

# **Data acquisition of real time CAN bus data from a car**

*A MINI PROJECT REPORT*

*submitted by*

**ANKITH SAMUEL ABRAHAM (MGP20URB011)**

**ANAKHA MOHAN (MGP20URB007)**

**DAVID TITUS JOHN (MGP20URB023)**

**BALAMONY BIMALRAJ (MGP20URB017)**

*in partial fulfilment of the requirements for the  
award of the degree of*

**BACHELOR OF TECHNOLOGY**



**DEPARTMENT OF ELECTRONICS ENGINEERING  
SAINTGITS COLLEGE OF ENGINEERING  
(AUTONOMOUS) KOTTAYAM**

**JULY 2023**

## DECLARATION

We hereby declare that the project report “DATA ACQUISITION OF REAL TIME CAN BUS DATA FROM A CAR”, submitted for the partial fulfillment of the requirements for the award of degree of Bachelor of Technology of the APJ Abdul Kalam Technological University, Kerala is a bonafide work done by us under supervision of Er. Prinu Chacko Philip. This submission represents our ideas in our own words and where ideas or words of others have been included. We have adequately and accurately cited and referenced the original sources. We also declare that we have adhered to ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in our submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not been previously formed the basis for the award of any degree, diploma or similar title of any other University.

1. ANKITH SAMUEL ABRAHAM
2. ANAKHA MOHAN
3. DAVID TITUS JOHN
4. BALAMONY BIMALRAJ

Place :- Pathamuttom

Date :-

**ELECTRONICS AND COMMUNICATION ENGINEERING  
SAINTGITS COLLEGE OF ENGINEERING (AUTONOMOUS)  
PATHAMUTTOM, KOTTAYAM, KERALA- 686532**



**CERTIFICATE**

This is to certify that the mini project report entitled “**DATA ACQUISITION OF REAL TIME CAN BUS DATA FROM A CAR**” submitted by **ANKITH SAMUEL ABRAHAM (MGP20URB011), ANAKHA MOHAN (MGP20URB007), DAVID TITUS JOHN (MGP20URB023) BALAMONY BIMALRAJ (MGP20URB017)**, to the APJ Abdul Kalam Technological University in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Electronics and Communication Engineering is a bonafide record of the project work carried out. This report in any form has not been submitted to any other University or Institute for any purpose.

**Er. Prinu Chacko Philip**

Project Guide

Assistant Professor

Dept. of Electronics Engineering

**Er. Ashwin P. V.**

Project Co-ordinator

Assistant Professor

Dept. of Electronics Engineering

**Dr. Sreekala K. S.**

Associate Professor & Head

Dept. of Electronics Engineering

Place: Pathamuttom

Date:

# ACKNOWLEDGEMENT

We express our profound gratitude to all those who have supported us. Our sincere thanks to:

**Er. Prinu Chacko Philip**, under whose inspiring guidance and supervision this academic work was carried out. His insistent attitude and timely help led to the successful completion of our Mini Project.

**Er. Ashwin P. V**, for providing timely instructions and fruitful suggestions as the Project Co-ordinator.

**Dr. Sreekala K. S.**, Head, Department of Electronics Engineering for providing all the departmental facilities.

**Prof. (Dr.) Sudha T.**, Dean, ECR, for her kind support and motivation.

**Prof. (Dr.) Josephkunju Paul C**, Principal, Saintgits College of Engineering (Autonomous) for providing an excellent ambiance that led to the completion of this work on time.

Our colleagues, all the staff and students of the Department of Electronics Engineering for rendering help and support.

Our parents, friends and well-wishers for their constant support and timely help.

Above all we thank the Almighty God for His Grace and Blessings that made it possible for us to complete this Mini Project successfully.

# ABSTRACT

**KEYWORD** :- CAN bus, Controller Area Network, Data acquisition, Real-time

CANBus networks provide a means of interconnecting various electronic control units (ECUs) within a vehicle, enabling efficient communication and control of critical functions. This project aims to develop a robust and versatile system for real-time data acquisition from the CANBus of cars, enabling researchers, engineers, and enthusiasts to analyze and interpret vehicle performance parameters, diagnostics, and sensor data. The proposed project leverages the existing infrastructure of the CANBus network in vehicles to extract valuable information that can contribute to a variety of applications such as vehicle performance analysis, fleet management, predictive maintenance, and automotive research. The primary focus lies on the development of a data acquisition system that captures, interprets, and logs the data transmitted over the CANBus network in real time.

To achieve this, the project involves the implementation of a hardware interface capable of connecting to the vehicle's CANBus system, adhering to the standardized protocols and electrical specifications. This interface will collect the raw data packets from the CANBus network and transmit them to a processing unit. The processing unit, powered by a microcontroller or a dedicated computer system, will decode and interpret the data packets, extracting relevant information from various ECUs. Furthermore, the project intends to develop a user-friendly software interface that enables users to visualize and analyze the collected data in real time. This interface will offer tools for generating comprehensive reports, conducting statistical analysis, and presenting visual representations of the acquired data, aiding researchers, engineers, and enthusiasts in gaining insights into vehicle performance, diagnostics, and behavior.

In conclusion, this project aims to develop a comprehensive system for real-time CANBus data acquisition from cars. By providing researchers, engineers, and enthusiasts with an accessible and customizable toolset, the project aims to empower the automotive community with the ability to extract valuable insights from the complex network of data transmitted over the CANBus.

# CONTENT

Content	PageNo.
<b>ACKNOWLEDGEMENT</b>	<b>i</b>
<b>ABSTRACT</b>	<b>ii</b>
<b>LIST OF FIGURES</b>	<b>v</b>
<b>1. INTRODUCTION</b>	<b>1</b>
<b>2. LITERATURE SURVEY</b>	<b>3</b>
2.1 In-vehicle data logging and analysis for real-world fuel consumption evaluation of hybrid electric vehicles.....	3
2.2 CAN data analysis for predictive vehicle maintenance.....	3
2.3 Data acquisition and analysis for automotive CAN bus network.....	3
2.4 CAN bus data analysis for intelligent driving.....	4
2.5 A Monitoring and analysis of CAN bus data for vehicle diagnostics and prognostics.....	4
<b>3. METHODOLOGY</b>	<b>5</b>
3.1 Challenges in obtaining CAN bus data.....	5
3.2 System model.....	6
3.3 Implementation.....	8
3.3.1 Hardware selection.....	8
3.3.2 Software development environment.....	8
3.3.3 Connect to CAN bus.....	8
3.3.4 CAN communication.....	8
3.3.5 Data processing.....	9
3.3.6 Data logging or transmission.....	9
3.3.7 Hardware used.....	9
3.3.8 Connection used.....	12

