UPPSALA
UNIVERSITET

UPTEC E 22012
Examensarbete 30 hp
Juni 2022

OBD-II monitor using smartphone and cloud solutions

- For ECO driving

Didrik Olofsson

Civilingenjörsprogrammet i elektroteknik

UPPSALA
UNIVERSITET

OBD-II monitor using smartphone and cloud solutions

Didrik Olofsson

Abstract

Fuel consumption of transportation vehicles account for nearly 27% of CO2 emissions from fossil fuel. The environmental impact of of transportation can be lowered with effective ECO-friendly driving. As modern vehicles are equipped with the on-board diagnostics (OBD) system that provides vehicle's self-diagnostic data which is accessible to from the OBD-II socket in vehicles.

The aim of this thesis is to design and implement such a monitor system that consists of an OBD-II reader (containing a Bluetooth communication module) to fetch data from the OBD-II socket and then, via a smartphone, send the data to a cloud server for analysis and visualisation. To this purpose, an ELM327 Bluetooth module (an off-the-shelf OBD-II dongle) was bought, an application has been created for an Android smartphone, and a cloud server has been created using ThinkSpeak (an online solution for cloud data collection and visualisation).

As a result, such monitor system has been implemented at low cost, and then tested and evaluated together with a Toyota Prius driving a round-trip between the Swedish cities of Uppsala and Arlanda. The test results have shown that the system is successful in gathering data from the OBD-II socket and then sending visualizing data.

It needs, however, further work to increase the user friendliness and accuracy of the gathered information from the vehicle

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala/Visby

Handledare: Daniel Nygren Ämnesgranskare: Ping Wu

Examinator: Mikael Bergqvist

Populärvetenskaplig sammanfattning

OBD-II är en internationell standard i fordon som är obligatorisk i moderna bilar möjliggör att med hjälp av hårdvara läsa av information från bilen under färd. Det medför att man kan överföra data så som hastighet, bränsleförbrukning och motortemperatur från bilen till en annan tjänst där det presenteras för föraren. Metoden går ut på att transmittera informationen som läses av sensorer i bilen till en internettupkopplad mobiltelefon som för över informationen till en molntjänsst. Datan från bilen ska sedan kompletteras med positionsinformation från mobilenhetens GPS enhet. Detta medför att skapandet av en applikation som kan uppfylla målen blir är en central del av projektet.

ELM327 är en hårdvarumodul som är designad för att kopplas till ett OBD-II uttag. Modulen kan då läsa av data från CAN bussen i bilen och sedan konvertera informationen till en Bluetooth enhet. En mobiltelefon kan då kommunicera med Bluetooth enheten genom att skapa en Android applikation i utveklarverktyget Android studio. Applikationen för sedan över datan till molntjänsten ThingSpeak, som är designat för IoT project. ThingSpeak har en implementering av MatLab, en programmerings plattform för ingenjörer, som låter användaren analysera och visualisera data i molntjänsten. En begränsning som uppstår är hur mycket pengar som investeras i projetet vilket har inflytande på hur ofta datan kan uppdateras i molntjänsten, vilket påverkar nogrannheten i resultatet. I tesen diskutteras även begränsningarna av systemet och olika utvecklingsmöjligheter av projektet.

Acknowledgements

This thesis was completed at Uppsala University and Syntronic AB in Gävle, Sweden. I would like to thank my supervisor at Syntronic, Daniel Nygren, who has been both helpful and supportive during this project. I would also like to thank Syntronic AB for having me do my thesis at their office. Finally i would like to gratefully thank PhD Ping Wu for his guidance in forming the scope of this thesis.

Contents

Acronyms

AFR Air Fuel Ratio.

API Application Programming Interface.

CAN Controller Area Network.

CANH CAN High.

CANL CAN Low.

DTC Diagnostic Trouble Codes.

ECU Electronic Control Unit.

EOBD European On-Board Diagnostics.

GPS Global Positioning System.

HTTP Hypertext Transfer Protocol.

IoT Internet of Things.

ISO International Organization for Standardization.

MAC Media Access Control.

MAF Mass Air Flow.

OBD On-Board Diagnostics.

PID Parameter Identifiers.

RPM Revolutions Per Minute.

RS-232 Recommended Standard 232.

Rx Receive.

Tx Transmit.

URL Uniform Resource Locator.

UUID Universally Unique Identifier.

XML Extensible Markup Language.

List of Figures

List of Tables

1 Introduction

1.1 Background

Fossil fuel combustion causes a large amount of CO_2 emissions globally. Vehicles account for nearly 27 % of these CO_2 emissions as well as emitting a large number of other pollutants. [1] Vehicle emissions are affected by driving habits and vehicle parameters such as speed, distance travelled, and fuel consumption.

A large number of vehicles in society were developed and built before the increased development of modern digital electronics. These vehicles lack the opportunity of post-driving digital feedback to the driver to increase the driver's awareness of their driving habits.

All cars manufactured after 2001 in the EU have a socket for the OBD-II (On-board diagnostics II) or EOBD (European On-board diagnostics) standard which is a standard for collecting data from a vehicle such as gear, fuel economy, revolutions per minute (RPM) and other diagnostic information. [2] This allows for digital data to be gathered from a vehicle and sent to a computer or cloud for processing. An idea has been conceived to build a monitor that reads the given data from an OBD-II and processes the information to give the driver feedback on how safe and environmentally friendly their car journey has been, in order to use the information for the driver's improvement.

1.2 Project Purpose and Specifications

The purpose of the project is to build a car monitoring system that collects data from an OBD-II module and transfers it from a mobile phone to a cloud for analysis of the data, resulting in increased feedback for the driver. Allowing the driver to improve their driving habits becoming safer and more environmentally friendly drivers.

The method for the project is to use OBD-II data that is collected via an OBD-II dongle (with Bluetooth) to a smartphone that logs the data. The data is then sent to a cloud server such as ThingSpeak (which is an IoT analytics platform service that allows people to aggregate, visualize, and analyze live data streams in the cloud) for analysis. [3] The system allows for multiple users to send their data to the cloud.

In particular, the system should be able to:

- Collect data from an OBD-II module to an android app
- Collect data from phone GPS to an android app
- Send data from the android app to a cloud-based server for analysis
- Allow multiple users to send their driving data to the cloud

- Return information to the user that gives greater insight in the environmental impact of their driving

To obtain these goals a system will be designed that includes an OBD-II dongle with Bluetooth (The ELM327 module), a smartphone and a cloud server. A general overview of the system is presented in Figure 1.1 where the arrows represent the movement of information in the system. The three blue blocks represent the three main parts of the system while the existing tools used are presented in the light yellow blocks and the parts of the system that need to be designed and implemented are presented in green blocks.

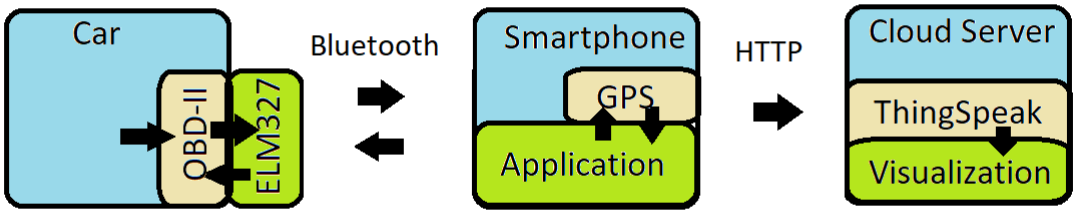

Figure 1.1: Overview of desired design of the system.

2 Theoretical Background

2.1 Vehicle network overview

Automotive vehicles are designed with multiple sensors and actuators that needs to communicate with each other. These sensors and actuators are controlled by electronic control units, ECUs, that are connected by a network called CAN. The in-vehicle network is connected to an OBD-II port that allows an outside user to get access to read data from the in-vehicle CAN network, and in extension read data from the ECUs.s. [4]

Figure 2.1 visualises a general structure of an in-vehicle network based on a CAN network. The structure of in-vehicle networks is not standardised and its design is highly dependent on the manufacturer.

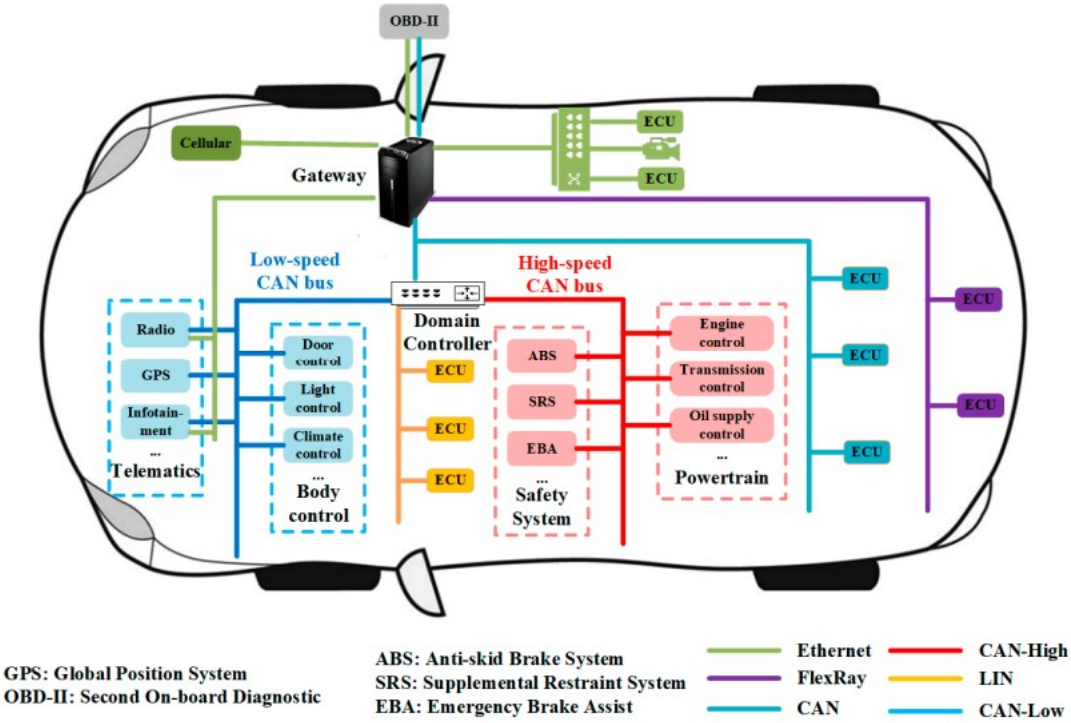

Figure 2.1: Example of an in-vehicle network for a vehicle [4].

