



Dynamic Car Information

Project Engineering

Year 4

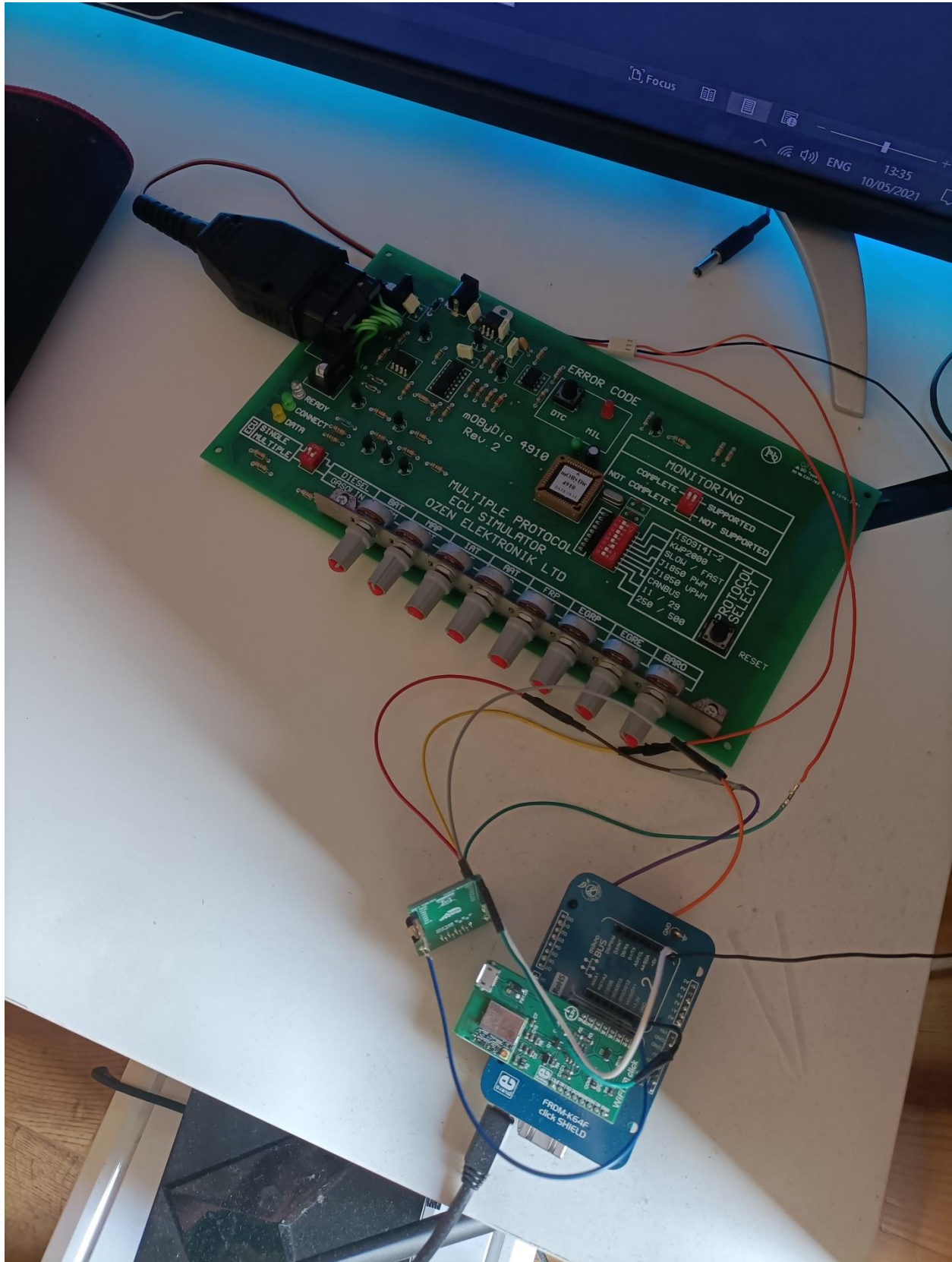
Sean O'Shaughnessy

Bachelor of Engineering (Honours) in Software and

Electronic Engineering

Galway-Mayo Institute of Technology

2020/2021



Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Sean O'Shaughnessy (G00358883) 17/05/2021

Acknowledgements

I would like to thank Niall O'Keefe, Michelle Lynch, Paul Lennon, and Brian O'Shea for their help with my project.

They answered any questions I had in detail and helped me with different types of challenges I faced throughout the year completing my project.

