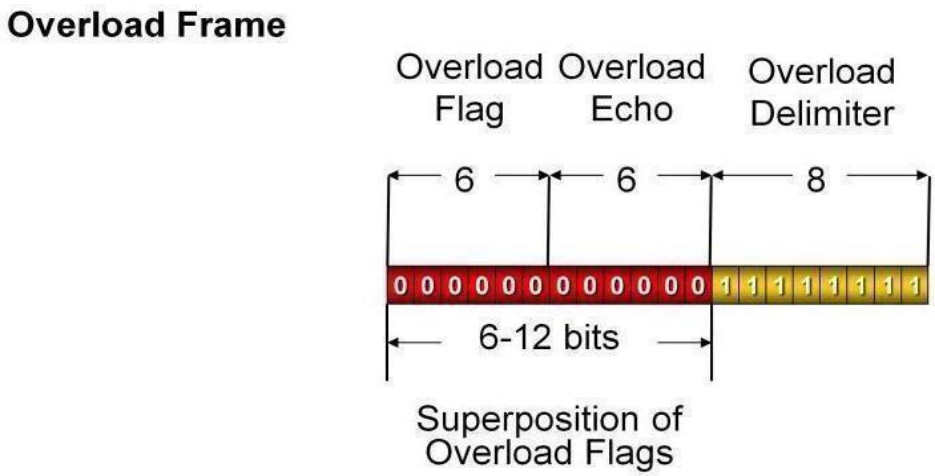


- **Passive Error Frame**



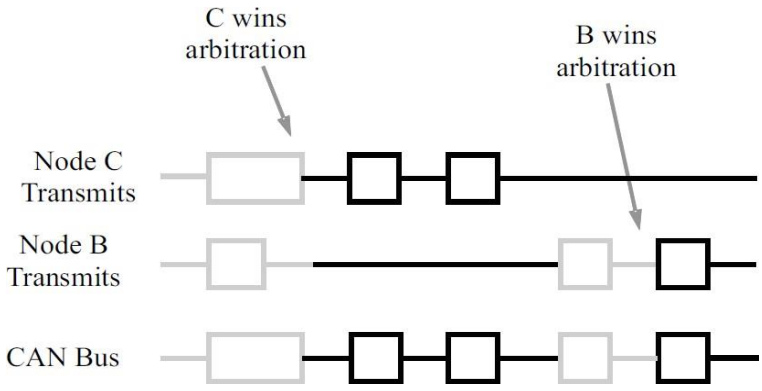
Overload Frame:

The overload frame is mentioned for completeness. It is like the error frame with regardto the format, and it is transmitted by a node that becomes too busy. It is primarily used to provide for an extra delay between messages.



- BUS ARBITRATION
  The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP). CSMA means that each node on a bus must wait for a prescribed period of inactivity before attempting to send a message. CD+AMP means that collisions are resolved through a bit-wise arbitration, based on a pre- programmed priority of each message in the identifier field of a message. The higher priority identifier always wins bus access. That is, the last logichigh in the identifier keeps on transmitting because it is the highest priority.
  Whenever the bus is free, any unit may start to transmit a message. If two or more units start transmitting messages at the same time, the bus access conflict is resolved by bit-wise arbitration using the Identifier. The mechanism of arbitration guarantees that neither information nor time is lost. If a data frame and a remote frame with the same identifier are initiated at the same time, the data frame prevails over the remote frame. During arbitration every transmitter compares the level of the bit transmitted with the level that is

monitored on and a "dominant" level is monitored, the unit has lost arbitration and must withdraw without sending one more bit. the bus. If these levels are equal the unit may continue to send.
When a "recessive" level is sent



6.2 Inter-Integrated Circuit

I2C, short for Inter-Integrated Circuit, is a multi-master, multi-slave, packet-switched, single-ended, serial communication bus. It is used to connect low-speed peripherals to a microcontroller in a costeffective and simple manner. I2C was developed by Philips (now NXP Semiconductors) in the early 1980s.

Key Features

- Two-Wire Interface: I2C communication requires only two bi-directional lines: Serial Data Line (SDA) and Serial Clock Line (SCL).
- Addressing: Each device on the I2C bus has a unique address. The address can be 7-bit or 10-bit long.
- Data Transfer: Data is transferred in bytes (8 bits), with each byte accompanied by an acknowledgment bit.
- Clock Synchronization: The clock line (SCL) is controlled by the master device, which synchronizes the data transfer**.**

Communication Protocol Start

Condition:

Initiated by the master device by pulling the SDA line low while SCL is high. This signals the beginning of a communication session.

Addressing:

The master sends the address of the target slave device. The address is followed by a read/write bit indicating the desired operation.

Data Transfer:

Data is transferred in 8-bit bytes. After each byte, the receiver sends an acknowledgment bit to confirm receipt of the data.

Stop Condition:

Initiated by the master device by pulling the SDA line high while SCL is high. This signals the end of the communication session.

7 RTOS

7.1 Free RTOS

The firmware code for the project extensively utilizes FreeRTOS, an open-source real- time operating system kernel, to manage tasks, synchronize operations, and handle interrupts effectively. The implementation of FreeRTOS within the firmware code encompasses several key aspects:

7.1.1 Task Creation

- Two tasks are created using FreeRTOS APIs to enable concurrent execution of different functionalities within the embedded system.
- Task creation involves defining task functions, specifying task priorities, and allocating stack space for each task.

7.1.2 Semaphore Creation

- Semaphores are created using FreeRTOS APIs to facilitate synchronization and mutual exclusion between tasks.
- Semaphores are utilized to coordinate access to shared resources and prevent race conditions in critical sections of the firmware code.