2.2 Electronic Control Unit

Modern vehicles are more and more controlled by electronics. The electrical system includes a variety of electronic modules that control or monitor the vehicle. These electronic modules are called Electronic Control Units, or ECUs in short. Each ECU has a specified purpose of either monitoring or controlling a part of the system. Some of the functionalities of ECU are gearbox control, emission, control, tire pressure monitoring and speed monitoring. The ECUs are interconnected via communication channels (buses) that allow for data to be transferred between the units as well as to other parts of the

vehicle's electrical system. [5]

2.3 Controller Area Network

Controller Area Network (CAN) is a serial communications protocol used for internal communication in most automotive vehicles. The CAN protocol allows for secure transmission and receiving of information between Electronic Control Units. CAN was developed by Bosh in 1980 and was first used in a car in 1991 with the release of the Mercedes W140. Bosh released CAN 2.0 in 1991 which introduced CAN A (Standard CAN) and CAN B (Extended CAN), CAN 2.0 became mandatory for all automobiles in the USA in 1996 and has become common in most vehicles. [6]

CAN is an International Standard Organization (ISO) defined communications protocol that was originally developed to replace complex wiring with a two-wire bus while not being susceptible to electrical interference. The functionalities of CAN are determined by the ISO 11898 standard which sets the properties of the CAN. There are three main subsets to the ISO 11898 standard that determines some of the main properties of CAN, namely ISO 11898-1, -2 and -3. [7]

The Control Area Network consists of two networks which operate at different speeds, these are determined by the ISO 11898 -2 and -3. The ISO 11898-2 CAN bus network, commonly called high speed CAN, operates at a higher speed of up to (1Mbit/s) and monitors the powertrain of the vehicle ECUs, such as engine and transmission control, as well as the safety of the vehicle system, such as the ABS system. The ISO 11898-3 network, commonly called low-speed CAN or Fault Tolerant CAN, operates at a lower speed of up to (250Kbit/s) and controls ECUs, such as door control and climate control. [4]

2.3.1 The CAN bus

The CAN protocol uses a bus topology for communication, this allows for multiple units to send messages through a single bus. This has the advantage that it greatly lowers the amount of wiring inside the vehicle to two wires. These two wires are called CAN high and CAN low, where electronic units are connected between the CAN high and CAN low. The specifics of the CAN bus differ between the High and the Low speed CAN bus, which will be explained in the following sections. [6]

2.3.1.1 High-speed CAN bus

The high-speed CAN bus consists of two twisted wires terminated at each end by 120 Ω resistors. the high-speed bus also has a characteristic impedance of 120 Ω. The two twisted wires are called CAN high and CAN low, or CANH and CANL in short. ECUs can be connected as modules in the CAN bus by connecting a CAN receiver and transceiver to the CANH and CANL. The distance between the main CAN bus and the ECU is not

recommended to be over 30 cm, as to avoid reflections of the signals. An overview of the High speed CAN bus can be viewed in 2.2. [7]

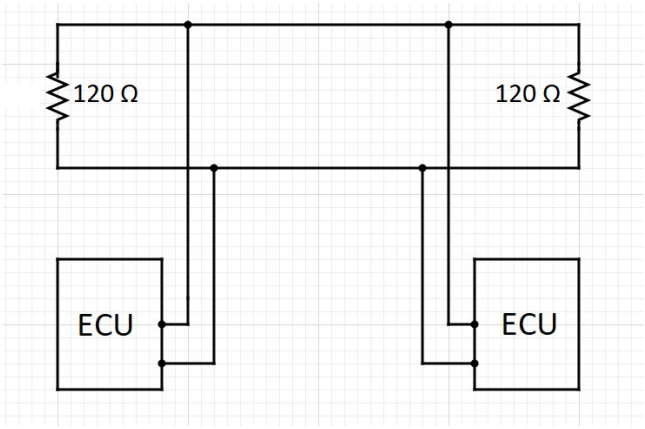

Figure 2.2: High-speed bus schematic with two connected ECUs.

,

The signal that is sent through the high-speed CAN bus shown in Figure 2.2 will look similar to Figure 2.3. When the bus transmits a logical 0 CANL will be driven low as CANH is driven high, the opposite occurs when the bus transmits a logical 1. As there is only 1V differential between when the bus transmits a dominant 0 or a recessive 1 the bus can switch between states at a fast rate, but it's more susceptible to interference and noise. This results in the high-speed CAN bus needing protection from electrical interference, resulting in a higher cost for manufacturing the high-speed bus. The faster data rate of the high CAN bus has made it crucial for time-critical data transmissions, such as communication between the engine and the throttle. The high speed CAN design cannot function in case of a short circuit or wiring failures in the CAN bus. [7]

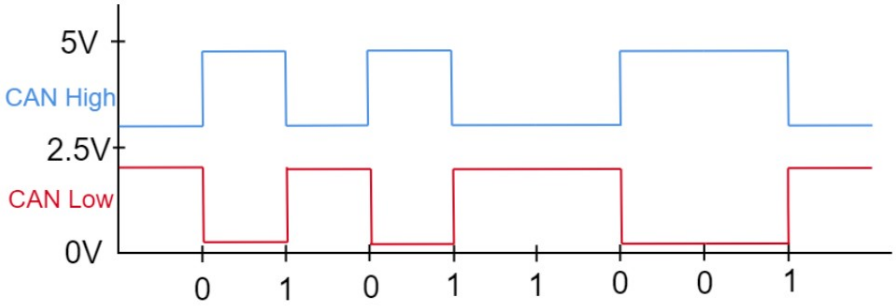

Figure 2.3: High-speed CAN signal.

2.3.1.2 Low-speed CAN bus

The low-speed CAN bus network structure differs from the high-speed CAN bus network in that instead of termination occurring by 120 Ωresistors at the end of the line, termination of the low-speed network occurs at each ECU connected to the CAN bus.

The advantage of low-speed CAN bus networking is that the CAN bus can operate even if one of the CANL or CANH wires is short-circuited och cut. The disadvantage of the low-speed CAN bus network is the decreased speed at which information can be sent over the bus. [7]

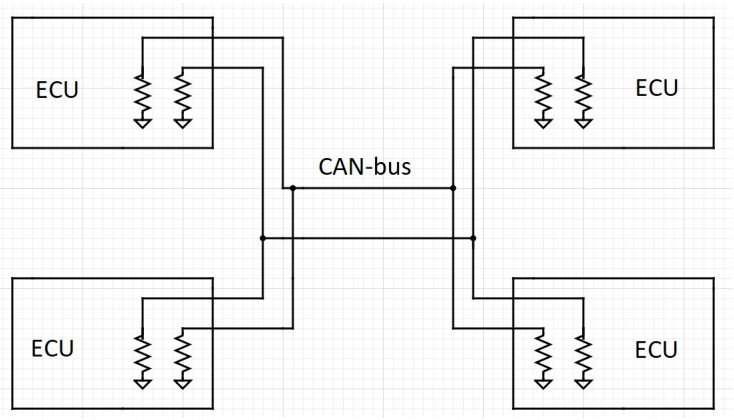

Figure 2.4: Low-speed bus schematic with four connected ECUs.

The signal that is sent over the low-speed CAN bus is presented in Figure 2.5.It can be seen in the figure that the voltage differential is larger than in the high-speed CAN, decreasing the effect of noise on the circuit. If a fault in either the CAN low or the CAN high wire occurs the CAN bus will operate in one wire transmission mode without any information loss, this has resulted in CAN low speed also being called fault-tolerant CAN. [7]

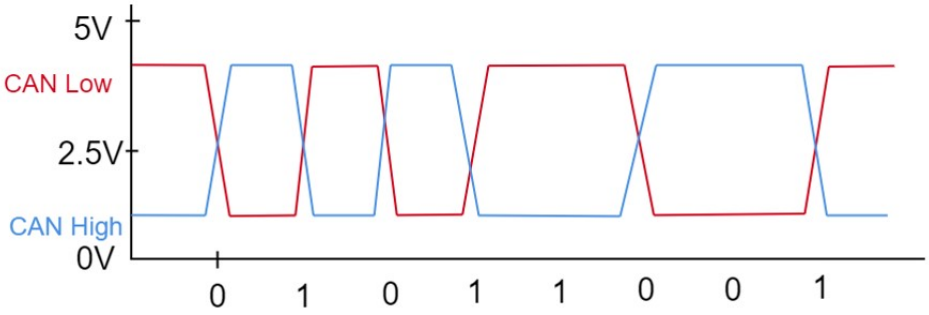

Figure 2.5: Low-speed CAN signal.

2.3.2 CAN message Frame

Messages sent over a CAN network are standardised over what information each bit of the message contains. A visual indicator of a CAN message structure is presented in Figure 2.6. The CAN message transmitted over the CAN bus can be received and decoded at the OBD-II port of the vehicle.

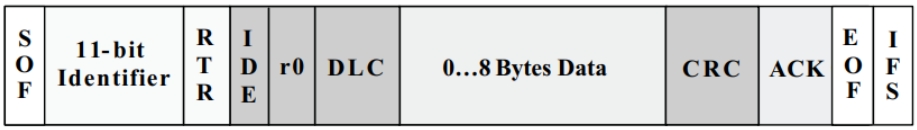

Figure 2.6: Standard CAN: 11 Bit Identifier [6].