Table of Contents

1	Summary.....	6
2	Poster.....	7
3	Introduction.....	8
3.1	Project Goals:	8
3.2	Project Motivation:	8
3.3	Overview	8
4	Background.....	9
4.1	On-Board Diagnostics.....	9
4.2	Automotive Can Bus.....	9
4.3	Free RTOS	10
4.4	Amazon Web Services (AWS) IoT Device Shadow	10
4.5	Android Application	11
5	Project Architecture	12
6	Project Plan.....	14
7	Technologies.....	17
7.1	On-Board-Diagnostics (OBD).....	17
7.2	Can Bus.....	18
7.3	Free RTOS	24
7.4	Amazon Web Services (AWS) IOT Core.....	26
7.5	Android Application	29
8	Ethics.....	32
9	Conclusion	33
10	References	34

1 Summary

Ever wanted to monitor important information from your car without being in it?

Important information will be gathered from a car's Electronic Control Unit (ECU) to allow the user to monitor their car from the comfort of the driver's seat or their home. This is more aimed to performance cars that have turbo charged engines as the tolerance for error is little to none. It would also be a very useful tool for fleet owners such as, taxis or trucks as the owner can easily monitor each vehicle without being present and they can see if there is any foul play with their vehicles. Oil Pressure, oil temperature and water temperature are some of the important aspects that would need to be monitored in a car/truck.

The OBD (On-Board Diagnostic) port would be used to access the ECU and CAN bus will be used to communicate with the ECU, the data from the vehicle can be monitored. I would then use the cloud to store the data that is retrieved from the ECU of the car as this would allow the user to access it from anywhere that you have internet connection. An android application will allow the user to view data that is recorded.

This complete package it very easy to install into a vehicle and the company can use a portable router that the device can connect to so it can be always online and sending the valuable data to the owner. Foul play to vehicles can cost fleet owners thousands, as well it gives peace of mind to owners of performance cars as they can easily monitor their vehicle.

2 Poster

Dynamic Car Information

Sean O'Shaughnessy
Bachelor of Engineering (Honours) in Software and Electronic Engineering

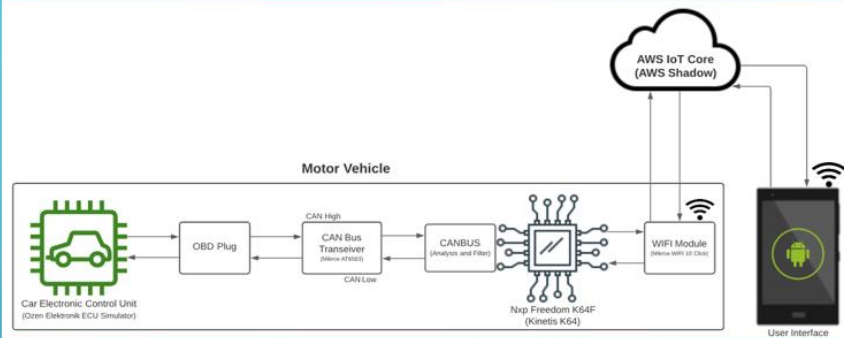
Ever wanted to monitor important information from your car without being in it?

Technologies:

- Can Bus
- AWS
- OBD
- Free RTOS

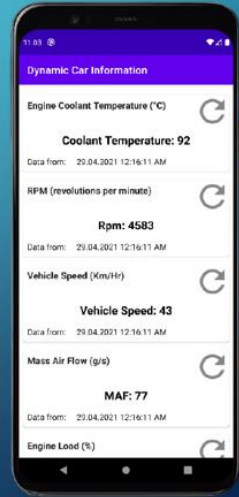
Hardware:

- NXP Freedom 64F
- Multiple Protocol OBD Ecu Simulator
- Wi-Fi Interface chip
- Can Bus Transceiver
- Mobile Phone (Android)



The diagram illustrates the system architecture. A 'Motor Vehicle' block contains a 'Car Electronic Control Unit (Ozen Elektronik ECU Simulator)' connected to an 'OBD Plug'. The OBD Plug is connected to a 'CAN Bus Transceiver (MCP2150)' which interfaces with the 'CAN High' and 'CAN Low' lines. These lines connect to a 'CANBUS (Analysis and Filter)' block, which then connects to an 'Nxp Freedom K64F (Kinetics K64)' microcontroller. The microcontroller is connected to a 'WiFi Module (Mikrotik Wifidino 2P)' which communicates with an 'AWS IoT Core (AWS Shadow)' cloud service. A 'User Interface' (represented by a smartphone) is connected to the cloud service via a wireless signal.