7.1.3 Queue Creation

- Queues are implemented using FreeRTOS APIs to enable inter-task communication and data exchange.
- Queues allow tasks to send and receive messages, sensor readings, or control commands asynchronously, ensuring seamless communication between tasks.

7.1.4 Interrupt Handling

- FreeRTOS provides mechanisms for handling interrupts and integrating interrupt service routines (ISRs) into the real-time operating system.
- Interrupts are configured and managed using FreeRTOS APIs to ensure timely and deterministic response to hardware interrupts.

7.1.5 Scheduler Configuration

- The FreeRTOS scheduler is configured to prioritize tasks based on their criticality and execution requirements.
- Task priorities are carefully assigned to optimize system performance and responsiveness, ensuring that high-priority tasks are executed with minimal latency.

8 SOURCE CODE

• TRANSMIT CODE main.c

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * Authors: Prasad Kulkarni,Onkar chougule, Lalit Deshpande,Karan Solanki
  * CDAC ACTS,pune DESD(march 2024)
  * @file           : main.c
  * @brief          : Transmitting ultrasonic data to another node via CAN
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <string.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

#define TRIG_PORT_FRONT  GPIOA
```

```
#define TRIG_PIN_FRONT   GPIO_PIN_0
#define ECHO_PIN_FRONT   GPIO_PIN_1
```

```
#define TRIG_PORT_REAR   GPIOD
#define TRIG_PIN_REAR    GPIO_PIN_12
#define ECHO_PIN_REAR    GPIO_PIN_13

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
CAN_HandleTypeDef hcan2;
TIM_HandleTypeDef htim2;
/* USER CODE BEGIN PV */
uint8_t distance_message[8];
float distance_front;
float distance_rear;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_CAN2_Init(void);
static void MX_TIM2_Init(void);
/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_CAN2_Init(void);
static void MX_TIM2_Init(void);
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
float getDistance(GPIO_TypeDef *GPIO_PORT, uint16_t GPIO_Pin_Trigger, uint16_t
GPIO_Pin_Echo);
void CAN_Filter_Config(void);
void CAN2_Tx(float,float);
```

```
void Error_Handler(void);
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
```

```
  */
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick.
*/

    HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_CAN2_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */
  CAN_Filter_Config();

  if (HAL_CAN_Start(&hcan2) != HAL_OK)
  {
```

```
      Error_Handler();
    }

    HAL_TIM_Base_Start(&htim2);  // Start the timer

    /* USER CODE END 2 */

    /* Infinite loop */
    /* USER CODE BEGIN WHILE */
    while (1)
```

```
    {
        /* receiving front distance from ultrasonic sensor */
        distance_front =
  getDistance(TRIG_PORT_FRONT,TRIG_PIN_FRONT,ECHO_PIN_FRONT);
        /* for optimization , cm conversion dividing it with 2*/
        distance_front = distance_front/2;

        /* receiving rear distance from ultrasonic sensor */
        distance_rear = getDistance(TRIG_PORT_REAR,TRIG_PIN_REAR,ECHO_PIN_REAR);
        /* for optimization , cm conversion dividing it with 2*/
        distance_rear = distance_rear/2;

        /* transmitting front and rear distance to can for transmission*/
        CAN2_Tx(distance_front,distance_rear);
        HAL_Delay(500);
        /* USER CODE END WHILE */
      /* USER CODE BEGIN 3 */
    }
    /* USER CODE END 3 */
  }

  /**
    * @brief System Clock Configuration
    * @retval None
    */
  void SystemClock_Config(void)
  {
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage
```

```
    */
    __HAL_RCC_PWR_CLK_ENABLE();
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /** Initializes the RCC Oscillators according to the specified parameters
    * in the RCC_OscInitTypeDef structure.
    */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
      Error_Handler();
```

```
    }

    /** Initializes the CPU, AHB and APB buses clocks
    */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
    {
      Error_Handler();
    }
  }

  /**
    * @brief CAN2 Initialization Function
    * @param None
    * @retval None
    */
  static void MX_CAN2_Init(void)
  {

    /* USER CODE BEGIN CAN2_Init 0 */
```

```
  /* USER CODE END CAN2_Init 0 */

  /* USER CODE BEGIN CAN2_Init 1 */

  /* USER CODE END CAN2_Init 1 */
  hcan2.Instance = CAN2;
  hcan2.Init.Prescaler = 24;
  hcan2.Init.Mode = CAN_MODE_NORMAL;
  hcan2.Init.SyncJumpWidth = CAN_SJW_1TQ;
  hcan2.Init.TimeSeg1 = CAN_BS1_13TQ;
  hcan2.Init.TimeSeg2 = CAN_BS2_2TQ;
  hcan2.Init.TimeTriggeredMode = DISABLE;
  hcan2.Init.AutoBusOff = DISABLE;
  hcan2.Init.AutoWakeUp = DISABLE;
  hcan2.Init.AutoRetransmission = DISABLE;
  hcan2.Init.ReceiveFifoLocked = DISABLE;
  hcan2.Init.TransmitFifoPriority = DISABLE;
  if (HAL_CAN_Init(&hcan2) != HAL_OK)
```

```
  {
    Error_Handler();
  }
  /* USER CODE BEGIN CAN2_Init 2 */

  /* USER CODE END CAN2_Init 2 */

}

/**
  * @brief TIM2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
```

```
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM2_Init 1 */

    /* USER CODE END TIM2_Init 1 */
    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 83;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 4294967295;
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
    {
      Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
    {
      Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
```