<b>4. SOFTWARE DESCRIPTION</b>	<b>14</b>
4.1 Software requirement.....	14
4.1.1 Arduino IDE.....	14
4.1.2 CANHACKER v2.00.01.....	14
4.2 System requirement.....	15
<b>5. EXPERIMENTAL ANALYSIS AND RESULTS</b>	<b>16</b>
5.1 Result.....	16
5.2 Performance evaluation.....	17
<b>6. CONCLUSION AND FUTURE SCOPE</b>	<b>19</b>
<b>REFERENCES</b>	<b>20</b>
<b>APPENDIX</b>	<b>21</b>

## **LIST OF FIGURES**

<b>No.</b>	<b>Title</b>	<b>PageNo.</b>
3.1	Circuit Diagram	6
3.2	3D model of components used	7
3.3	Pins of OBD-II port	12
5.1	Output obtained	16
5.2	Sample of CANBus data obtained	17



# **CHAPTER 1**

## **INTRODUCTION**

Data acquisition of real-time CAN (Controller Area Network) bus data from a car is a process that involves capturing and analyzing various parameters and messages exchanged between the electronic control units (ECUs) within the vehicle. The CAN bus is a widely used communication protocol in modern automobiles, enabling different components to communicate and coordinate their functions effectively. The primary purpose of acquiring real-time CAN bus data is to gain insights into the car's performance, diagnose issues, and develop advanced applications for vehicle monitoring, analysis, and control. By monitoring the data flowing through the CAN bus, it becomes possible to extract valuable information about engine performance, transmission, braking, suspension, and other crucial systems.

To acquire real-time CAN bus data, specialized hardware and software are employed. The hardware typically includes a device known as a CAN bus interface, which connects to the vehicle's OBD-II (On-Board Diagnostics) port or directly interfaces with the CAN bus network. This interface acts as a bridge between the vehicle's CAN bus and the data acquisition system. The acquisition of real-time CAN bus data from a car involves capturing, decoding, and analyzing the messages exchanged between ECUs. It enables valuable insights into the vehicle's performance and behavior, contributing to various aspects of automotive engineering, research, and development.

To collect CAN bus data from cars, the following steps can be followed:

1. Identify the Target Vehicle: Determine the make, model, and year of the vehicle from which you want to collect CAN bus data. Different vehicles may have variations in their CAN bus systems, so understanding the specific vehicle's architecture is essential.
2. Obtain the Necessary Hardware: Acquire the required hardware tools for data collection, such as a CAN bus adapter or data logger. Choose a device that is compatible with the vehicle's CAN bus system and offers features like message filtering, high data acquisition rates, and appropriate connectivity options.

3. Connect to the CAN Bus: Connect the hardware interface to the vehicle's CAN bus network. Locate the CAN bus connectors in the vehicle, which are typically found under the dashboard, near the OBD-II port, or in the engine compartment. Connect the hardware interface securely to the CAN bus network using the appropriate cables or connectors.

4. Configure the Hardware and Software: Install any necessary drivers or software for the hardware interface. Configure the hardware and software settings according to the specific requirements of the vehicle and the desired data collection parameters, such as message filters, baud rate, and data logging options.

5. Start Data Logging: Begin the data logging process by initiating the recording or monitoring function on the data acquisition software or hardware interface. This will enable the capture and storage of CAN bus messages transmitted on the network.

Applications of include steering control without human interference , direction control for autonomous vehicles,achieves less traffic accidents induced from human errors , achieve optimal turning radius, reduce oversteering and understeering.It iscloud based vehicle control.The collected data can be used for performance analysis, maintenance, and safety improvements.

## **CHAPTER 2**

### **LITERATURE SURVEY**

#### **2.1 In-vehicle data logging and analysis for real-world fuel consumption evaluation of hybrid electric vehicles**

"In-vehicle data logging and analysis for real-world fuel consumption evaluation of hybrid electric vehicles"

Authors: X. Jiao, J. Mi, G. Yin, Z. Zhang

Published: 2017

This paper focuses on collecting CAN bus data from hybrid electric vehicles for real-world fuel consumption evaluation. It discusses the data collection methodology, including selecting relevant parameters, hardware setup, and data analysis techniques.

#### **2.2 CAN data analysis for predictive vehicle maintenance**

"CAN data analysis for predictive vehicle maintenance"

Authors: C. Deluca, L. Giarré, D. La Rosa, G. Pau, S. Ranieri

Published: 2018

This paper presents a study on utilizing CAN bus data for predictive vehicle maintenance. It discusses the challenges of collecting and processing the data, identifies key performance indicators for maintenance prediction, and proposes a methodology for analyzing the collected data.

#### **2.3 Data acquisition and analysis for automotive CAN bus network**

"Data acquisition and analysis for automotive CAN bus network"

Authors: R. Pachauri, V. Gupta

Published: 2019

This research paper focuses on the data acquisition and analysis of an automotive CAN bus network. It discusses the hardware setup for data acquisition, data preprocessing techniques, and the application of machine learning algorithms for fault detection and classification.

## **2.4 CAN bus data analysis for intelligent driving**

"CAN bus data analysis for intelligent driving"

Authors: X. Chen, H. Zhang, J. Guo

Published: 2019

This paper presents a study on CAN bus data analysis for intelligent driving applications. It discusses the data collection process, data preprocessing techniques, and the application of machine learning algorithms for tasks such as driver behavior analysis and intelligent decision-making.

## **2.5 Monitoring and analysis of CAN bus data for vehicle diagnostics and prognostics**

"Monitoring and analysis of CAN bus data for vehicle diagnostics and prognostics"

Authors: A. S. Putrus, M. H. Yaqub, S. Z. Rahman, M. A. Aslam, R. K. Aggarwal

Published: 2018

This research paper explores the collection and analysis of CAN bus data for vehicle diagnostics and prognostics. It discusses the selection of relevant parameters, data acquisition methods, and the application of statistical and machine learning techniques for fault detection and prognostics.

These papers provide a starting point for understanding the methodologies, techniques, and challenges associated with collecting and analyzing CAN bus data from cars. They cover various aspects, including data acquisition, parameter selection, data preprocessing, and the application of statistical and machine learning methods for different automotive applications.