The CAN message frame contains information needed for the communication protocol on the CAN bus, for example, a start frame and 11-bit priority. The information in the CAN message frame that is needed to extract data sent from the frame is the IDE identifier and the 8 bytes of data. A full table with an explanation of the bytes in the CAN message frame is presented in table 2.1. [6]

Field	Length(bits)	Description
Start of Frame (SOF)	1	Marks the start of a message
Identifier	11	Establishes the priority of the message on the CAN bus.
Remote Transmission Request (RTR)	1	Dominant bit when requiring information from another node on the CAN bus.
Identifier extension (IDE)	1	Identifier showing which device transmitted the data
Reserved bit (r0)	1	Reserved bit for future amendment to the message frame. standard, currently not used
Data Length Code (DLC)	4	Indicates how many bits of data being transmitted.
Data Field	0-8 bytes	Data are being transmitted
Cyclic Redundancy Check (CRC)	15	Used for error detection
Acknowledge (ACK)	2	Indicator of error free message being send
End of Frame (EOF)	7	Marker for end of CAN message frame
Interframe Space (IFS)	7	Indicates the time required to move a received frame to its position in a message buffer.

Table 2.1: Standard CAN message frame table [6].

2.4 On-Board Diagnostics

On-Board Diagnostics (OBD) is a built-in diagnostics system for vehicles that allows users to connect and receive information from the vehicle's CAN. The diagnostic information that can be collected includes system status, rpm, air flow and malfunctions. The OBD standard has developed into the modern OBD-II standard, which specifies the pinout of the OBD-II connector, signal protocol and messaging format. EOBD is the European

equivalent to the American OBD standard. A 16-pin OBD-II access port has been mandatory in automotive vehicles since 2001 for petrol cars and 2004 for diesel cars in Europe. The advent of OBD-II ports in vehicles has resulted in the development of a large variety of OBD-II scanning tools, commonly called OBD-II readers. [2]

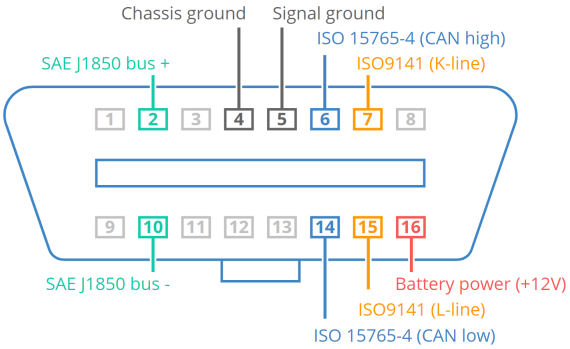

Figure 2.7: OBD-II connector [2].

There are a verity of different communication protocols supported by the OBD-II standard. The communication protocol used in a vehicle is determined by its manufacturer. The ISO 15765 standard is the most used communication protocol that communicates through the CAN bus. Other communication standards include ISO 14230-4 (KWP2000), which is less commonly used than the CAN standard.

OBD II uses two different codes to request data from ECU's. These codes are Diagnostic Trouble Codes (DTC) and Parameter Identifiers (PIDs). The Diagnostic Trouble Codes are, as the name suggests, used to diagnose failures in the subsystems of the vehicle. These failures can consist of a variety of malfunctions, such as malfunction indicator lights for engine failure, and are often displayed to the driver via the vehicle's dashboard.

The OBD-II PIDs are the identifiers for requesting data from the sensors connected to the ECUs in a vehicle. There are approximately 196 different OBD-II PIDs, depending on the vehicle manufacturer, that is accessible through the OBD-II. Table 2.2 presents the PIDs relevant to driver feedback and is used in this project. [8]

PID (hex)	PID (dec)	Data bytes returned	Description
04	4	1	Engine coolant
05	5	1	Calculated engine load
0D	13	1	Vehicle Speed
10	16	2	Mass Air Flow
11	17	1	Throttle position
1F	31	2	Run time since engine start

Table 2.2: PIDs used for gathering data from OBD-II and sent to the cloud server

2.4.1 Fuel Consumption

Fuel consumption is the measurement of the fuel consumed in litres for a vehicle for a given distance travelled in kilometres. The fuel used while driving is closely linked to the emissions of a vehicle. This makes fuel consumption a valuable tool for determining the environmental impact of a driver's habits. Fuel consumption is not directly measured by the OBD-II PID standard but a method for calculating fuel consumption is presented in [9] and is used in this project. To calculate fuel consumption the relationship between Fuel flow, the fuel burnt by the vehicle per hour, and the vehicle's speed in kilometres per hour is used. The fuel consumption is calculated in the following manner,

$$FuelConsumption = \frac{FuelFlow}{Speed} \tag{2.1}$$

Both Fuel flow and speed have defined PIDs in the OBD-II standard, fuel flow however is not available on a large number of cars. This creates the need for an alternative and more reliable of measuring fuel flow. A method for calculating fuel consumption without relying on receiving fuel flow measurements from the OBD-II PIDs is proposed in [9].

To calculate the fuel flow the mass to air ratio (MAF), given in grams of air per second, in relationship to the air to fuel ratio (AFR) and the density of the fuel ($\frac{g}{dm^3}$). MAF is reliably available from the OBD-II PIDs and AFR is the ratio of the mass of air in grams to one gram of fuel in the engine. The AFR and density of the fuel (D) are constants for the given fuel used, for gasoline the AFR value is 14.7 and the density is 820 $\frac{g}{dm^3}$.[9] The calculation of fuel flow is given,

$$FuelFlow = \frac{MAF}{AFR * D} * 3600 \tag{2.2}$$

Inserting Equation (2.2) into (2.1) leads to the calculation of fuel consumption in the following manner,

$$FuelConsumption = \frac{MAF}{AFR * D * Speed} * 3600 \tag{2.3}$$

Where the values of MAF, AFR, D and Speed can be read by the ELM327 module.

2.5 Bluetooth

Bluetooth is a short-range wireless technology for exchanging data over short distances. The wireless technology based on sending data over radio waves allows Bluetooth to replace two wire connections with a wireless option. For a Bluetooth connection between devices to be made they both need to have active Bluetooth functionalities, such as a Bluetooth device or a Bluetooth available smartphone. When a Bluetooth device is active it generates a MAC address (Media Access Control address) which can be used to identify a specific device.

Bluetooth devices have UUIDs, universally unique identifiers, that are used to identify specific devices. This allows for a device to set a UUID that specifies the device functionalities, resulting in connecting devices being able to check the functionalities of the device it is connecting to. The UUIDs can be set in Android smartphone applications while its most commonly already set by the manufacturer on external Bluetooth devices.

For two devices to connect one must scan the radio waves in the Bluetooth spectrum and select a device through its MAC address, the devices can then pair when the first device has found the device it is looking for.

2.6 Hypertext Transfer Protocol

Hypertext Transfer Protocol, typically called HTTP, is a communication protocol and the foundation of communication on the internet. HTTP is used to send and read data to and from online servers. The HTTP protocol consists of several communications methods where the GET and POST requests are the most frequently used. GET requests are used to retrieve information from a server using a URL, and POST requests are used to send data to the server. HTTP allows for easy use of sending and receiving information between hardware devices, such as a smartphone connected to the internet, and cloud servers. [10]

2.7 ThingSpeak

ThingSpeak is a online solution for cloud data collection and visualisation. It can receive data from any source connected to the internet and that is able to send HTTP requests. It is a web based cloud platform that allows users to connect and view the cloud server through a web-browser. ThingSpeak operates using different "channels" that a user can create that has a unique identifier and password so that user is the only one able to send and read data from that channel.

The free version of ThingSpeak allows the user to create three different channels that can receive up to eight different fields of data that can be updated once every 15 seconds. ThingSpeak has different payed services for increasing the amount of channels or data fields as well to increase the update frequency. The payed versions of ThingSpeak can support updating of data each second and receive up to 90 000 messages each day from external sources. The versions are divided into a "Student", "Academic", "Standard" versions that are priced differently and allow for increased data uploading and computional times.

ThingSpeak has become a popular tool for Internet of Things (IoT) projects for its powerful visualisation tools. ThingSpeaks visualisation has integrated support from Matlab. Matlab is a programming platforms designed for engineers and scientists [3]. This allows a ThingSpeak user to visualize their data using all basic functionalities of Matlab. [3]

3 Implementation

3.1 Overview of the system

The overall structure of the system is presented in Figure 1.1. The system to be designed and implemented in this degree project consists of three parts (presented in the green block. The first part of the system is the actual hardware connected to the OBD-II port in a car. This part of the system has the purpose of reading the information transmitted from the ECUs over the CAN bus to the OBD-II port and converting the data to the same communication standard that the bluetooth module operates with. The bluetooth module then has the function of communicating with the other parts of the system.

The second part of the system is the creation of an application, simply app in short, on a smartphone that can receive the information from the hardware part of the system as well as access the GPS data on the smartphone. The application also has the function of transmitting the information that it gathers to the last part of the system, a cloud based server.

The cloud based server is the third part of the system, Its function is to receive information from the app running on the smartphone and then analysing and presenting the information gathered in a way that is useful for a driver to improve their driving habits.

3.2 Hardware

3.2.1 The ELM327 module [11]

The ELM327 bluetooth module used in the project can be connected to the OBD-II port in a car and read from the OBDs communication protocolls. It is a low priced module containing a ELM327 chip, which given the name of the module, and a HC-06 Bluetooth module that allows for Bluetooth communications with the chip. The module costs approximately 70 Swedish kronor, making it a low cost option that lowers the monetary investment needed in to replicate the project. Figure 3.1 presents the module used in the project, which has a male connector to a OBD-II socket and ELM327 functionalities.

Figure 3.1: ELM327 module with Bluetooth functionalities

The ELM327 chip and the Bluetooth device used will be described in detail in the following sections.