```
    {
      Error_Handler();
    }
    /* USER CODE BEGIN TIM2_Init 2 */

    /* USER CODE END TIM2_Init 2 */

  }

  /**
    * @brief GPIO Initialization Function
    * @param None
    * @retval None
    */
  static void MX_GPIO_Init(void)
  {
    GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */
```

```
  /* USER CODE END MX_GPIO_Init_1 */

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET);

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, GPIO_PIN_RESET);

    /*Configure GPIO pin : PA0 */
    GPIO_InitStruct.Pin = GPIO_PIN_0;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /*Configure GPIO pin : PA1 */
    GPIO_InitStruct.Pin = GPIO_PIN_1;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /*Configure GPIO pin : PD12 */
```

```
    GPIO_InitStruct.Pin = GPIO_PIN_12;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

    /*Configure GPIO pin : PD13 */
    GPIO_InitStruct.Pin = GPIO_PIN_13;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

  /* USER CODE BEGIN MX_GPIO_Init_2 */
  /* USER CODE END MX_GPIO_Init_2 */
```

```
}

/* USER CODE BEGIN 4 */



/**
  * @brief This function configures the CAN filter
  * @retval None
  */
void CAN_Filter_Config(void)
{
  CAN_FilterTypeDef can2_filter_init;

  can2_filter_init.FilterActivation = ENABLE;
  can2_filter_init.FilterBank = 0;
  can2_filter_init.FilterFIFOAssignment = CAN_RX_FIFO0;
  can2_filter_init.FilterIdHigh = 0x65D << 5; // Standard ID shifted to 11 bits
  can2_filter_init.FilterIdLow = 0x0000;
  can2_filter_init.FilterMaskIdHigh = 0xFFFF;
  can2_filter_init.FilterMaskIdLow = 0x0000;
  can2_filter_init.FilterMode = CAN_FILTERMODE_IDMASK;
  can2_filter_init.FilterScale = CAN_FILTERSCALE_32BIT;

  if (HAL_CAN_ConfigFilter(&hcan2, &can2_filter_init) != HAL_OK)
  {
    Error_Handler();
  }
}

```

```
/**
  * @brief This function sends a CAN message with the distance data
  * @param distance: The distance value to send
  *
  */
void CAN2_Tx(float f_distance,float r_distance)
{
  CAN_TxHeaderTypeDef TxHeader;
  uint32_t TxMailbox;

  /* here we are sending front and rear distance in one single array
```

```
     * where first half of array will contain front ultrasonic distance value
     * and second half of array will contain rear ultrasonic distance value*/
    memcpy(&distance_message[0],&f_distance,sizeof(float));
    memcpy(&distance_message[4],&r_distance,sizeof(float));


    TxHeader.DLC = 8;                           /* The Data Length Code specifies the
length
                                                 *of the data in the CAN frame */

    TxHeader.StdId = 0x65D;                     /*The Standard Identifier is the
unique address used
                                                 *to identify the CAN message. It's
used for arbitration on
                                                 *to the CAN bus.*/

    TxHeader.IDE = CAN_ID_STD;                  /* The Identifier Extension bit
determines whether the CAN message
                                                 * uses a standard 11-bit identifier
(CAN_ID_STD)
                                                 * or an extended 29-bit identifier
(CAN_ID_EXT)*/

    TxHeader.RTR = CAN_RTR_DATA;                /*RTR: Stands for Remote Transmission
Request.
                                                 *CAN_RTR_DATA indicates that the
message is a data frame,
                                                 *CAN_RTR_DATA meaning it contains
actual data to be sent*/


    TxHeader.TransmitGlobalTime = DISABLE;      /* This option allows you to include
a global timestamp in the CAN frame,
                                                 * which could be used for
synchronization purposes.*/
```

```

    /* transmitting the message */
    if (HAL_CAN_AddTxMessage(&hcan2, &TxHeader, distance_message, &TxMailbox) !=
HAL_OK)
    {
      Error_Handler();
    }
     while (HAL_CAN_IsTxMessagePending(&hcan2, TxMailbox)); // Wait
until transmission is complete
```

```
}



/**
  * @brief This function gets the distance from the ultrasonic sensor
  * @retval Distance in cm
  */
float getDistance(GPIO_TypeDef *GPIO_PORT, uint16_t GPIO_Pin_Trigger, uint16_t
GPIO_Pin_Echo)
{
  float local_time = 0;
  HAL_GPIO_WritePin(GPIO_PORT, GPIO_Pin_Trigger, GPIO_PIN_SET);      /* Pull the
TRIG pin HIGH          */
  HAL_Delay(1);                                                      /* Wait for
1 ms                   */
  HAL_GPIO_WritePin(GPIO_PORT, GPIO_Pin_Trigger, GPIO_PIN_RESET);    /* Pull the
TRIG pin low           */
   while (!(HAL_GPIO_ReadPin(GPIO_PORT, GPIO_Pin_Echo)));             /* Wait
for the ECHO pin to go high*/


  __HAL_TIM_SET_COUNTER(&htim2, 0);                                  /* Clear the
timer counter         */

  while (HAL_GPIO_ReadPin(GPIO_PORT, GPIO_Pin_Echo));                /* Wait for
the ECHO pin to go low */

  local_time = __HAL_TIM_GET_COUNTER(&htim2);                         /* Read the
timer counter          */

  return local_time * 0.34 / 2;                                      /* Convert
to distance in cm       */

}


/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
```

```
  {
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
  }

  #ifdef  USE_FULL_ASSERT
  /**
    * @brief  Reports the name of the source file and the source line number
    *         where the assert_param error has occurred.
    * @param  file: pointer to the source file name
    * @param  line: assert_param error line source number
    * @retval None
    */
  void assert_failed(uint8_t *file, uint32_t line)
  {
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the file name and line number,
       ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* USER CODE END 6 */
  }
  #endif /* USE_FULL_ASSERT */

```

- **RECEIVE CODE main.c**