## **CHAPTER 3**

### **METHODOLOGY**

#### **3.1 CHALLENGES IN OBTAINING CANBUS DATA**

Obtaining CAN bus data can pose several challenges, including:

1. **Access and Permissions:** Accessing CAN bus data from vehicles typically requires specialized hardware interfaces or diagnostic tools. Obtaining the necessary permissions and authorizations to access the CAN bus can be challenging, especially in commercial or regulated environments.
2. **Vehicle Compatibility:** Different vehicle makes and models may have variations in their CAN bus systems, such as different protocols, baud rates, or message formats. Ensuring compatibility between the data collection tools and the targeted vehicles can be a challenge, as it may require specific hardware or software configurations.
3. **Data Filtering and Interpretation:** The CAN bus generates a large volume of data, including various types of messages from different modules within the vehicle. Filtering and interpreting the relevant data among the vast stream can be challenging, requiring proper identification of relevant messages and parameters.
4. **Data Synchronization:** When collecting CAN bus data from multiple sources or integrating it with other data sources, ensuring proper synchronization of timestamps or data streams can be challenging. Achieving accurate synchronization is important for meaningful analysis and correlation with other data sources.
5. **Privacy and Legal Considerations:** Collecting CAN bus data may involve personal or sensitive information, such as vehicle location, driver behavior, or diagnostics. Respecting privacy laws and regulations and obtaining proper consent or permissions is crucial when handling such data.
6. **Real-time Data Collection:** Capturing real-time CAN bus data can be challenging due to the need for high-speed data acquisition and processing capabilities. Timing constraints and the need for continuous data logging can pose technical challenges, especially for applications that require real-time monitoring or control.
7. **Data Volume and Storage:** The continuous stream of CAN bus data can result in a large volume of data, requiring adequate storage capacity and efficient data management strategies. Handling, storing, and processing the large amount of data collected from multiple vehicles or over extended periods can be a logistical challenge. Addressing these

challenges often requires a combination of technical expertise, appropriate hardware and software tools, and compliance with legal and ethical considerations.

### 3.2 SYSTEM MODEL

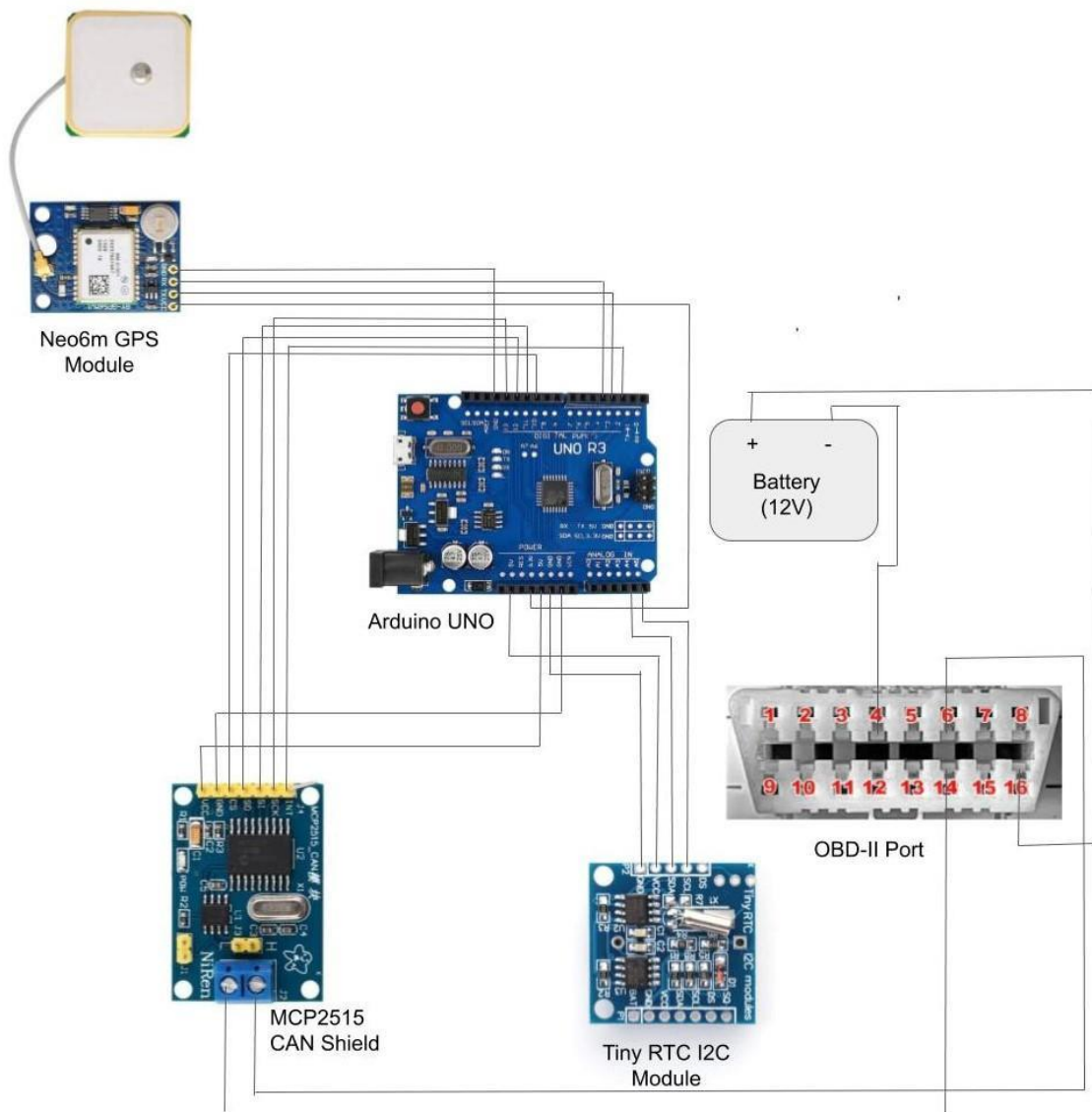


Fig. 3.1: Circuit Diagram

Fig. 4.1 shows the circuit diagram for the hardware part of the project for obtaining CANBus data from cars along with real time and location data. The circuit is connected with the OBD-II port in the cars through which we can access the CANBus data. In OBD-II port pin 6 is

the CANHigh pin and pin 14 is the CANLow pin. The connections are made accordingly and the code is uploaded to the arduino uno board with the pins to be used are mentioned in the code accordingly as the circuit diagram. The RX/TX pins or MISO/MOSI pins should be mentioned carefully in the program before running the code. VCC and GND should be properly connected according to the voltage and current needs of each module.