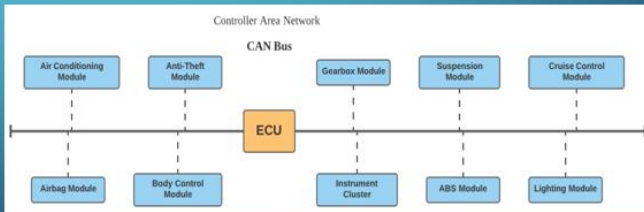
Results:




The screenshot shows the 'Dynamic Car Information' app interface on a smartphone. It displays the following data: Engine Coolant Temperature (°C) at 92, RPM (revolutions per minute) at 4583, Vehicle Speed (Km/Hr) at 43, Mass Air Flow (g/s) at 77, and Engine Load (%). Each data point includes a timestamp from 29.04.2021 12:16:11 AM and a refresh icon.


Project Summary:

Important information such as RPM, oil pressure, oil temperature can be measured from a vehicle's ECU (Electronic Control Unit) that will allow the user to monitor the vehicle without being in the vehicle. This can also aid fleet owners to monitor their trucks or vehicles on the go and be aware of any foul play to their property.



The diagram shows a 'Controller Area Network' (CAN Bus) with an 'ECU' at the center. Various modules are connected to the ECU via the CAN Bus, including: Air Conditioning Module, Anti-Theft Module, Gearbox Module, Suspension Module, Cruise Control Module, Airbag Module, Body Control Module, Instrument Cluster, ABS Module, and Lighting Module.





**Department of
Electronic and
Electrical Engineering**

3 Introduction

3.1 Project Goals:

- To create a portable, device that can make owning your car/cars safer by monitoring valuable data.
- To help car enthusiasts monitor their vehicle, allowing the oil to properly warm up whilst keep an eye on boost levels and water temperatures.
- Help fleet managers to monitor their vehicles and prevent or catch any foul play to the vehicles by members of staff.
- Ease of use so it can be a plug and play device with minimal setup needed and once internet connection is available it keeps posting data to the cloud.

3.2 Project Motivation:

My motivation for my project is that I have a great interest in the automotive industry and this project can help both car enthusiasts fleet owners monitor their vehicles. In Europe there are more than 35 million commercial vehicles and buses whilst there are over 245 million cars in the European union alone [1].

As I am a car owner and a car enthusiast, I believe that a project like this can help many people and vehicle owners as well it can highlight areas where preventative maintenance is needed. It can highlight damages that are caused to vehicles in fleet by foul play by staff by over revving the engine or not staying in the optimal fuel zone to save fuel and costing the company more.

These problems got me thinking into how I could create a simple way to monitor the vehicle without being near the vehicle.

3.3 Overview

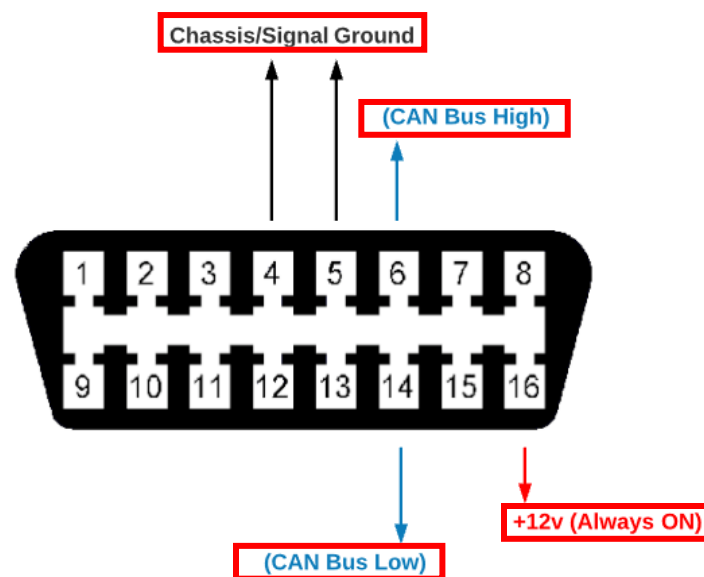
My final year project report contains all the work that I have completed throughout the year, and it explains the main areas of the project in detail which are Can Bus, Amazon Web Services, Android Application, On-Bored Diagnostics and Free RTOS.