```
  /* USER CODE BEGIN Header */
  /**
  ------------------------------------------------------------------------------

  Authors: Prasad kulkarni , Onkar chougale , Lalit Deshpande ,Karan Solanki
```

```
 
  CODE: receiving distance with CAN (front, rear) and displaying on LCD and
  controlling Buzzer
  Date: 05-08-2024
  ---------------------------------------------------------------------------
    ******************************************************************************
    * @file           : main.c
    * @brief          : Main program body
    ******************************************************************************
    * @attention
    *
    * Copyright (c) 2024 STMicroelectronics.
    * All rights reserved.
    *
    * This software is licensed under terms that can be found in the LICENSE file
    * in the root directory of this software component.
    * If no LICENSE file comes with this software, it is provided AS-IS.
    *
    ******************************************************************************
    */
  /* USER CODE END Header */
  /* Includes ------------------------------------------------------------------*/
  #include "main.h"
  /* Private includes ----------------------------------------------------------*/

  /* USER CODE BEGIN Includes */
  #include <stdio.h>
  #include <string.h>
  #include <math.h>
  #include <stdlib.h>
  #include <lcd_i2c.h>        /* LCD required functionality */

  #include "FreeRTOS.h"       /* RTOS Related configurations */
  #include "task.h"
  #include "timers.h"
```

```
  #include "queue.h"
  #include "semphr.h"
  #include "event_groups.h"
  #include "SEGGER_SYSVIEW.h"
  /* USER CODE END Includes */
```

```
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/

/* USER CODE BEGIN PM */
/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
CAN_HandleTypeDef hcan2;

I2C_HandleTypeDef hi2c1;

/* USER CODE BEGIN PV */

//uint8_t Queue_rx[8];
uint8_t RxData[8];
uint32_t min_distance;        /* this variable will store minimum distance of two
UltraSonic sensor */
float distance_rear;        /* received rear distance via CAN */
float distance_front;         /* received rear distance via CAN */
uint32_t period;
extern I2C_HandleTypeDef hi2c1;

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_CAN2_Init(void);
static void MX_I2C1_Init(void);
/* USER CODE BEGIN PFP */

/* Private function prototypes -----------------------------------------------*/
```

```
#define STACK_SIZE 128

void Deferred_Task( void* argument );
void WasteFullLED( void* argument );
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_CAN2_Init(void);
void CAN_Filter_Config(void);
void Error_Handler(void);

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

//static QueueHandle_t Data_transmitt = NULL;

TaskHandle_t xDeferrefTaskHandle = NULL;

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
```

```
    SystemClock_Config();

```

```
    /* USER CODE BEGIN SysInit */

    /* USER CODE END SysInit */

    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_CAN2_Init();
    MX_I2C1_Init();
    /* USER CODE BEGIN 2 */
    /* segger system configuration for segger system view  */
    SEGGER_SYSVIEW_Conf();

    /* making priority of task to 4
     * With FreeRTOS, it's important that critical system interrupts
     *(like the SysTick handler or PendSV handler) are not blocked by other less
     * critical interrupts. By using NVIC_PRIORITYGROUP_4, you can assign the
  lowest preemption
     * priority to FreeRTOS interrupts, ensuring they are not blocked.*/
    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

    /* filtering the CAN configurations before starting the can */
    CAN_Filter_Config();

    /**
      * @brief  Start the CAN module.
      * @param  hcan pointer to an CAN_HandleTypeDef structure that contains
      *         the configuration information for the specified CAN.
      * @retval HAL status
      */
    if(HAL_CAN_Start(&hcan2) != HAL_OK)
    {
      Error_Handler();
    }
    /**
      * @brief  Enable interrupts.
      * @param  hcan pointer to an CAN_HandleTypeDef structure that contains
      *         the configuration information for the specified CAN.
      * @param  ActiveITs indicates which interrupts will be enabled.
      *         This parameter can be any combination of @arg CAN_Interrupts.
```

```
      * @retval HAL status
      */
    if(HAL_CAN_ActivateNotification(&hcan2, CAN_IT_RX_FIFO0_MSG_PENDING)!=HAL_OK)
    {
      Error_Handler();
    }
```

```

    //creating tasks
    assert_param(xTaskCreate(Deferred_Task, "Task for Deferred work", STACK_SIZE,
NULL, tskIDLE_PRIORITY + 3, &xDeferrefTaskHandle) == pdPASS);
    assert_param(xTaskCreate(WasteFullLED, "LED_Blink on low priority", STACK_SIZE,
NULL, tskIDLE_PRIORITY + 1, NULL) == pdPASS);

    //Data_transmitt= xQueueCreate(4,sizeof(float));


    //Starting Scheduler
    vTaskStartScheduler();

    /* USER CODE END 2 */

    /* Infinite loop */
    /* USER CODE BEGIN WHILE */

    while (1)
    {
      /* program execution should not come here */
    }
      /* USER CODE END WHILE */

      /* USER CODE BEGIN 3 */

    /* USER CODE END 3 */
  }

  /**
    * @brief System Clock Configuration
    * @retval None
    */
  void SystemClock_Config(void)
  {
```

```
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage
    */
    __HAL_RCC_PWR_CLK_ENABLE();
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /** Initializes the RCC Oscillators according to the specified parameters
    * in the RCC_OscInitTypeDef structure.
```

```
    */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
      Error_Handler();
    }

    /** Initializes the CPU, AHB and APB buses clocks
    */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
    {
      Error_Handler();
    }
  }