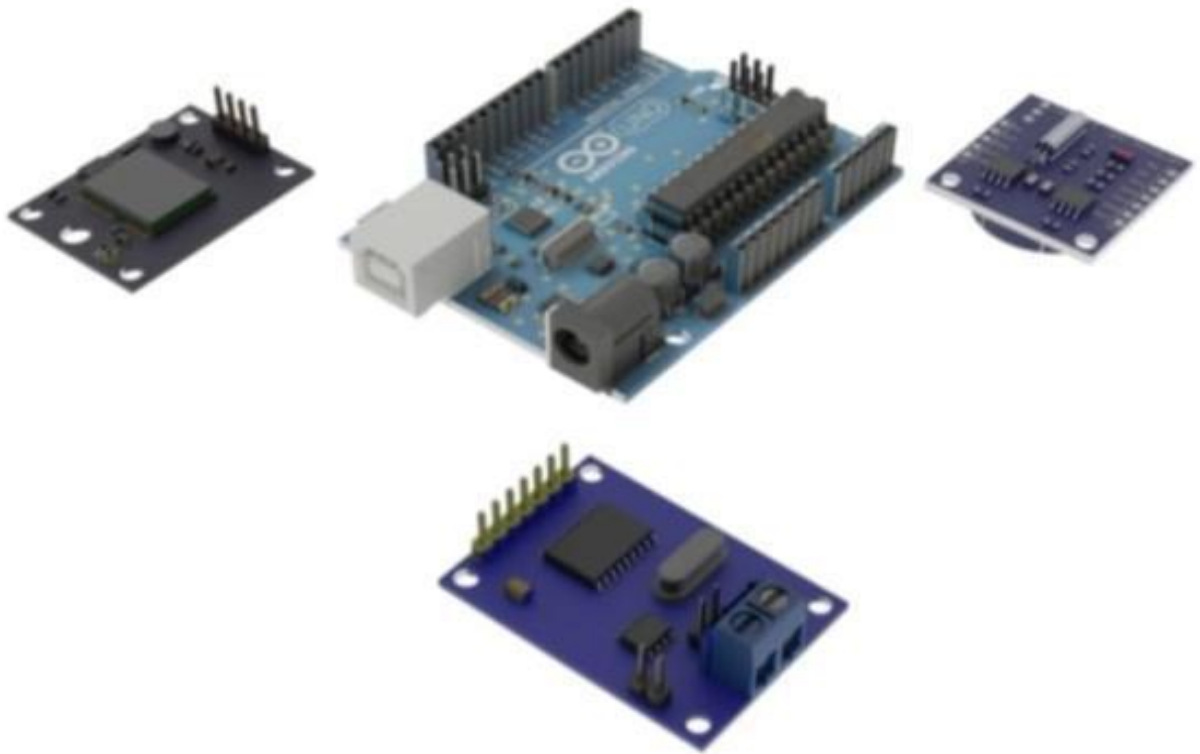


Fig. 3.2: 3D Model of components used

## **3.3 IMPLEMENTATION**

Implementing data acquisition from a real-time CANbus in cars involves a combination of hardware and software components. Here's a high-level overview of the steps involved in the implementation process:

### **3.3.1. Hardware Selection:**

- Choose a suitable hardware interface for connecting to the CANbus in the car. Common options include USB-to-CAN adapters or CAN interface modules.
- Ensure that the hardware supports the necessary communication protocols and has drivers compatible with your development environment.

### **3.3.2. Software Development Environment:**

- Set up a development environment on your computer, including an integrated development environment (IDE) and the required libraries for CAN communication.
- Install the necessary drivers and software for the chosen hardware interface.

### **3.3.3. Connect to the CANbus:**

- Connect the chosen hardware interface to the car's CANbus system, following the manufacturer's instructions.
- Ensure proper electrical isolation and protection to prevent damage to the hardware or the car's electronics.
- Power up the hardware interface and establish a connection to the car's CANbus.

### **3.3.4. CAN Communication:**

- Utilize the provided libraries or APIs to establish a connection with the hardware interface from your software.
- Configure the CAN communication parameters, such as the CAN channel, bitrate, and



message filters.

- Implement the necessary functions to send and receive CAN messages.

### **3.3.5. Data Processing:**

- Define the specific data that you want to extract from the CANbus, such as vehicle speed, engine RPM, or sensor readings.
- Implement data processing algorithms to interpret the received CAN messages and extract the desired information.
- Apply any necessary scaling or conversion operations to obtain meaningful values from the raw data.

### **3.3.6. Data Logging or Transmission:**

- Decide on the desired data storage or transmission method.
- Implement mechanisms to log the acquired data to a local file or a database for further analysis.
- Optionally, implement real-time data streaming or transmission to a remote server or cloud platform.

Remember that implementing data acquisition from a real-time CANbus requires expertise in both hardware and software development, as well as a good understanding of the CAN protocol and the specific data you want to extract. It's important to prioritize safety and follow best practices to prevent any potential damage to the car's electronics or compromise the system's reliability.

### **3.3.7. Hardwares used:**

#### **1) Arduino UNO:**

The Arduino Uno is a popular microcontroller board based on the ATmega328P microcontroller. It is widely used in prototyping and DIY electronics projects due to its simplicity and versatility.

#### **2) Neo6m GPS Module:**

The NEO-6M GPS module is a popular and widely used GPS (Global Positioning System) module that provides accurate positioning and timing information. It is commonly used in various applications such as navigation systems, vehicle tracking, drones, and other projects that require precise location data.

### **3) DS1307 RTC Module:**

The DS1307 is a popular Real-Time Clock (RTC) module developed by Maxim Integrated. The RTC module is commonly used in electronic projects where accurate timekeeping is required, such as in clocks, data loggers, and other time-sensitive applications.

### **4) MCP2515 CANShield module:**

The MCP2515 module is a popular integrated circuit (IC) that serves as a standalone controller area network (CAN) controller. CAN is a communication protocol commonly used in automotive and industrial applications for reliable and efficient data transfer between electronic devices.

The MCP2515 module, based on the MCP2515 IC, provides a convenient way to interface with a CAN bus using microcontrollers or other devices. It offers features such as message filtering, message buffering, and baud rate configuration. The module typically includes a CAN transceiver, which handles the physical layer of the CAN bus communication.

Here are some key features and specifications of the MCP2515 module:

**SPI Interface:** The module communicates with the host device (e.g., microcontroller) using the Serial Peripheral Interface (SPI) protocol. It uses SPI to send and receive data, control the MCP2515 IC, and configure various parameters.

**Baud Rate Configuration:** The MCP2515 supports baud rates ranging from 5 kbps to 1 Mbps, allowing flexibility in setting the communication speed according to the specific application requirements.

**Message Filtering and Masking:** The module supports both standard (11-bit identifier) and extended (29-bit identifier) frames. It provides configurable acceptance filters and masks, enabling the device to receive only the relevant messages on the CAN bus.