3.2.2 ELM327 chip

To gather data from the ECUs in a vehicle, through a CAN bus accessed via an OBD II port, hardware needs to be connected that can communicate with the desired protocols. The ELM 327 is an electronic chip that is designed to act as a bridge between the OBD ports and a RS232 serial interface. [11]

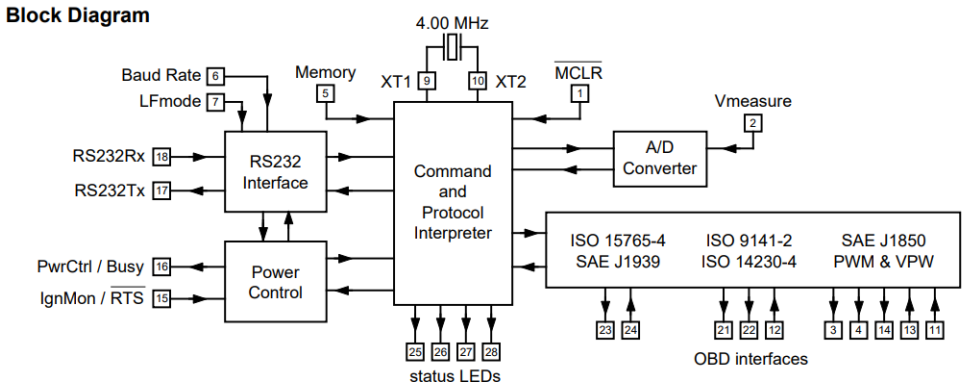

Figure 3.2: ELM327 block diagram [11].

It can be seen in Figure 3.2 that the ELM327 chip offers support for the most common OBD II signal protocols, most notably a support for ISO 15765-4 (CAN) messages via at port 23 (CAN Tx) and 24 (CAN Rx). The ISO 15765-4 (CAN Tx and CAN Rx) can be directly connected to their corresponding pins on the OBD II socket presented in Figure 2.7. The ELM327 can interpret these signal protocols and convert it to a more common

RS232 interface on pin 18 (Tx) and pin 18 (Rx) for communication with other external devices, for example a Bluetooth module.[11]

The ELM327 is designed to be connected to a OBD II socket in an car, and receive power directly from the cars 12 V battery, it will always be turned on as long as power is supplied. To decrease the power consumption of the module a ignition monitor is connected to pin 15, this monitor transmits a logical low0 when the ignition is turned off, signaling to the ELM327 to go into low power mode. If the ignition is turned on the ignition monitor will transmit a logical high 1, turning the ELM327 module to resume its full power functionalities. [11]

3.2.2.1 OBD messages

To communicate with the OBD-II device through the ELM327 the datasheet for the ELM module presents a large amount of commands that can change the setting on the ELM327 or communicate with the OBD-II socket. The most basic command that is needed in the implementation of the device is a OBD-II PID request. The documentation for the ELM327 states that a PID request is sent when the ELM327 device receives a 01 followed by a given PID number. Given this information messages can be created for requesting information through the OBD-II socket. The table 2.2 presents a selection of useful PID values that can be used, an example of a OBD message is presented in Figure 3.3. One thing to note is that the ELM327 operates using hexadecimal values, hence the conversion from the decimal numbering of 11 to the hexadecimal numbering of 0B. [11]

```
Message sent            >01 0B
Message received        <41 0B 00
```

Figure 3.3: An example of a OBD request and response message sent for measuring speed.

The response received in Figure 3.3 can be decoded using the datasheet of the ELM327. It states that all 01 requests gets a response of 41 followed by the requested PID and then the data from that PID. In the example of Figure 3.3 the request sent is for the vehicle speed and the last byte in the response 00 states that the vehicle has a speed of 0, as its standing still.

3.2.2.2 Multiple PID messages

While it is possible to send multiple different single OBD messages as presented in Figure 3.3 it requires sending a large amount of messages over the CAN bus. The ELM327 has a functionality for sending up to six different OBD-II PID requests in a single OBD message. When a multiple OBD message is sent all ECUs that has the requested information will respond. It is therefore required to activate headers for the response message, this will include a numeric indicator for which ECU is responding the the received message.

Headers can be activating by sending a "AT H1" message to the ELM327, activating headers in the ELM327s setting. [11]

To send a multiple OBD message requesting the PIDs previously presented in Table 2.2 a message can be constructed as shown in Figure 3.4 (a). The response shows that three different ECUs has relevant information (ECU 7E8, 7EA and 7EB) and that the requested information takes seven responses. To simplify the received information the response can be filtered so that only the responses from ECU 7E8 is read, the choice of which ECU to read will be shown later on. The message that is then received will have the format shown in Figure 3.4 (b).

```
Message sent          >01 05 0D 10 11 1F 04
Message recived       <7E8 10 0F 41 05 42 0D 05 10
                       7E8 21 00 5C 11 1F 1F 00 00
                       7EA 10 0C 41 05 42 0D 00 11
                       7E8 22 04 99 00 00 00 00 00
                       7EA 21 30 1F 00 00 04 B0 00
                       7EB 10 0C 41 05 42 0D 00 11
                       7EB 21 30 1F 00 43 04 B0 00
```

(a) Full response

```
Message sent          >01 05 0D 10 11 1F 04
Message recived       <7E8 10 0F 41 05 42 0D 05 10
                       7E8 21 00 5C 11 1F 1F 00 00
                       7E8 22 04 99 00 00 00 00 00
```

(b) Filtered response

Figure 3.4: Full and filtered response from a multiple PID message.

The received message string now contains an identifier (header) followed by 8 bytes of data. This can be compared to the previously discussed CAN message frame, where it can be seen that the message received on the ELM327 follows the CAN message frame only with certain bytes in the frame not being printed. To print the full CAN frame AT commands can be sent to the ELM327 in the same way as the "AT H1" command.

To decode and understand the received OBD message response from the ELM327 the datasheet describes what each byte in the response means. The first byte in the message indicates which ECU that has responded to the message while the second byte indicates which order these responses should be read, 10 being the first response and the following responses should be ordered $2X$ where X increases for each response. The third byte in the first response indicates how many bytes of data that will be received, in this case hexadecimal 0F is 15 bytes of data that will be followed with zeroes. The fourth byte shows that the response is to a 01 request and all following bytes is the data received. The received data is colour coded in Figure 3.5 [8] and can be read using the OBD-II PID table presented in table 2.2.

```
Message sent          >01 05 0D 10 11 1F 04
Message recived       <7E8 10 0F 41 05 42 0D 05 10
                       7E8 21 00 5C 11 1F 1F 00 00
                       7E8 22 04 99 00 00 00 00 00
```

Figure 3.5: Decyphering a multiple PID OBD message.

It can be seen in Figure 3.5 that the ECU numbered 7E8 can respond with all requested

information in the OBD message. It is therefore not needed to decode the information from the other ECUs that responded in Figure 3.4 (a).

3.2.3 Bluetooth module

To communicate wirelessly to the ELM327 chip Bluetooth is used. The Bluetooth functionalities of the ELM327 module is handled by an HC-06 Bluetooth module that is connected to the Rx and TX pins on the ELM327 chip. The HC-06 module requires 3.3V to operate which is obtained from a voltage conversion from the ELM327 chip and is connected to ground through the OBD-II port. The HC-06 is low cost and operates with low power consumption, making it ideal for a low cost/power module such as the chosen ELM327. Figure 3.6 presents the HC-06 module used in the ELM327 module, where the antenna is located on the left side of the figure.[12]

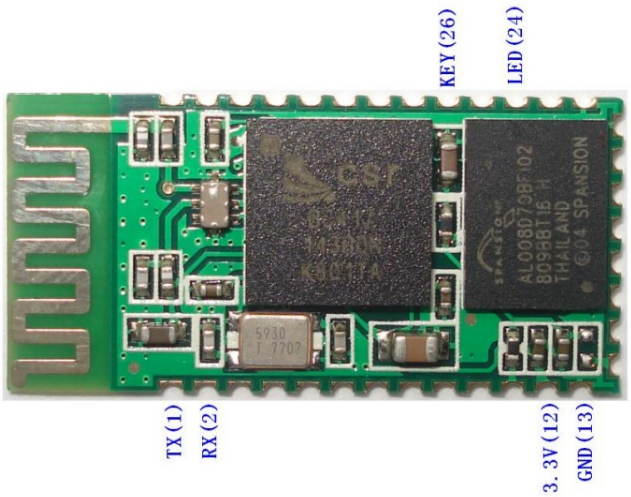

Figure 3.6: HC-06 Bluetooth module

3.3 Software implementation

3.3.1 Android Studio

Android studio is a development environment created and supported by Google for development of Android applications. Android studio allows for development of applications in either Kotlin or Java programming languages, Java was used in this project. The software written to achieve the goals of this project can be divided into two groups. The backend programming, which is the underlining functionalities of the application as well as the communication with hardware and outside databases. The frontend programming, which is the functions available and visible in the application to the user, is written in XML. [13].

3.3.2 Android Backend

3.3.2.1 Bluetooth terminal

To enable some basic Bluetooth functionality such as enabling Bluetooth, scanning for devices, and connecting to a device the Android studio development documentation references sample code for a Bluetooth Chat application [14]. The Bluetooth Chat application demonstrates the usage of all the fundamentals of the android Bluetooth API, making it a useful foundation for developing more advanced Bluetooth applications.

There are two major changes that needs to be made in order to apply the Bluetooth Chat sample to the desired OBD-II monitor. The first change that needs to be made is to set a Universally Unique Identifier (UUID) for the bluetooth communication. The UUID is a 128-bit long identifier which needs to be the same on both the receiving and transmitting bluetooth device. [15] The OBD-II bluetooth device has a set UUID that has to match with the one set in the application. The setting of the UUID is presented in Figure 3.7.

```
private final UUID MY_UUID = UUID.fromString("0001101-0000-1000-8000-00805F9B34FB");
```

Figure 3.7: UUID code.

For the Bluetooth terminal to be able to continuously receive messages the application needs to continuously read the bytes it receives over Bluetooth. This can be seen in the while(true) loop in Figure 3.8. To prevent the application from getting stuck in the while loop, it is set in a separate Thread. Threads can run simultaneously as the main activity of the application and will in this case send the received message to the main activity when a new message is received.

```
while (true){
    try {
        bytes = inputStream.read();
        if (bytes == 0x0A){
        }else if (bytes == 0x0D){
            buffer = new byte[arr_byte.size()];
            for (int i = 0 ; i < arr_byte.size() ; i++){
                buffer[i] = arr_byte.get(i).byteValue();
            }
            handler.obtainMessage(MainActivity.MESSAGE_READ, buffer.length, arg2: -1, buffer).sendToTarget();
            arr_byte = new ArrayList<Integer>();
        }else {
            arr_byte.add(bytes);
        }
```

Figure 3.8: Bluetooth response code.