  /**
    * @brief CAN2 Initialization Function
    * @param None
    * @retval None
    */
  static void MX_CAN2_Init(void)
```

```
{

  /* USER CODE BEGIN CAN2_Init 0 */

  /* USER CODE END CAN2_Init 0 */

  /* USER CODE BEGIN CAN2_Init 1 */

  /* USER CODE END CAN2_Init 1 */
  hcan2.Instance = CAN2;
  hcan2.Init.Prescaler = 24;
  hcan2.Init.Mode = CAN_MODE_NORMAL;
  hcan2.Init.SyncJumpWidth = CAN_SJW_1TQ;
  hcan2.Init.TimeSeg1 = CAN_BS1_13TQ;
```

```
  hcan2.Init.TimeSeg2 = CAN_BS2_2TQ;
  hcan2.Init.TimeTriggeredMode = DISABLE;
  hcan2.Init.AutoBusOff = DISABLE;
  hcan2.Init.AutoWakeUp = DISABLE;
  hcan2.Init.AutoRetransmission = DISABLE;
  hcan2.Init.ReceiveFifoLocked = DISABLE;
  hcan2.Init.TransmitFifoPriority = DISABLE;
  if (HAL_CAN_Init(&hcan2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN CAN2_Init 2 */

  /* USER CODE END CAN2_Init 2 */

}

/**
  * @brief I2C1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_I2C1_Init(void)
{
  /* USER CODE BEGIN I2C1_Init 0 */

  /* USER CODE END I2C1_Init 0 */
```

```
    /* USER CODE BEGIN I2C1_Init 1 */

    /* USER CODE END I2C1_Init 1 */
    hi2c1.Instance = I2C1;
    hi2c1.Init.ClockSpeed = 100000;
    hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
    hi2c1.Init.OwnAddress1 = 0;
    hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
    hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
    hi2c1.Init.OwnAddress2 = 0;
    hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
    hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
    if (HAL_I2C_Init(&hi2c1) != HAL_OK)
    {
      Error_Handler();
    }
    /* USER CODE BEGIN I2C1_Init 2 */
```

```
    /* USER CODE END I2C1_Init 2 */

  }

  /**
    * @brief GPIO Initialization Function
    * @param None
    * @retval None
    */
  static void MX_GPIO_Init(void)
  {
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /*USER CODE BEGIN MX_GPIO_Init_1 */

    /*USER CODE END MX_GPIO_Init_1 */

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
```

```
    HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15,
  GPIO_PIN_RESET);

    /*Configure GPIO pins : PD12 PD13 PD14 PD15 */
    GPIO_InitStruct.Pin = GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

  /* USER CODE BEGIN MX_GPIO_Init_2 */

  /* USER CODE END MX_GPIO_Init_2 */
  }

  /* USER CODE BEGIN 4 */
  void LCD_Init(void)
  {
    lcd_init();
  }


  /**
```

```
    * @brief This function configures the CAN filter
    * @retval None
    */

  void CAN_Filter_Config(void)
  {
    CAN_FilterTypeDef can2_filter_init;
    can2_filter_init.FilterActivation = ENABLE;
    can2_filter_init.FilterBank = 14;
    can2_filter_init.FilterFIFOAssignment = CAN_RX_FIFO0;
    can2_filter_init.FilterIdHigh = 0x0000;
    can2_filter_init.FilterIdLow = 0x0000;
    can2_filter_init.FilterMaskIdHigh = 0x0000;
    can2_filter_init.FilterMaskIdLow = 0x0000;
    can2_filter_init.FilterMode = CAN_FILTERMODE_IDMASK;
    can2_filter_init.FilterScale = CAN_FILTERSCALE_32BIT;
    can2_filter_init.SlaveStartFilterBank=14;
```

```
    if(HAL_CAN_ConfigFilter(&hcan2, &can2_filter_init) != HAL_OK)
    {
    Error_Handler();
    }
  }



  /* The vTaskDelay function is used to create a delay,
   * allowing the RTOS to manage the task's timing. */
  void WasteFullLED(void *NotUsed)
  {
    while(1)
    {
      HAL_GPIO_WritePin(GPIOD, GPIO_PIN_15, SET);
      vTaskDelay(100/portTICK_PERIOD_MS);
      HAL_GPIO_WritePin(GPIOD, GPIO_PIN_15, RESET);
      vTaskDelay(100/portTICK_PERIOD_MS);
    }
  }

  /* 1->This task waits for a notification from an ISR using ulTaskNotifyTake,
   * which blocks the task until the notification is received.
   * 2->Once notified, it extracts two float values from RxData,
   * representing distances for the front and rear */
  void Deferred_Task(void *argument)
  {
```

```
    char buffer[20];

  while(1)
    {
      SEGGER_SYSVIEW_PrintfHost("Waiting for mque receive\n");

      ulTaskNotifyTake(pdTRUE,portMAX_DELAY);

      memcpy(&distance_front,&RxData[0],sizeof(float));
      memcpy(&distance_rear,&RxData[4],sizeof(float));

      SEGGER_SYSVIEW_PrintfHost("Distance Front: %d Rear:
  %d",distance_front,distance_rear);
```

```
    /* The task logs the distances and then calculates the minimum distance. */
    min_distance= fmin(distance_front,distance_front);

    /* In this part of the code, the buzzer is controlled based on the min_distance value.
     * The delay duration (period) is calculated by multiplying the min_distance by 5.
     * This means that the closer the object, the shorter the delay, resulting in faster buzzing. */
    if(min_distance < 30)
    {
      HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, SET);
      period=(5 * min_distance);
      vTaskDelay(period/portTICK_PERIOD_MS);
      HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, RESET);
      vTaskDelay(period/portTICK_PERIOD_MS);
    }
    /* else reset the buzzer */
    else
      HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, RESET);