**Transmit and Receive Buffers:** The MCP2515 module has transmit and receive buffers that store CAN messages. This allows the host device to send and receive messages asynchronously, buffering them when necessary.

**Error Detection and Handling:** The MCP2515 includes error detection mechanisms, such as

bit error detection, frame error detection, and error counters. It also provides error interrupt functionality to notify the host device of error conditions.

**Wake-Up and Sleep Modes:** The module supports wake-up and sleep modes to conserve power when the CAN bus is idle or during low-power operation.

**Voltage Compatibility:** The MCP2515 module typically operates at a voltage level of 5V, although some versions may be designed for 3.3V operation. It's important to check the specifications and requirements of the specific module you are using.

## **5) OBD-II port:**

The OBD-II (On-Board Diagnostics II) port, also known as the OBD2 port, is a standardized diagnostic port found in most vehicles manufactured from the mid-1990s onwards. It is typically located under the dashboard, usually on the driver's side.

The OBD-II port serves as a connection point for diagnostic tools and devices to access and retrieve information from a vehicle's onboard computer system. It provides a standardized interface for communicating with the various sensors, modules, and systems within the vehicle.

By plugging a compatible diagnostic tool into the OBD-II port, technicians, mechanics, and even vehicle owners can retrieve valuable diagnostic information. This information includes engine performance data, emission levels, sensor readings, trouble codes, and other parameters that can help diagnose and troubleshoot issues with the vehicle.

The OBD-II port uses a standard 16-pin connector, and the communication protocol employed is also standardized across vehicles, known as the OBD-II protocol. This allows for compatibility and interoperability among different makes and models of vehicles.

It's worth noting that while the OBD-II port provides access to a wealth of diagnostic information, it does not control or modify the vehicle's systems directly. It primarily serves as a monitoring and diagnostic interface.

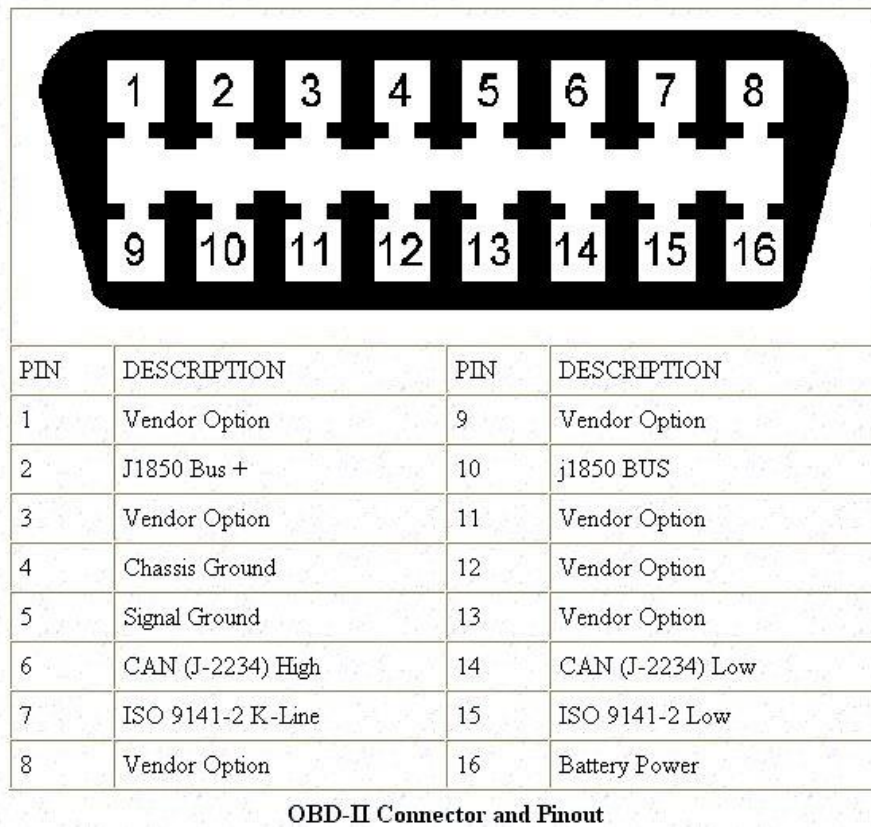


Fig. 3.3: Pins of OBD-II port.

### 3.3.8. Connections used:

#### 1) MCP2515 CAN module:

VCC to 5V  
 GND to GND  
 CS to Arduino pin 10  
 SO (MISO) to Arduino pin 12  
 SI (MOSI) to Arduino pin 11  
 SCK to Arduino pin 13  
 INT to Arduino pin 2

#### 2) DS1307 RTC module:

VCC to 5V  
 GND to GND  
 SDA to Arduino A4 (SDA)  
 SCL to Arduino A5 (SCL)

**3) GPS module:**

VCC to 5V

GND to GND

RX to Arduino pin 4 (TX)

TX to Arduino pin 3 (RX)

## **CHAPTER 4**

### **SOFTWARE DESCRIPTION**

#### **4.1 SOFTWARE REQUIREMENTS**

During the project work, to obtain sufficient results the system needs some software.

Software description gives the details about the softwares that is used.

##### **4.1.1 Arduino IDE:**

The Arduino Integrated Development Environment (IDE) is a software platform that provides a user-friendly interface for programming Arduino boards. It is a versatile and beginner-friendly tool that simplifies the process of writing, compiling, and uploading code to Arduino microcontrollers. The Arduino IDE supports the Arduino programming language, which is based on C/C++, making it accessible to both novice and experienced programmers. With its intuitive interface and a rich library of pre-built functions, the Arduino IDE enables users to quickly prototype and develop a wide range of projects, from simple electronics experiments to complex embedded systems. It also offers features like syntax highlighting, code autocompletion, and a serial monitor for debugging and interacting with the Arduino board. The Arduino IDE's open-source nature and active community make it a popular choice among hobbyists, students, and professionals working on various electronics and robotics projects.

##### **4.1.2 CANHacker v2.00.01**

CANHacker v2.00.01 is a software tool specifically designed for analyzing and monitoring Controller Area Network (CAN) bus data. It provides a user-friendly interface that allows users to capture and interpret CAN bus messages transmitted between different devices in a network. With CANHacker, users can view real-time data streams, analyze message frames, and gain insights into the communication patterns of CAN bus systems. The software supports various CAN interfaces, enabling seamless integration with different hardware devices. Additionally, CANHacker v2.00.01 offers features like filtering and sorting options, allowing users to focus on specific messages of interest. CANHacker is a valuable tool for automotive engineers, researchers, and enthusiasts working with CAN bus networks, as it simplifies the process of monitoring and understanding the data flow

in a CAN system.