The inputStream in Figure 3.8 is the received bytes from the Bluetooth device. The app checks if the inputStream send a newline (0x0A) or a return (0x0D) character, where a return character represents the end of a received message. The Bluetooth reading

function sends the received message to the main activity using a handler function and then clears the buffer.

3.3.2.2 Reading OBD messages

After a message is received and sent to the main activity it needs to be decoded as in Figure 3.5 in order to store the PID data variables. This is done in Figure 3.9 where it is checked if the received message matches the correct hexadecimal message structure, this is done using regular expressions. [16] The received message string is then split into each byte and stored into an array. The array is then stepped thorough where each byte is checked for a desired PID value. In Figure 3.9 the last row shows when a decimal 16 is found, which according to table 2.2 represents the Mass air flow value.

```
private void setDataReceivedData(String dataReceived){
    if (dataReceived.matches( regex: ".*[0-9A-Fa-f]{2}.*")){
        dataReceived = dataReceived.trim();
        Log.v( tag: "message recevied", dataReceived);
        String data[] = dataReceived.split( regex: " ");

        Integer i = 1;
        if (data[0].matches( regex: "7E8")){
            while (i<data.length){
                Integer PID = Integer.parseInt(data[i].trim(),  radix: 16);
                Log.v( tag: "PID ", PID.toString());
                switch (PID){
                    case 16:
```

Figure 3.9: Parsing input response code.

3.3.2.3 Global Positioning Data

The Global Positioning Data (GPS) functionality on an Android smartphone allows for reading of the longitude and latitude of the devices location. The Android studio development tool includes a location API that simplifies the communication with the GPS. [17] The smartphone user needs to give permission for the application to use the location API, the application therefore asks for permission at launch. The longitude and latitude of the device can then be read and logged as in Figure 3.10.

```
private void updateLocationData(Location location) {
    this.latitude = String.valueOf(location.getLatitude());
    this.longitude = String.valueOf(location.getLongitude());
```

Figure 3.10: Location update code.

The distance between two locations in longitude and latitude can be converted into

meters. To obtain the distance of a driven track the distance between each location update can be added into a cumulative distance.

3.3.2.4 HTTP

HTTP GET requests can be formatted in different ways depending on what information is being sent and what server its being sent to. Figure 3.11 presents an example of a GET request. The server that the information is transmitted to is marked in blue, in this example its the URL to the service ThingSpeak. A security key called api key is included in green and is specified by the service of the desired website. For this URL the information that is to be transmitted is presented in red and in this case sets that the field1 is to be set to zero. [3]

```
GET https://api.thingspeak.com/update?api_key&field1=0
```

Figure 3.11: Example of HTTP GET request.

The Volley library for Android studio makes it possible to construct HTTP GET request strings as presented in Figure 3.11. The library manages networking requests over HTTP allowing an application in Android Studio to send networking data over HTTP through a string. Figure 3.11 presents the string constructed by the application in Android Studio sending data to the cloud server. [18]

3.3.3 Andorid Frontend

3.3.3.1 Terminal

The Android frontend is written in XLM code, which stands for extensible markup language, that describes how the user interface will be setup. Figure 3.12 describes the XML code for a part of the interface which the user can interact with. It creates a ListView, which is a list of inputs, that fits the full screen of the smartphone with a margin on the bottom of the screen.

```
<ListView
    android:id="@+id/list_terminal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginBottom="135dp"
    >
</ListView>
```

Figure 3.12: XML code for displaying Bluetooth messages.

The full XML code for the main activity of the application will result in Figure 3.13, where the code from Figure 3.12 creates the rows of text in the middle of the screen. In the Example of Figure 3.13 a message has been sent following the structure of 3.5 and the deciphered data is printed in the terminal. The top right of the application is reserved for three different settings and the bottom row of the application is reserved for a text editor where a message can be sent. The button "START TRIP" will start a trip where the data is monitored.

Figure 3.13: Main activity for Bluetooth terminal.

3.3.3.2 Settings

The top right of the application shown in Figure 3.13 includes three different options of settings. These settings are critical for the application to achieve the desired goals of the application. For the application to be able to connect to a Bluetooth device, such as the ELM327, the smartphone needs to activate Bluetooth and be able to find surrounding Bluetooth devices. [14]

The left figure in Figure 3.14 presents the devices that are Bluetooth paired to the smartphone. The user can choose which device to connect to the application, for example the OBD-II device. The user can also scan for surrounding devices to connect to by pressing the scan button in the top right.

The middle figure in Figure 3.14 asks the user if they want to allow the application to get access to the smartphones Bluetooth functionalities. If the user declines the give the application access no devices can be connected and the user will be informed that Bluetooth functionalities are necessary for the application to work as intended.

The right figure in Figure 3.14 shows the setting for choosing ThingSpeak channel. This allows the user to choose where they want to upload their data. This allows for multiple users to use the application and send their collected data to different ThingSpeak channels.

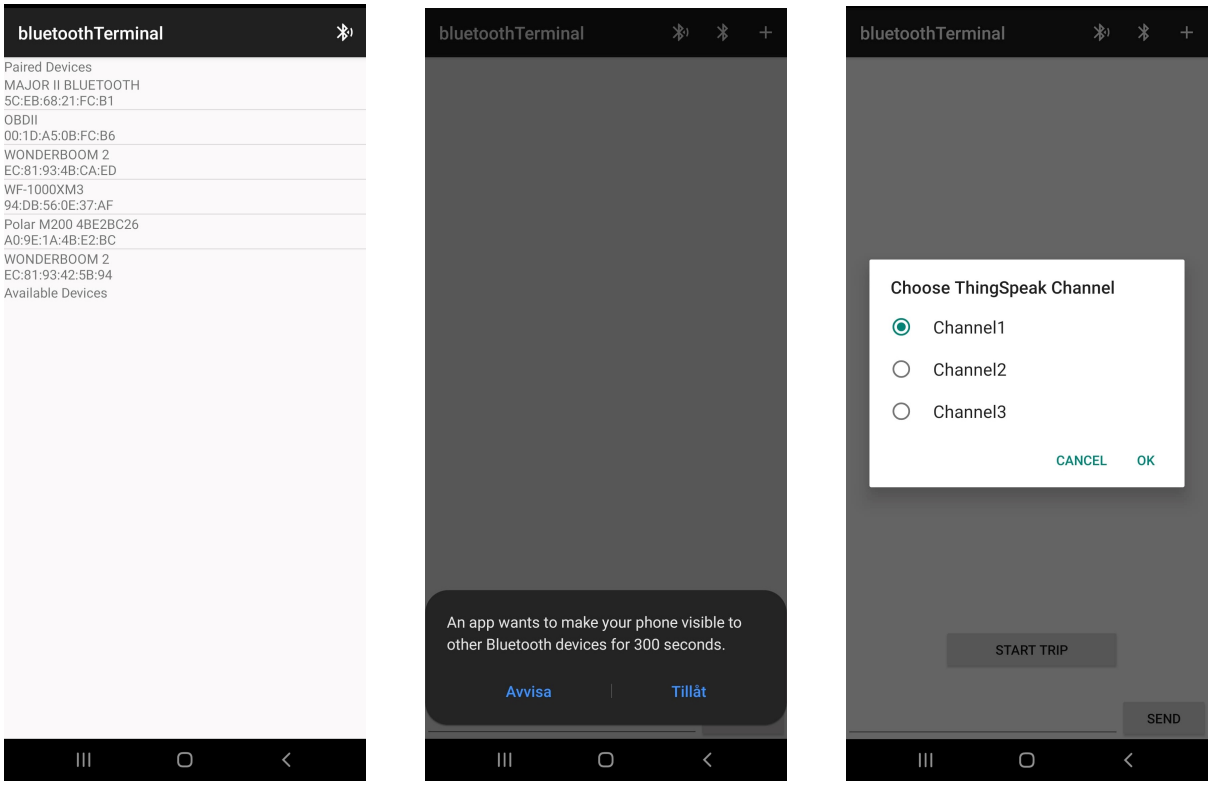

Figure 3.14: Settings available in the application.

3.3.3.3 Gauges

When the user starts a trip by pressing the start button in Figure 3.13 the program for when running will start. This will open the page shown in Figure 3.15. While running the application will send the multiple OBD Bluetooth message presented in Figure 3.5 once a second. This update speed was chosen as its fast enough for the user to get accurate data while giving the application more than enough time to decipher the response. When a value is updated it will send the new value to four gauges as well as one numeric indicator. It will update the gauges with the current speed, engine coolant temperature and engine load received from the ELM327 in addition to the fuel consumption calculated with Equation (2.3). The numeric indicator shown in the bottom of Figure 3.15 displays the current distance travelled since start.

While a trip is running the application will send HTTP GET requests to the ThingSpeak Channel selected in the settings once every 15 seconds. The update frequency for the HTTP GET request is set to 15 seconds due to limitations in the update frequency allowed in the free version of ThingSpeak.

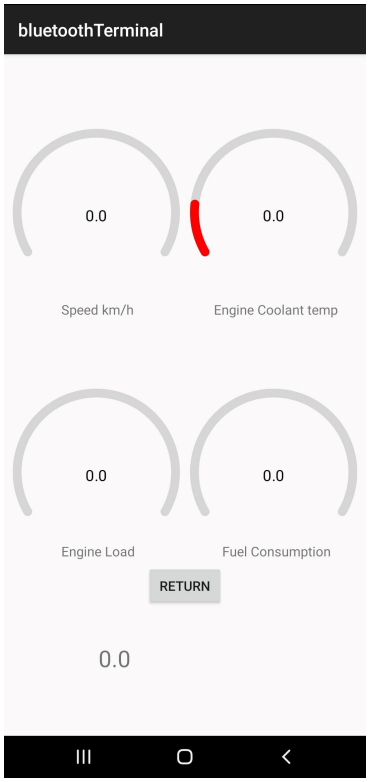

Figure 3.15: User interface while driving.