    /* for segger system view */
    SEGGER_SYSVIEW_PrintfHost("Distance Front: %d Rear: %d",(int)distance_front,(int)distance_rear);
    lcd_clear();
    lcd_put_cur(0, 0);
    sprintf(buffer, "Front Distance:");
    lcd_send_string(buffer);

    /* printing front distance on LCD */
```

```
    lcd_put_cur(1, 0);
    sprintf(buffer, "%d cm",(int)distance_front);
    lcd_send_string(buffer);

    lcd_put_cur(2, 0);
    sprintf(buffer, "Rear Distance:");
    lcd_send_string(buffer);

    /* printing rear distance on LCD */
    lcd_put_cur(3, 0);
```

```
      sprintf(buffer, "%d cm",(int)distance_rear);
      lcd_send_string(buffer);

      bzero(buffer,sizeof(buffer));
      vTaskDelay(100/portTICK_PERIOD_MS);

    }

  }
   void
  CAN2_RX0_IRQHandler(void)

  {
      /* USER CODE BEGIN CAN2_RX0_IRQn 0 */

    /* xHigherPriorityTaskWoken is a flag used to check
     * if a higher priority task has been woken by an ISR. It is initialized to
  pdFALSE */
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* Format and send an ISR entry event.
     * This function is used to log the entry into the ISR using SEGGER SystemView,
     * a real-time analysis tool*/
    SEGGER_SYSVIEW_RecordEnterISR();

    /* Declares a structure to hold the header information of the received CAN
  message */
    CAN_RxHeaderTypeDef RxHeader;

    /* Reads the Interrupt Enable Register (IER) of the CAN peripheral
     *  to check which interrupts are enabled. */
    uint32_t interrupts = READ_REG(hcan2.Instance->IER);

    /* Checks if the FIFO0 message pending interrupt is enabled
     * by using a bitwise AND operation with CAN_IT_RX_FIFO0_MSG_PENDING */
```

```
    if((interrupts & CAN_IT_RX_FIFO0_MSG_PENDING)!=0U)
    {
      /* Checks if there are any messages pending in the FIFO0 receive buffer.
       * This is done by checking the FIFO0 Message Pending bits in the Receive
  FIFO 0 Register (RF0R) */
      if((hcan2.Instance->RF0R & CAN_RF0R_FMP0)!=0U)
      {
```

```
      /* Calls the HAL_CAN_GetRxMessage function to retrieve the message from FIFO0.
       * If it fails, the Error_Handler() function is called */
      if(HAL_CAN_GetRxMessage(&hcan2, CAN_RX_FIFO0, &RxHeader, RxData)!= HAL_OK )
      {
        Error_Handler();
      }
      /* This checks if the received message has a standard ID of 0x65D, is a standard frame (IDE == CAN_ID_STD)
       * and has a data length code (DLC) of 8. If these conditions are met, the code proceeds to process the data. */
      if((RxHeader.StdId==0x65D)&&(RxHeader.IDE==CAN_ID_STD)&&(RxHeader.DLC==8))
      {
        /* printing received data on segger for debug and user view */
        SEGGER_SYSVIEW_PrintfHost("In Interrupt\n");
        SEGGER_SYSVIEW_PrintfHost("Data: %c\n",RxData[0]);
        SEGGER_SYSVIEW_PrintfHost("Data: %c\n",RxData[4]);
        //xQueueSendFromISR(Data_transmitt, &RxData, &xHigherPriorityTaskWoken);

        /* This function is used to notify a task (referred to by xDeferrefTaskHandle) from within the ISR
         * The xHigherPriorityTaskWoken flag is set to pdTRUE if the notified task has a higher priority than the currently running task. */
        vTaskNotifyGiveFromISR(xDeferrefTaskHandle,&xHigherPriorityTaskWoken);
        SEGGER_SYSVIEW_PrintfHost("Data Sent\n");
      }
    }
  }

  SEGGER_SYSVIEW_RecordExitISR();

  /* f xHigherPriorityTaskWoken was set to pdTRUE, this macro will cause a context
   * switch to the higher priority task that was woken by the ISR,
   *  ensuring that it runs immediately after the ISR completes. */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

  /* USER CODE END CAN2_RX0_IRQn 0 */
```

```

  /* USER CODE BEGIN CAN2_RX0_IRQn 1 */

  /* USER CODE END CAN2_RX0_IRQn 1 */
```

```
}

/* USER CODE END 4 */

/**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM1 interrupt took place, inside
  * HAL_TIM_IRQHandler(). It makes a direct call to HAL_IncTick() to increment
  * a global variable "uwTick" used as application time base.
  * @param  htim : TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM1) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  /* USER CODE END Callback 1 */
}

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
```

```
  /**
    * @brief  Reports the name of the source file and the source line number
    *         where the assert_param error has occurred.
    * @param  file: pointer to the source file name
    * @param  line: assert_param error line source number
    * @retval None
    */
  void assert_failed(uint8_t *file, uint32_t line)
  {
    /* USER CODE BEGIN 6 */

    /* User can add his own implementation to report the file name and line number,

       ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */

    /* USER CODE END 6 */
  }
  #endif /* USE_FULL_ASSERT */

```

- **lcd_i2c.c**

```
  #include "lcd_i2c.h"

  extern I2C_HandleTypeDef hi2c1;
  #define SLAVE_ADDRESS_LCD 0x4E // Change this according to your setup

  void lcd_send_string(char *str)
  {
      while (*str) lcd_send_data(*str++);
  }

  void lcd_send_cmd(char cmd) {
      char data_u, data_l;
      uint8_t data_t[4];
      data_u = (cmd & 0xf0);
      data_l = ((cmd << 4) & 0xf0);
      data_t[0] = data_u | 0x0C;
      data_t[1] = data_u | 0x08;
      data_t[2] = data_l | 0x0C;
```