## **4.2 SYSTEM REQUIREMENT**

Processor: Intel core i5 or above.

64-bit, quad-core, 2.5 GHz minimum per core

Ram: 4 GB or more

Hard disk: 10 GB of available space or more.

Display: Dual XGA (1024 x 768) or higher resolution monitors

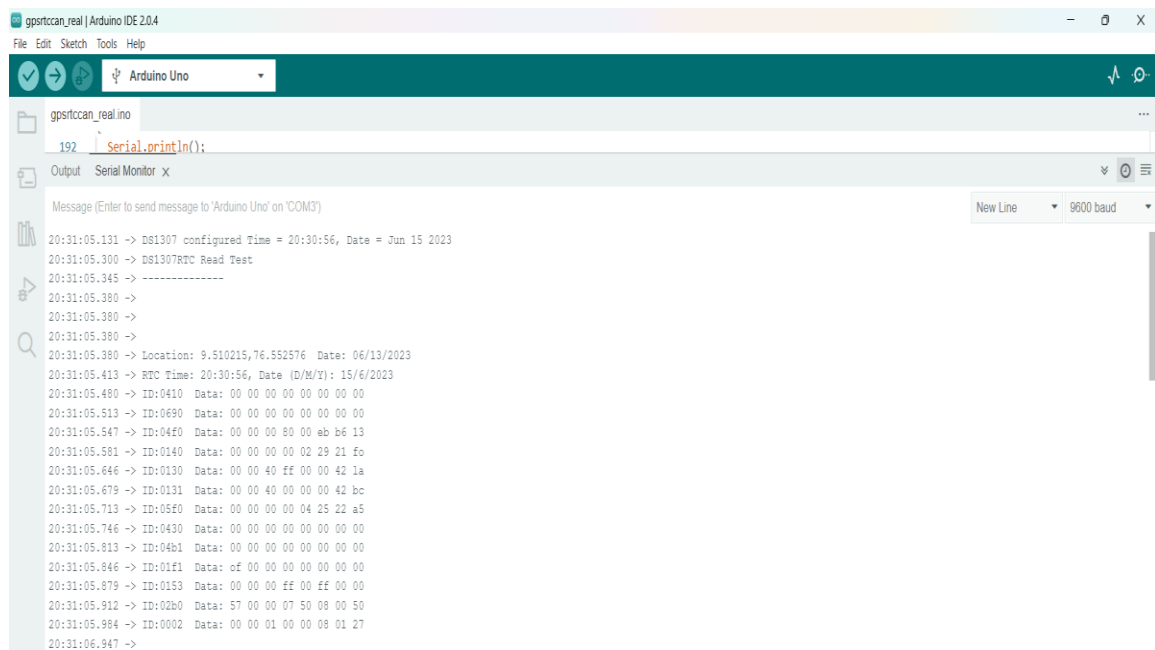
Operating system: Windows

# CHAPTER 5

## EXPERIMENTAL ANALYSIS AND RESULTS

### 5.1 RESULT

After much research and studies, we were able to incorporate the Neo6m GPS Module, DS1307 RTC Module and MCP2515 CANShield module together to obtain CANBus data from cars along with the GPS signals and real time data. From these data we can find out various CANBus data like steering angle, speed, rpm, brake status, etc and we can also determine at which instant and particular location these data are collected. These data can be incorporated with the video signals obtained and all these data together can be used for development in the field of autonomous vehicles.



```
gpsrtccan_realino | Arduino IDE 2.0.4
File Edit Sketch Tools Help
Arduino Uno
gpsrtccan_realino
192 Serial.println();
Output Serial Monitor x
Message (Enter to send message to 'Arduino Uno' on 'COM3')
New Line 9600 baud
20:31:05.131 -> DS1307 configured Time = 20:30:56, Date = Jun 15 2023
20:31:05.300 -> DS1307RTC Read Test
20:31:05.345 -> -----
20:31:05.380 ->
20:31:05.380 ->
20:31:05.380 ->
20:31:05.380 -> Location: 9.510215,76.552576 Date: 06/13/2023
20:31:05.413 -> RTC Time: 20:30:56, Date (D/M/Y): 15/6/2023
20:31:05.480 -> ID:0410 Data: 00 00 00 00 00 00 00 00
20:31:05.513 -> ID:0690 Data: 00 00 00 00 00 00 00 00
20:31:05.547 -> ID:04f0 Data: 00 00 00 80 00 ab b6 13
20:31:05.581 -> ID:0140 Data: 00 00 00 00 02 29 21 fo
20:31:05.646 -> ID:0130 Data: 00 00 40 ff 00 00 42 1a
20:31:05.679 -> ID:0131 Data: 00 00 40 00 00 00 42 bc
20:31:05.713 -> ID:05f0 Data: 00 00 00 00 04 25 22 a5
20:31:05.746 -> ID:0430 Data: 00 00 00 00 00 00 00 00
20:31:05.813 -> ID:04b1 Data: 00 00 00 00 00 00 00 00
20:31:05.846 -> ID:01f1 Data: 0f 00 00 00 00 00 00 00
20:31:05.879 -> ID:0153 Data: 00 00 00 ff 00 ff 00 00
20:31:05.912 -> ID:02b0 Data: 57 00 00 07 50 08 00 50
20:31:05.984 -> ID:0002 Data: 00 00 01 00 00 08 01 27
20:31:06.947 ->
```

Fig. 5.1: Output obtained



## 5.2 PERFORMANCE EVALUATION

After completing the circuit as needed and doing the code as we needed we were able to obtain the output as shown in the fig 7.5 where we can see the gps location, time and the CANBus data at that particular time and that particular location of the car. These data can be recorded and can be used in further development of the project in order to train the AI for the future development in the field of automation of vehicles.

ID	DLC	DATA	Timestamp
0410	8	00 00 00 00 00 00 00 00	2.116588
0690	8	00 00 00 00 00 00 00 00	2.135407
04f0	8	00 00 00 80 00 eb b6 13	2.144996
0140	8	00 00 00 00 02 29 21 fo	2.158382
0130	8	00 00 40 ff 00 00 42 la	2.176891
0131	8	00 00 40 00 00 00 42 bc	2.177941
05f0	2	00 00 00 00 04 25 22 a5	2.184367
0430	8	00 00 00 00 00 00 00 00	2.206446
04b1	8	00 00 00 00 00 00 00 00	2.20751
01f1	8	of 00 00 00 00 00 00 00	2.207884
0153	8	00 00 00 ff 00 ff 00 00	2.208233
02b0	5	57 00 00 07 50 08 00 50	2.210872
0002	8	00 00 01 00 00 08 01 27	2.216572

Fig. 5.2: Sample of CANBus data obtained.