When the driving trip is finished the user can press return which will stop updating the data, send the last collected data, return the gauges to default position and return the user to the starting page shown in Figure 3.13.

3.4 Cloud solution

3.4.1 ThingSpeak

A channel on ThingSpeak can receive up to eight different fields of data, the most common way to receive data is through HTTP GET requests. The smartphone sends the following data shown in Figure 3.1 to a ThingSpeak channel chosen by the user as in Figure 3.14. The data collected by ThingSpeak is then presented to the user in seven different figures on the ThingSpeak website. [3] The resulting figures are presented in the Results.

ThingSpeak	Collected data
Field 1	Engine coolant temperature
Field 2	Throttle position
Field 3	Vehicle speed
Field 4	Mass air flow
Field 5	Engine Load
Field 6	Distance traveled
Field 7	Longitude
Field 8	Latitude

Table 3.1: Data logged on the ThingSpeak server.

The data that has been collected on the ThingSpeak channel is then visualized using the integrated Matlab support. Each figure created in ThingSpeak using Matlab needs its own Matlab code. Figure 3.16 shows the code for reading data from the ThingSpeak field number eight to the Matlab code. The readAPIKey is specific for the ThingSpeak channel and the oneDay-days(t) specifies the desired time period of data. For this specific case the time period is of the current day. The code presented in Figure 3.16 can then be repeated for each desired field of data needed in the specific ThingSpeak figure.

```
% Read location Data.
lat = thingSpeakRead(readChannelID,'Fields',myFieldID1,...
                                    'ReadKey',readAPIKey,'dateRange',oneDay-days(t));
```

Figure 3.16: Matlab code for reading of data on ThingSpeak.

When the ThingSpeak fields are read as in Figure 3.16 it still needs to be processed and displayed. The processing of the data is specific for each collected data. Where some data not needing processing and other data such as fuel consumption needs to be calculated. Figure 3.17 presents the code used for visualisation of fuel consumption on a geographical map using the Matlab function geoscatter and geobasemap. [19] Geoscatter creates geometric circles at given locations which colour is determined by a variable, such as fuel consumption. The colour scale of the geometric circles can also be changed using a colour map, which in Figure 3.17 is set to "turbo", a colourmap given by Matlab. The data can then be displayed on maps with different geographical data, in this case the topographical map gave useful geographical information while creating a clean look.

```
% Visualize the data
A = ones(1,length(lat));
geoscatter(lat,lon,A*10,smoothFuel,'filled')
title('Fuel consumption at GPS positions')
colormap(turbo( length(fuelCons)));
c = colorbar;
c.Label.String = 'Fuel consumption L/100km' ;
geobasemap('topographic');
```

Figure 3.17: Matlab code for visualization of fuel consumption data.

The above examples were then applied to the different data fields in ThingSpeak and visualized in figures presented to the user and in the Results section below.

4 Results and Discussion

4.1 The OBD-II monitor system

When the system presented in Figure 1.1 has been realised according to the implementation the real system setup is presented in Figure 4.1. The three main system parts are marked in the figure, namely the ELM327 Bluetooth module, Android application and the ThingSpeak web server. The ELM327 module is connected to the OBD-II socket and the Android application is updating its gauges as shown in Figure 3.15 according to the data received from the ELM327 module. The application then sends the data to the ThingSpeak server that visualizes the data on a tablet or computer. The computer running the ThingSpeak web page does not need to be located in or near the vehicle and should not be used while driving. The ThingSpeak web page presents a more detailed summary of the data collected during the trip which will be presented and discussed in the following section.

Figure 4.1: System setup in vehicle

The ELM327 Bluetooth module is connected to the OBD-II socket located under the steering wheel in a Toyota Prius, the location of the OBD-II socket differs for vehicle manufacturers. The location of the ELM327 Bluetooth module which is connected to an OBD-II port is shown in Figure 4.1 and a closeup of the connected module is shown in Figure 4.2

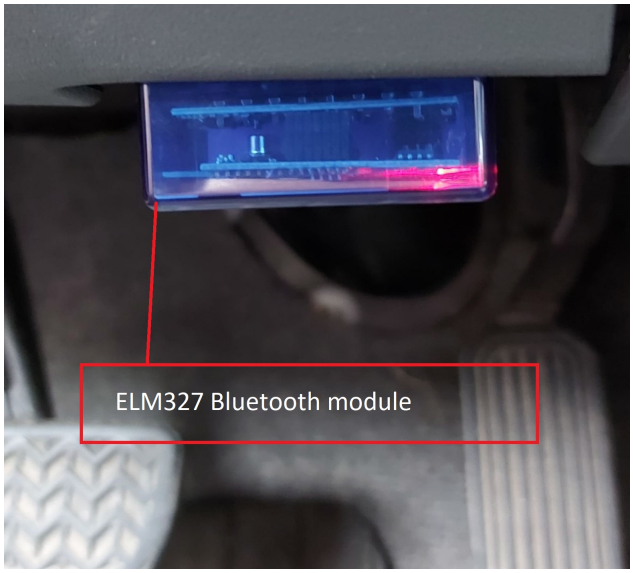

Figure 4.2: ELM327 module connected to the vehicles OBD-II port

4.2 Data Visualization and Analysis in the Cloud Server

The following results were created during a round-trip between the Swedish cities of Uppsala and Arlanda with a Toyota Prius. The trip duration was around one hour and resulted in 240 data points. The following figures are presented to the user in the web-based cloud server ThingSpeak.

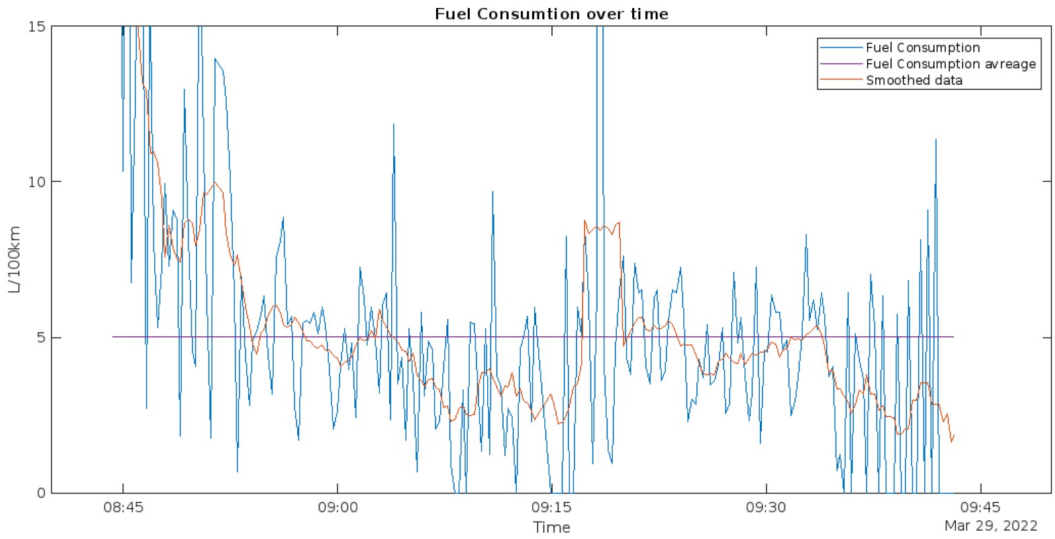

Figure 4.3: Fuel Consumption over time.

Figure 4.3 presets the fuel consumption calculated using Equation (2.3). The raw data is presented in blue and shows a large amount of spikes, a moving average was used to give the user a more visible view of their fuel consumption. The average fuel consumption is also plotted in purple and was around five for this trip.

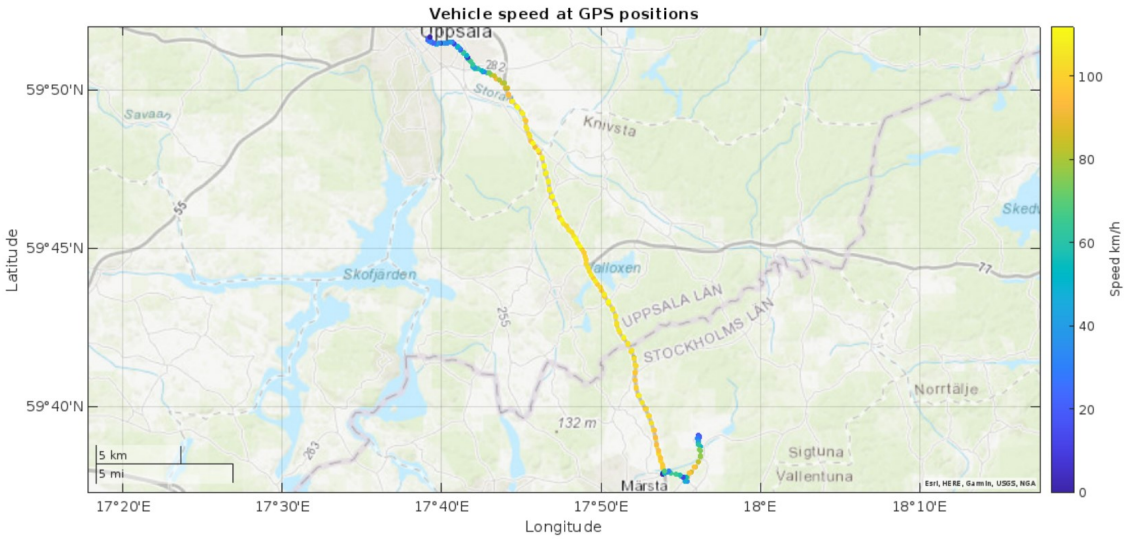

Figure 4.4: Vehicle speed at geographical location.

The vehicles speed is presented in Figure 4.4 in relation to the location. The main point that can be viewed in the figure is the difference in speed between the city and the highway connecting the cities.