```
    data_t[3] = data_l | 0x08;

    HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDRESS_LCD, (uint8_t *)data_t, 4, 100);
}

void lcd_send_data(char data) {
    char data_u, data_l;
    uint8_t data_t[4];
    data_u = (data & 0xf0);
    data_l = ((data << 4) & 0xf0);
    data_t[0] = data_u | 0x0D;
    data_t[1] = data_u | 0x09;
    data_t[2] = data_l | 0x0D;
    data_t[3] = data_l | 0x09;
    HAL_I2C_Master_Transmit(&hi2c1, SLAVE_ADDRESS_LCD, (uint8_t *)data_t, 4, 100);
   // HAL_Delay(50);
}

void lcd_clear(void) {
    lcd_send_cmd(0x01);
    HAL_Delay(2);
}

void lcd_put_cur(int row, int col) {
    switch (row) {
        case 0:
            col |= 0x80;
            break;
        case 1:
            col |= 0xC0;
            break;
        case 2:
            col |= 0x90;
            break;
        case 3:
            col |= 0xD0;
            break;
    }
    lcd_send_cmd(col);
}
```

```
void lcd_init(void) {
    HAL_Delay(50);
    lcd_send_cmd(0x30);
    HAL_Delay(5);
    lcd_send_cmd(0x30);
    HAL_Delay(1);
    lcd_send_cmd(0x30);
    HAL_Delay(10);
    lcd_send_cmd(0x20);
    HAL_Delay(10);

    lcd_send_cmd(0x28);
    HAL_Delay(1);
    lcd_send_cmd(0x08);
    HAL_Delay(1);
    lcd_send_cmd(0x01);
    HAL_Delay(1);
    HAL_Delay(1);
    lcd_send_cmd(0x06);
    HAL_Delay(1);
    lcd_send_cmd(0x0C);
}
```

- **lcd_i2c.h**

```
#ifndef LCD_I2C_H
#define LCD_I2C_H

#include "stm32f4xx_hal.h"

void lcd_init(void);
void lcd_send_cmd(char cmd);
void lcd_send_data(char data);
void lcd_send_string(char *str);
void lcd_put_cur(int row, int col);
void lcd_clear(void);