The above data is obtained from the CANHacker v2.00.01 software. the obtained code is in hexadecimal format and it can be decoded to obtain data which can be understood easily by

humans. The various terms in the obtained data are:

**ID:** ID refers to the Identifier of a CAN message. The Identifier is a unique value assigned to each CAN message to distinguish it from other messages on the CAN bus. The ID field is a fundamental component of a CAN message frame and plays a crucial role in determining message priority and filtering.

**DLC:** In CAN bus data, "DLC" stands for Data Length Code. DLC is a field within the CAN message frame that indicates the number of data bytes contained in the message payload. It specifies the length of the data field and can range from 0 to 8 bytes.

**DATA:** In CAN bus data, the term "data" refers to the actual payload or information contained within a CAN message. It represents the meaningful content that is being transmitted between different devices on the CAN bus. The data field in a CAN message typically consists of 0 to 8 bytes, where each byte can hold 8 bits of information. The data can represent various types of information such as sensor readings, control commands, status updates, or any other relevant data specific to the application.

**TIMESTAMP:** In CAN bus data, the "timestamp" refers to the recorded time at which a particular CAN message was received or transmitted. It indicates the exact moment when the message was captured or generated by a CAN node.

## **CHAPTER 6**

### **CONCLUSION AND FUTURE SCOPE**

In conclusion, a project focused on collecting CAN bus data from cars can provide valuable insights into vehicle functioning and enable various applications such as diagnostics, performance monitoring, and predictive maintenance. By capturing and analyzing the data, researchers and engineers can gain a deeper understanding of vehicle systems, monitor sensor readings, detect anomalies, and make data-driven decisions.

The future scope of projects on collecting CAN bus data from cars is promising. It includes areas such as data analysis and machine learning for predictive maintenance and performance optimization. Additionally, there is a need to address cybersecurity challenges and develop intrusion detection systems to safeguard vehicle networks. The collected data can also be leveraged for autonomous driving, advanced driver assistance systems (ADAS), fleet management, telematics, and real-time monitoring solutions. Overall, these advancements can enhance vehicle safety, efficiency, and overall performance.

CAN bus data plays a crucial role in automating cars in the future. By capturing and analyzing the data transmitted on the CAN bus, advanced algorithms and artificial intelligence can make real-time decisions for autonomous driving. The data from various vehicle sensors, such as speed, steering angle, brake status, and engine RPM, can be processed to create a comprehensive understanding of the vehicle's surroundings. This information enables autonomous vehicles to navigate, detect obstacles, plan routes, and make intelligent decisions. Moreover, by integrating CAN bus data with machine learning models, the vehicles can continuously improve their driving capabilities, adapting to changing road conditions and enhancing safety and efficiency. CAN bus data thus serves as a fundamental component in the development and operation of self-driving cars in the future.

## REFERENCES

1. "A Practical Guide to CAN Bus Analysis and Logging" by Wilfried Voss: This book provides a comprehensive introduction to CAN bus systems, covering topics such as hardware interfaces, communication protocols, message structure, and data logging techniques. It also includes practical examples and case studies.
2. "Introduction to the Controller Area Network (CAN)" by Bosch: This technical document, published by Bosch, offers an in-depth introduction to the CAN bus protocol. It covers the basics of CAN communication, message structure, error handling, and message filtering. It also provides insights into analyzing CAN bus data.
3. GitHub repositories: There are several open-source projects on GitHub that provide code examples, libraries, and tools for collecting CAN bus data. Some popular repositories include:  
"SocketCAN" (<https://github.com/linux-can/can-utils>)  
"python-can" (<https://github.com/hardbyte/python-can>)  
which offer interfaces and utilities for working with CAN bus data in various programming languages

# APPENDIX

## 7.1 CODE

- 1) Code for printing data from GPS module, RTC module and CANBus on the serial monitor of Arduino IDE

```
#include <TinyGPS++.h>
#include <SoftwareSerial.h>
#include <Wire.h>
#include <Time.h>
#include <DS1307RTC.h>

#include <can.h>
#include <mcp2515.h>

#include <CanHacker.h>
#include <CanHackerLineReader.h>
#include <lib.h>

#include <SPI.h>

const char *monthName[12] = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};

tmElements_t tm;

static const int RXPin = 4, TXPin = 3; //gps pin
static const uint32_t GPSBaud = 9600;

TinyGPSPlus gps;
```

```

SoftwareSerial ss(RXPin, TXPin);

const int SPI_CS_PIN = 10;
const int INT_PIN = 2;

const int SS_RX_PIN = 12;
const int SS_TX_PIN = 11;

CanHackerLineReader *lineReader = NULL;
CanHacker *canHacker = NULL;

SoftwareSerial softwareSerial(SS_RX_PIN, SS_TX_PIN); //rtc

void setup()
{
    Serial.begin(9600);
    ss.begin(GPSBaud);

    bool parse = false;
    bool config = false;

    if (getDate(__DATE__) && getTime(__TIME__)) {
        parse = true;
        if (RTC.write(tm)) {
            config = true;
        }
    }

    while (!Serial) ; // wait for Arduino Serial Monitor
    delay(200);
    if (parse && config) {
        Serial.print("DS1307 configured Time = ");
        Serial.print(__TIME__);
        Serial.print(", Date = ");

```

```

Serial.println(__DATE__);
} else if (parse) {
Serial.println("DS1307 Communication Error :-{");
Serial.println("Please check your circuitry");
} else {
Serial.print("Could not parse info from the compiler, Time = \");
Serial.print(__TIME__);
Serial.print("\", Date = \");
Serial.print(__DATE__);
Serial.println("\");
}

while (!Serial) ; // wait for serial
delay(200);
Serial.println("DS1307RTC Read Test");
Serial.println("-----");
Serial.println();

Serial.begin(9600);
while (!Serial);
SPI.begin();
softwareSerial.begin(9600);

Stream *interfaceStream = &Serial;
Stream *debugStream = &softwareSerial;

canHacker = new CanHacker(interfaceStream, debugStream, SPI_CS_PIN);
//canHacker->enableLoopback(); // uncomment this for loopback
lineReader = new CanHackerLineReader(canHacker);

pinMode(INT_PIN, INPUT);
}

void loop()