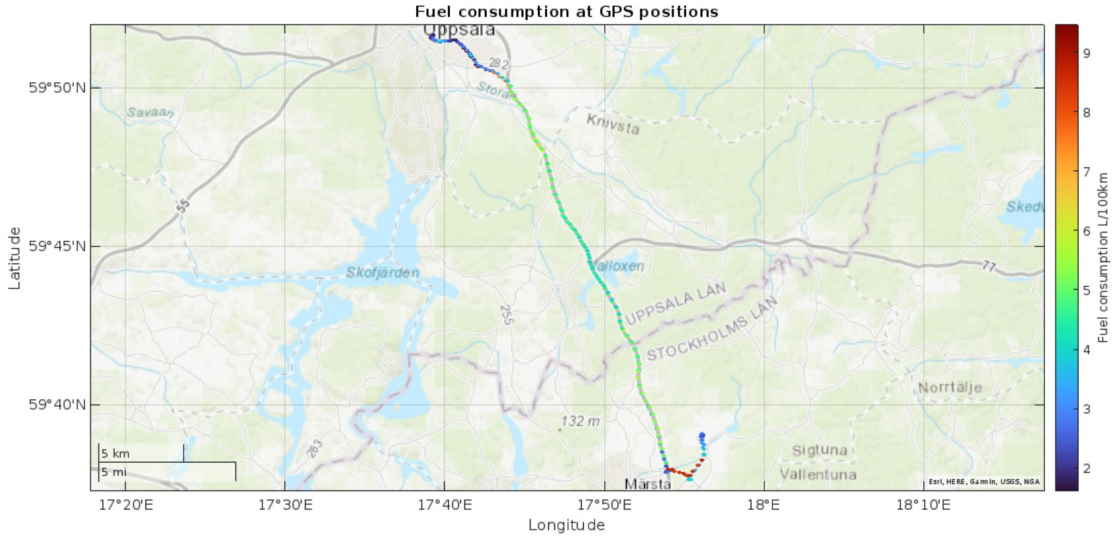

Figure 4.5: Fuel consumption at geographical position.

The fuel consumption is presented in Figure 4.5 and relates the fuel consumed at certain geographical locations. A difference in fuel consumed can be seen between driving in the city and driving on the highway. A limitation with the ThingSpeak server is the resolution of the plot, as two data points are plotted over each other when close.

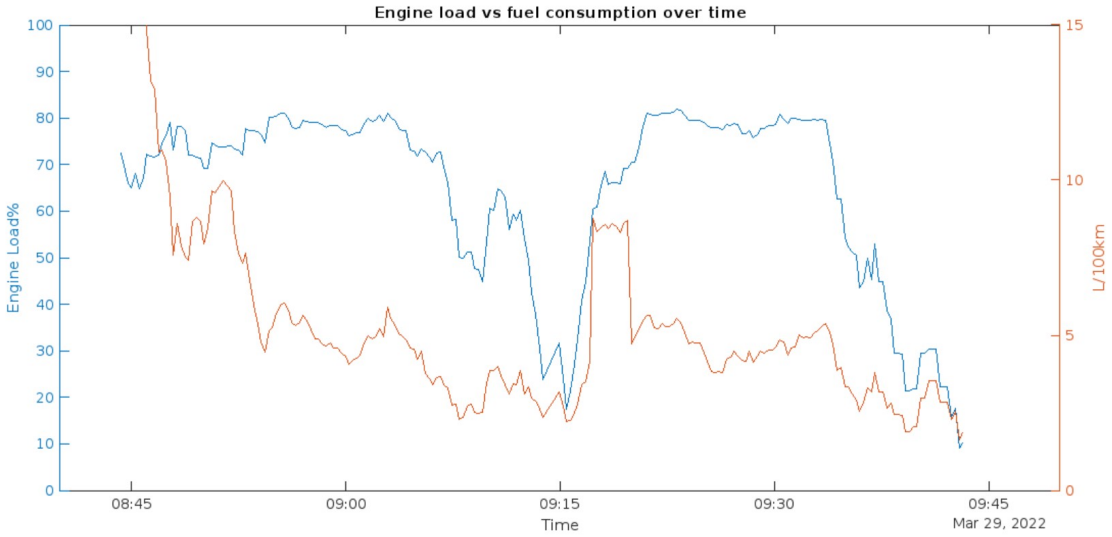

Figure 4.6: Engine load vs fuel consumption over time.

Figure 4.6 shows the relationship between the engine load and the fuel consumed. The fuel consumption is high in the beginning as its measured in L/km and the vehicle is close to or stationary. Another explanation to the high fuel consumption at the beginning of the trip can be seen in Figure 4.8. It can be seen at the $09:15$ time mark, which is at the turning point of the trip and the vehicle going to stationary and then starts moving again, that a large increase in engine load also corresponds to a large increase in fuel consumption.

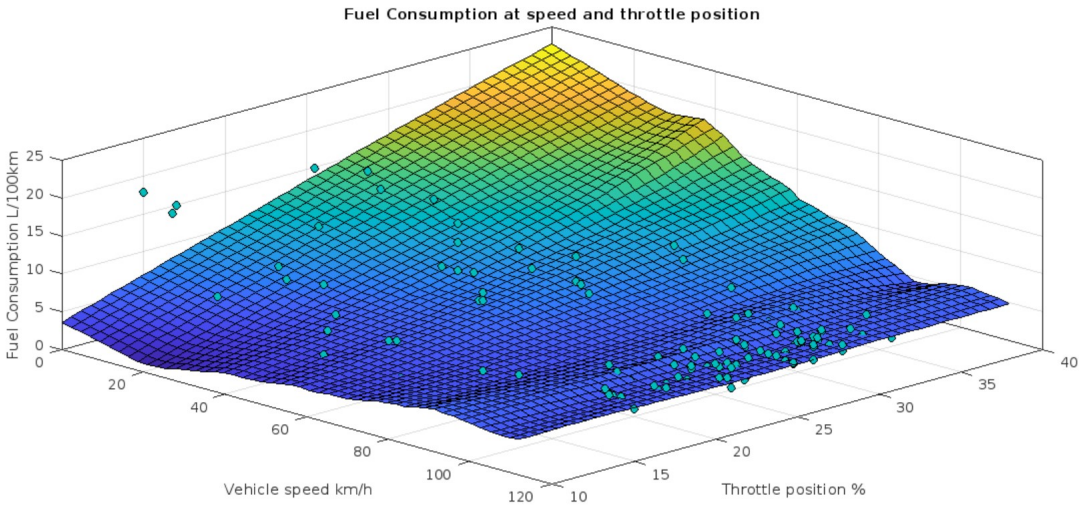

Figure 4.7: Fuel consumption at speed and throttle position.

Figure 4.7 presents fuel consumption in relation to vehicle speed and the throttles position. It shows that the larges fuel consumption occurs when the vehicle is stationary and at maximal throttle position. The lowest fuel consumption occurs at $20km/h$ with the lowest throttle position. According to fuel consumption theory for vehicles the lowest fuel consumption should occur between $60 - 80km/h$ [20], a substantial difference from

the figure. $20km/h$ occurs at the location of which some data points are substantially higher than the plane fitted over all other data points. This indicates a fault in the plane that is unable to accurately fit the higher data values at low speeds and a low throttle position. There are multiple possible explanations to the data points not fitting, one of these being that the vehicle used being a hybrid car, causing an electric motor to be active. Another fact that the data points that are not fitted to the graph occurs closely after the start of the vehicle will be discussed with Figure 4.8.

There is a small increase of fuel consumption in Figure 4.7 at speeds of approximately $80 - 100km/h$ that then decreases after $100km/h$. According to fuel consumption theory the lowest fuel consumption should occur between $60 - 80km/h$ and then increase at higher speeds [20]. If we look at the speed of $80 - 100km/h$ in Figure 4.4 we can see that these speeds occurs at certain location, most commonly on the on/off ramps of the highway. This increase of fuel consumption could therefore be related to the change of the vehicles speed at these locations as well as the fact that most on/off ramps of the highway having an increase/decrease of altitude.

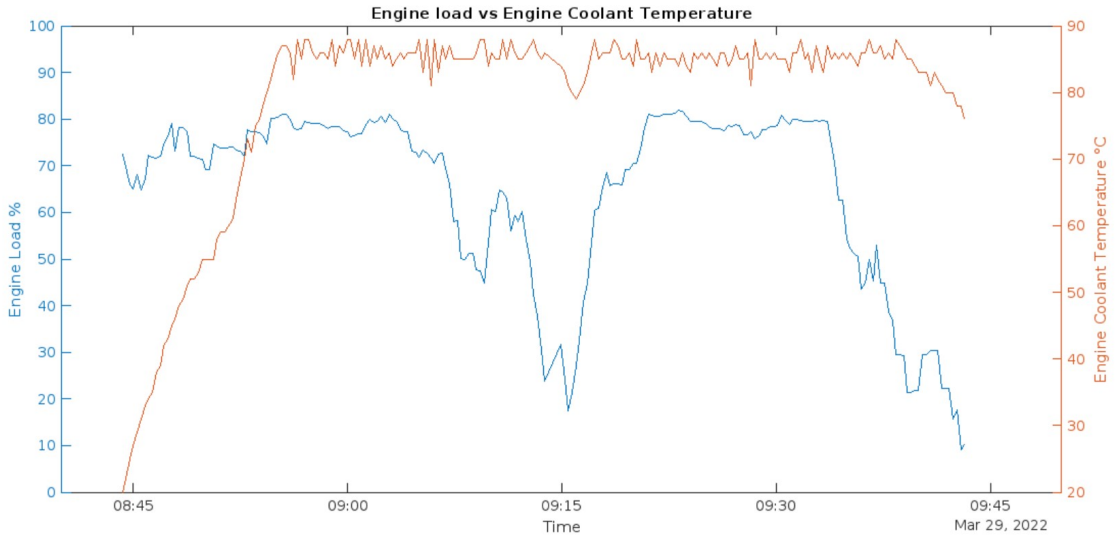

Figure 4.8: Engine load and engine coolant temperature.

Figure 4.8 presents the relationship between the engine load and the engines coolant temperature. The engine coolant temperature is an indicator of the engines temperature. At the start of the trip the temperature greatly increases until it reaches $80 - 90C$ where it somewhat stabilizes. The fuel efficiency in the engine increases at higher temperature. This explains why the largest fuel consummation measurements in Figure 4.7 occurred at the start of the trip, while the engine temperature was lower. The engine load then decreases in the middle of the trip and at the end of the trip, both these correspond to the stop in the middle of the trip and at the end. The decrease in engine load resulted in a decrease in the engine temperature, suggesting that a prolonged stop results in a decrease in engine temperature and an increase in fuel consumption.