#endif
```

9 OUTPUT

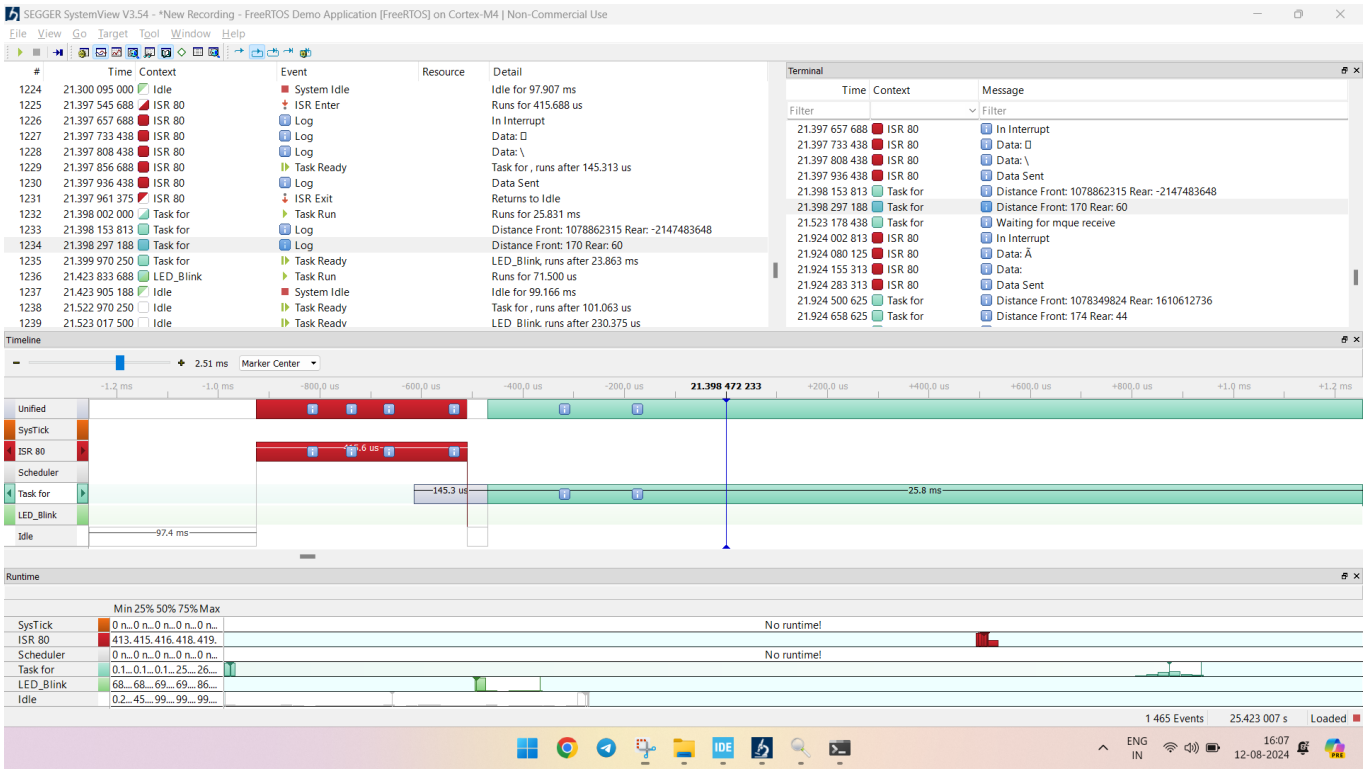


Figure8.2b Task running status on segger system view

Figure8.3: lcd view

10 Conclusion

The "CAN Shield Car Collision Avoidance System" successfully demonstrates the integration of advanced microcontroller technology, robust communication protocols, and real-time operating systems to enhance vehicle safety. Through the use of STM32F407VGT6 microcontrollers, MCP2551 CAN transceivers, and ultrasonic sensors, the system effectively monitors the distance between a vehicle and potential obstacles, providing realtime alerts through visual and auditory signals.

The project highlights the importance of using the CAN bus for reliable, fault-tolerant communication between multiple microcontrollers in a vehicle environment. The application of FreeRTOS on the receiving microcontroller ensures that the system can manage multiple tasks efficiently, responding promptly to changes in sensor data and maintaining consistent performance even under dynamic conditions.

Extensive testing and validation confirmed that the system operates reliably across various scenarios, accurately detecting obstacles and triggering alerts to prevent potential collisions. The integration of a 16x4 LCD for distance display and a buzzer for auditory warnings further enhances the system's usability and effectiveness in real-world applications.

In conclusion, this project not only fulfills its objective of creating a functional collision avoidance system but also lays the groundwork for future enhancements, such as the integration of additional sensors or the implementation of more advanced algorithms for obstacle detection. The success of this project demonstrates the feasibility and importance of incorporating such safety systems in modern vehicles, contributing to the broader goal of reducing road accidents and enhancing driver safety.

11 Future Scope

The "CAN Shield Car Collision Avoidance System" serves as a foundational step in the development of more advanced and comprehensive vehicle safety systems. The project's successful implementation opens several avenues for future enhancements and applications. Below are some potential directions for expanding and refining the system:

1. Integration with Additional Sensors

- **Radar and LiDAR Sensors**: Incorporating radar or LiDAR sensors could extend the system's range and accuracy, allowing for better detection of obstacles at greater distances and in varying environmental conditions, such as fog, rain, or darkness.
- **Camera Systems**: Adding camera-based vision systems could enable the identification of specific objects (e.g., pedestrians, cyclists) and provide more detailed environmental awareness, potentially integrating with advanced driver assistance systems (ADAS).

2. Enhanced Data Processing and Algorithms

- **Sensor Fusion**: Combining data from multiple sensor types (ultrasonic, radar, LiDAR, cameras) through sensor fusion algorithms can create a more comprehensive understanding of the vehicle's surroundings, improving the accuracy and reliability of obstacle detection.
- **Machine Learning**: Implementing machine learning algorithms could enhance the system's ability to predict potential collision scenarios based on patterns in sensor data, providing earlier and more accurate warnings to the driver.

3. Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) Communication

- **V2V Communication**: Expanding the system to include V2V communication would allow vehicles to share real-time data about their positions, speeds, and intentions, further enhancing collision avoidance capabilities, especially in scenarios where line-of-sight is obstructed.
- **V2I Communication**: Integrating with infrastructure, such as traffic signals or road hazard warnings, could enable the system to receive external alerts about potential dangers, allowing the vehicle to take preemptive actions.

12 References

1. **Bosch GmbH**. (1991). *CAN Specification Version 2.0*. Robert Bosch GmbH. Available at: Bosch CAN Specification
2. **STM32F407VGT6 Datasheet**. (n.d.). STMicroelectronics. Available at: STM32F407VGT6
3. **MCP2551 CAN Transceiver Datasheet**. (n.d.). Microchip Technology Inc. Available at: MCP2551 Datasheet
4. **FreeRTOS Documentation**. (n.d.). Real-Time Engineers Ltd. Available at: FreeRTOS
5. **HC-SR04 Ultrasonic Sensor Datasheet**. (n.d.). Available at: HC-SR04 Datasheet
6. **Motor Industry Software Reliability Association (MISRA)**. (2004). *MISRA-C: 2004, Guidelines for the use of the C language in critical systems*. MISRA.
7. **N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert**. (2005). *Trends in Automotive Communication Systems*. Proceedings of the IEEE, 93(6), 1204-1223.
8. **S. Schwarz and M. Obrecht**. (2010). *Sensor Fusion for Automotive Applications*. IEEE Transactions on Instrumentation and Measurement, 59(4), 946-957.
9. **B. P. Scaife and A. Dixit**. (2017). *Analysis of Real-Time Operating Systems for Automotive Applications*. Journal of Systems Architecture, 75, 69-82. 10. **N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell**. (2010). *A Survey of Mobile Phone Sensing*. IEEE Communications Magazine, 48(9), 140-150.
11. **L. Guzzella and A. Sciarretta**. (2007). *Vehicle Propulsion Systems: Introduction to Modeling and Optimization*. Springer Science & Business Media.
12. **K. Johansson, M. Törngren, and L. Nielsen**. (2005). *Vehicle Applications of Controller Area Network*. Handbook of Networked and Embedded Control Systems. Springer.
13. **M. Chui, M. Löhr, and R. H. Roberts**. (2018). *Automotive Radar and Lidar Systems for Next-Generation Advanced Driver Assistance Systems*. IEEE Sensors Journal, 18(12), 4892-4906.
14. **J. P. Hespanha, P. Naghshtabrizi, and Y. Xu**. (2007). *A Survey of Recent Results in Networked Control Systems*. Proceedings of the IEEE, 95(1), 138-162