```

```

{
    updateGPS();
    displayGPSLocation();
    displayRTCData();

    CanHacker::ERROR error;

    if (digitalRead(INT_PIN) == LOW) {
        error = canHacker->processInterrupt();
        handleError(error);
    }

    error = lineReader->process();
    handleError(error);

    if (canHacker->receiveCanFrame(NULL)) {
        struct can_frame frame;
        canHacker->receiveCanFrame(&frame);
        Serial.print("ID: ");
        Serial.print(frame.can_id, HEX);
        Serial.print(" Data: ");
        for (int i = 0; i < frame.can_dlc; i++) {
            Serial.print(frame.data[i], HEX);
            Serial.print(" ");
        }
        Serial.println();
    }

    delay(1000);
}

void updateGPS()
{
    while (ss.available() > 0)

```



```

{
  if (gps.encode(ss.read()))
  {
    // GPS data is processed
  }
}

if (millis() > 300000 && gps.charsProcessed() < 10)
{
  Serial.println("No GPS detected: check wiring.");
  while (true);
}
}

void displayGPSLocation()
{
  Serial.println();
  Serial.print("Location: ");
  if (gps.location.isValid())
  {
    Serial.print(gps.location.lat(), 6);
    Serial.print(",");
    Serial.print(gps.location.lng(), 6);
  }
  else
  {
    Serial.print("INVALID");
  }

  Serial.print(" Date: ");
  if (gps.date.isValid())
  {
    Serial.print(gps.date.month());
    Serial.print("/");

```

```

    Serial.print(gps.date.day());
    Serial.print("/");
    Serial.print(gps.date.year());
}
else
{
    Serial.print("INVALID");
}
Serial.println();
}

void displayRTCDData()
{
    tmElements_t rtcTime;
    if (RTC.read(rtcTime))
    {
        Serial.print("RTC Time: ");
        print2digits(rtcTime.Hour);
        Serial.write(':');
        print2digits(rtcTime.Minute);
        Serial.write(':');
        print2digits(rtcTime.Second);
        Serial.print(", Date (D/M/Y): ");
        Serial.print(rtcTime.Day);
        Serial.write('/');
        Serial.print(rtcTime.Month);
        Serial.write('/');
        Serial.print(tmYearToCalendar(rtcTime.Year));
        Serial.println();
    }
    else
    {
        if (RTC.chipPresent())
        {

```

```
    Serial.println("The DS1307 is stopped. Please run the SetTime example to initialize the  
time and begin running.");
```

```
    Serial.println();  
}  
else  
{  
    Serial.println("DS1307 read error! Please check the circuitry.");  
    Serial.println();  
}  
delay(9000);  
}  
}
```

```
void print2digits(int number)  
{  
    if (number >= 0 && number < 10)  
    {  
        Serial.write('0');  
    }  
    Serial.print(number);  
}
```

```
bool getTime(const char *str)  
{  
    int Hour, Min, Sec;  
  
    if (sscanf(str, "%d:%d:%d", &Hour, &Min, &Sec) != 3) return false;  
    tm.Hour = Hour;  
    tm.Minute = Min;  
    tm.Second = Sec;  
    return true;  
}
```

```
bool getDate(const char *str)
```

```

{
    char Month[12];
    int Day, Year;
    uint8_t monthIndex;

    if (sscanf(str, "%s %d %d", Month, &Day, &Year) != 3) return false;
    for (monthIndex = 0; monthIndex < 12; monthIndex++)
    {
        if (strcmp(Month, monthName[monthIndex]) == 0) break;
    }
    if (monthIndex >= 12) return false;
    tm.Day = Day;
    tm.Month = monthIndex + 1;
    tm.Year = CalendarYrToTm(Year);
    return true;
}

void handleError(const CanHacker::ERROR error) {
    switch (error) {
        case CanHacker::ERROR_OK:
        case CanHacker::ERROR_UNKNOWN_COMMAND:
        case CanHacker::ERROR_NOT_CONNECTED:
        case CanHacker::ERROR_MCP2515_ERRIF:
        case CanHacker::ERROR_INVALID_COMMAND:
            return;

        default:
            break;
    }

    softwareSerial.print("Failure (code ");
    softwareSerial.print((int)error);
    softwareSerial.println(")");
}

```

```

digitalWrite(SPI_CS_PIN, HIGH);
pinMode(LED_BUILTIN, OUTPUT);

while (1) {
    int c = (int)error;
    for (int i = 0; i < c; i++) {
        digitalWrite(LED_BUILTIN, HIGH);
        delay(500);
        digitalWrite(LED_BUILTIN, LOW);
        delay(500);
    }

    delay(2000);
}
}

```

## 2) Code for getting CANBus data on the CANHacker software

```

#include <can.h>
#include <mcp2515.h>

#include <CanHacker.h>
#include <CanHackerLineReader.h>
#include <lib.h>

#include <SPI.h>
#include <SoftwareSerial.h>

const int SPI_CS_PIN = 10;
const int INT_PIN = 2;

const int SS_RX_PIN = 12;
const int SS_TX_PIN = 11;

```

```

CanHackerLineReader *lineReader = NULL;
CanHacker *canHacker = NULL;

SoftwareSerial softwareSerial(SS_RX_PIN, SS_TX_PIN);

void setup() {
    Serial.begin(9600);
    while (!Serial);
    SPI.begin();
    softwareSerial.begin(9600);

    Stream *interfaceStream = &Serial;
    Stream *debugStream = &softwareSerial;

    canHacker = new CanHacker(interfaceStream, debugStream, SPI_CS_PIN);
    //canHacker->enableLoopback(); // uncomment this for loopback
    lineReader = new CanHackerLineReader(canHacker);

    pinMode(INT_PIN, INPUT);
}

void loop() {
    CanHacker::ERROR error;

    if (digitalRead(INT_PIN) == LOW) {
        error = canHacker->processInterrupt();
        handleError(error);
    }

    error = lineReader->process();
    handleError(error);
}

```

```

void handleError(const CanHacker::ERROR error) {

    switch (error) {
        case CanHacker::ERROR_OK:
        case CanHacker::ERROR_UNKNOWN_COMMAND:
        case CanHacker::ERROR_NOT_CONNECTED:
        case CanHacker::ERROR_MCP2515_ERRIF:
        case CanHacker::ERROR_INVALID_COMMAND:
            return;

        default:
            break;
    }

    softwareSerial.print("Failure (code ");
    softwareSerial.print((int)error);
    softwareSerial.println(")");

    digitalWrite(SPI_CS_PIN, HIGH);
    pinMode(LED_BUILTIN, OUTPUT);

    while (1) {
        int c = (int)error;
        for (int i=0; i<c; i++) {
            digitalWrite(LED_BUILTIN, HIGH);
            delay(500);
            digitalWrite(LED_BUILTIN, LOW);
            delay(500);
        }

        delay(2000);
    };
}

```