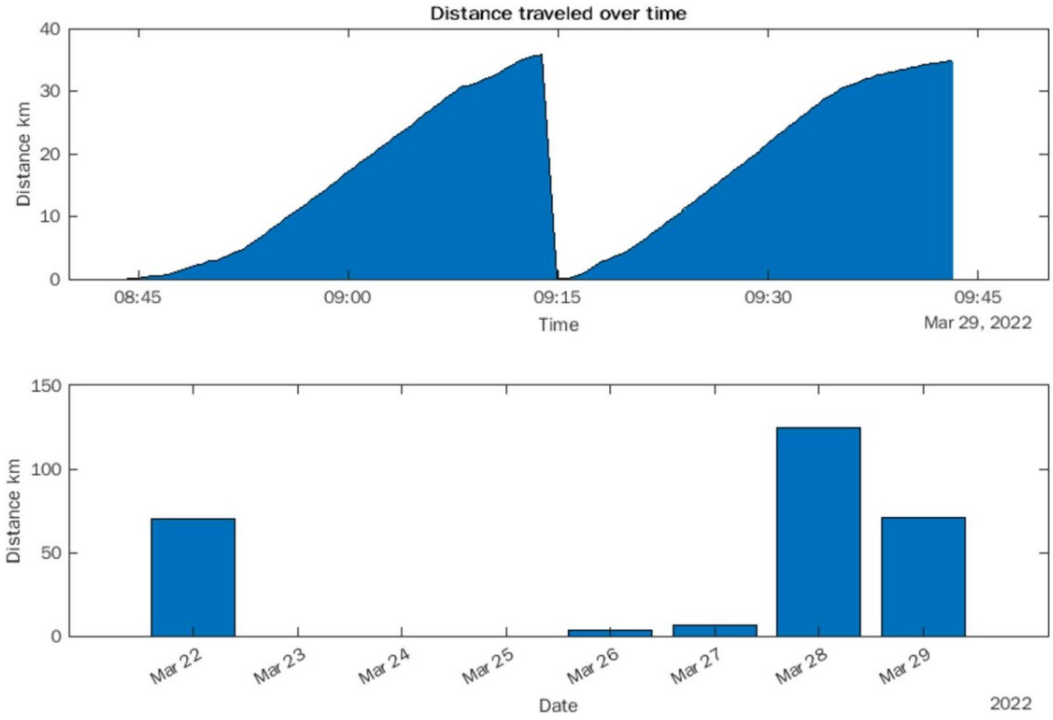

Figure 4.9: Plot of distance traveled over time.

The distance travelled is the greatest indicator in the environmental impact of a journey. The distance traveled is presented in two ways in Figure 4.9. The above part of the figure shows the distance travelled over time during the day, where the distance is set to zero at the start of a new journey. The lower part of the figure shows the distance travelled the last eight days. The purpose of this is to give the user insight in how long they drive, and in extent an indicator in their environmental impact.

5 Conclusions

The OBD-II socket can be used to connect external hardware to a vehicles communication network in order to read data from that vehicle. The collected data has then been transferred to a smartphone via a Bluetooth connection. The smartphone can then transmit and receive information from the vehicle and upload the data to a cloud network, where the data can be visualised for the driver. The data that is presented and visualised for the user consists of information that directly relates to the environmental impact of the drivers trip. With this having been achieved the project goals set at the start of the project has been reached.

The project goals has been achieved by creating a low cost system that relies on common hardware such as a smartphone with low cost hardware such as the ELM327 module as well as an open source cloud solution with a free option. The total cost of the project rounded up to 70 Swedish kronor.

Throughout the project limitations with the used cloud solutions were found, namely the low update frequency allowed for data. This has resulted in a decrease in accuracy in the results as a large amount of data points collected in the smartphone application are unable to be uploaded to the cloud server. This could have been addressed in multiple ways, one of which being investing money in a higher performance ThingSpeak account. This option was not chosen as a desire to keep the system cost effective was deemed to be valuable to the project. To make the system more accessible to a common user. Therefore a option to smooth the gathered data to make it more readable was used. This was done by averaging the collected data during a set time period and then sending the average value for that time period to the server. This decreased the accuracy of data at specific locations but could still return useful feedback to the driver.

In conclusion the method of using the OBD-II socket to monitor a vehicle using smartphone and cloud solutions was deemed to be a successful method. Improvement areas were found when it comes to the chosen cloud solution platform but the system is a reliable basis in which to continue to increase a drivers insight in their driving habits.

6 Further work

There are a multitude of options to improve and expand the scope of the designed system. One of these is to focus on the user friendliness of the system. The current system requires that the user connects to the vehicle with their smartphone, this could be deemed tedious over time. The smartphone could therefore be replaced with custom hardware which has the GPS and internet connectivity needed for the cloud solution. The hardware could also be set up with a wake up function when movement is detected, and then automatically send and receive data over OBD-II and the cloud platform.

One limitation that was found with the system was the updating frequency allowed on the ThingSpeak server. This could be addressed by creating a custom server solution for the project that can receive and visualise the collected data. This continuation of the project would also result with less restrictions whats able to be done than the ThingSpeak server.

With the designed system a large amount of data from car journeys can be logged. One important measurement that cannot be made using OBD-II is CO_2 emissions. Journeys that are combined with a CO_2 measurement device on the exhaust pipe of the vehicle to be logged would make it possible to use machine learning to identify a correlation with OBD-II data and CO_2 emissions.

7 References

[1] OECD/ITF Working Group, *Reducing transport ghg emissions - opportunities and costs,* International Transport Forum, Jan. 2009. [Online]. Available: https://www.internationaltransportforum.org/Pub/pdf/09GHGsum.pdf (visited on 06/18/2022).

[2] M. Falch, *Obd2 explained - a simple intro [2022]*, CSS electronics, 2022. [Online]. Available: https://www.csselectronics.com/pages/obd2-explained-simple-intro#obd2-pid-frame (visited on 05/15/2022).

[3] MathWorks Inc., *Thingspeak: The iot platform with matlab analytics*, MathWorks Inc., 2022. [Online]. Available: https://se.mathworks.com/help/thingspeak/ (visited on 06/18/2022).

[4] H. Zhang, X. Meng, X. Zhang, and L. Zhenglin, "Cansec: A practical in-vehicle controller area network security evaluation tool," *Sensors*, vol. 20, p. 4900, Aug. 2020. DOI: 10.3390/s20174900.

[5] R. Zalman, *Rugged Embedded Systems*, A Vega, P Bose, A Buyuktosunoglu, Ed. 2017, ch. 8 - Rugged autonomous vehicles, pp. 237–266, ISBN: 978-0-12-802459-1.

[6] S. Corrigan, *Introduction to the controller area network (can)*, Texas Instruments, USA, Application Report, SLOA101B–August 2002–Revised May 2016., Texas Instruments, 2016. [Online]. Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1655958009568&ref_url=https%253A%252F%252Fwww.google.com%252F.

[7] International Organization for Standardization, *Iso 11898-1:2015: Road vehicles — controller area network (can) — part 1: Data link layer and physical signalling*, International Organization for Standardization, 2015. [Online]. Available: https://www.iso.org/standard/63648.html (visited on 05/15/2022).

[8] Wikipedia, *Obd-ii pids*, Wikipedia Foundation, 2022. [Online]. Available: https://en.wikipedia.org/wiki/OBD-II_PIDs (visited on 06/18/2022).

[9] Malekian, Reza and Moloisane, Ntefeng Ruth and Nair, Lakshmi and Maharaj, B. T. and Chude-Okonkwo, Uche A. K., "Design and implementation of a wireless obd ii fleet management system," *IEEE Sensors Journal*, vol. 17, no. 4, pp. 1154–1164, 2017. DOI: 10.1109/JSEN.2016.2631542.

[10] Mozilla developer, *An overview of http*, Mozilla Foundation, 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (visited on 06/18/2022).

[11] ELM Electronics, *Obd to rs232 interpreter*, ELM327 datasheet, ELM327DSJ–July 2016–Revision 2.4, ELM Electronics, 2017. [Online]. Available: https://www.elmelectronics.com/wp-content/uploads/2016/07/ELM327DS.pdf (visited on 06/25/2022).

[12] Guangzhou HC Information Technology Co, *Product data sheet*, Revision 2.2, HC-06 datasheet, 2011. [Online]. Available: https://www.olimex.com/Products/Components/RF/BLUETOOTH-SERIAL-HC-06/resources/hc06.pdf (visited on 06/25/2022).

[13] Android developer, *Meet android studio*, Google, 2022. [Online]. Available: https://developer.android.com/studio/intro (visited on 04/10/2022).

[14] Androld developer, *Bluetooth overview*, Google, 2022. [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth (visited on 06/18/2022).

[15] P. Leach, M. Mealling and R. Salz, *A universally unique identifier (uuid) urn namespace*, The Internet Society, 2005. [Online]. Available: https://www.ietf.org/rfc/rfc4122.txt (visited on 06/18/2022).

[16] Mozzilla developer, *Regular expressions*, Mozilla Foundation, 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions (visited on 06/18/2022).

[17] Android developer, *Build location-aware apps*, Google, 2022. [Online]. Available: https://developer.android.com/training/location (visited on 06/18/2022).

[18] Volley developer, *Volley overview*, Volley developer, 2021. [Online]. Available: https://google.github.io/volley/ (visited on 06/18/2022).

[19] MathWorks Inc., *Geoscatter*, MathWorks Inc., 2022. [Online]. Available: https://se.mathworks.com/help/matlab/ref/geoscatter.html (visited on 06/18/2022).

[20] M Nasir, R Noor, M. Kalam and B. Masum, "Reduction of fuel consumption and exhaust pollutant using intelligent transport systems," *The Scientific World Journal*, M. Ferrari, Ed., 2014. DOI: https://doi.org/10.1155/2014/836375.