4 Background

4.1 On-Board Diagnostics

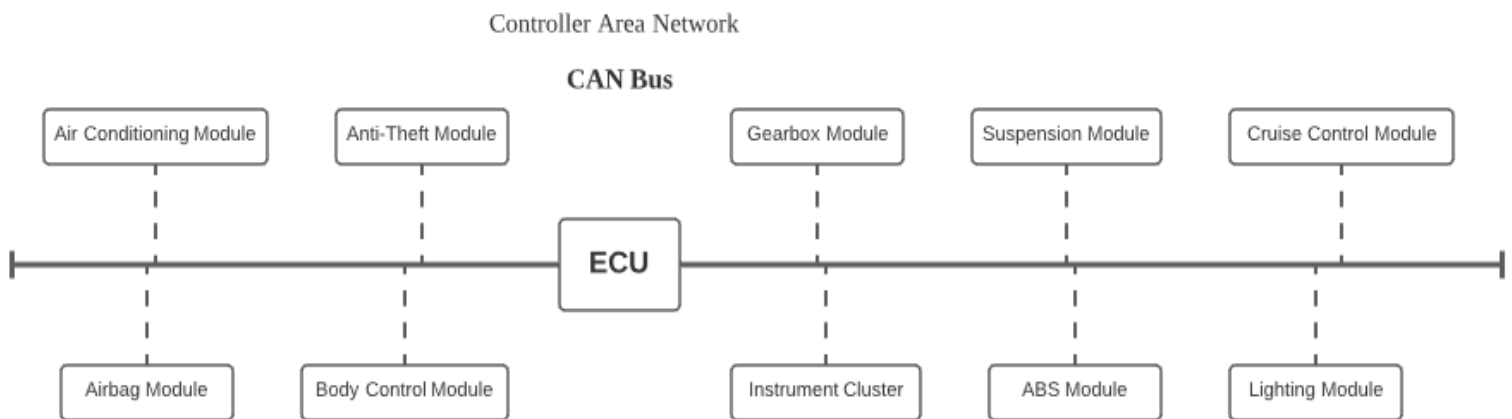
On Board Diagnostics (OBD) is an automotive electronic system that provides self-diagnostics and fault reporting capabilities. This gives the user access to the car's subsystem information allowing them to monitor and analyze the cars data. The data is generated by the Electronic Control Unit within the vehicle.

Pictured below is the standardized 16 pin ODB plug that is used to communicate [2].



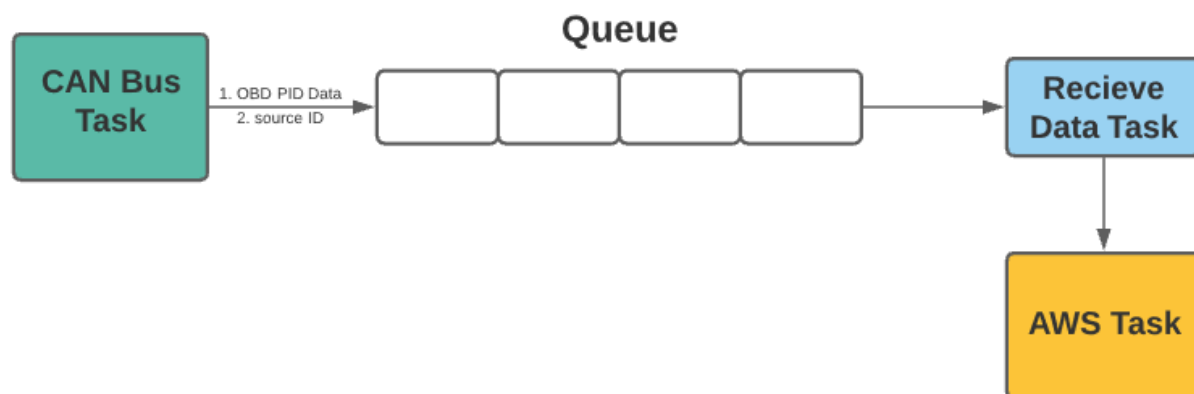
4.2 Automotive Can Bus

- CAN bus is a method of communication to the vehicle's Electronic Control Unit (ECU).
- ISO 15765 is a standardisation for CAN bus in automotive vehicles.
- Controller Area Network (CAN) is a robust vehicle communication standard designed to allow micro controllers and other devices to communicate with other applications without a host computer [3].



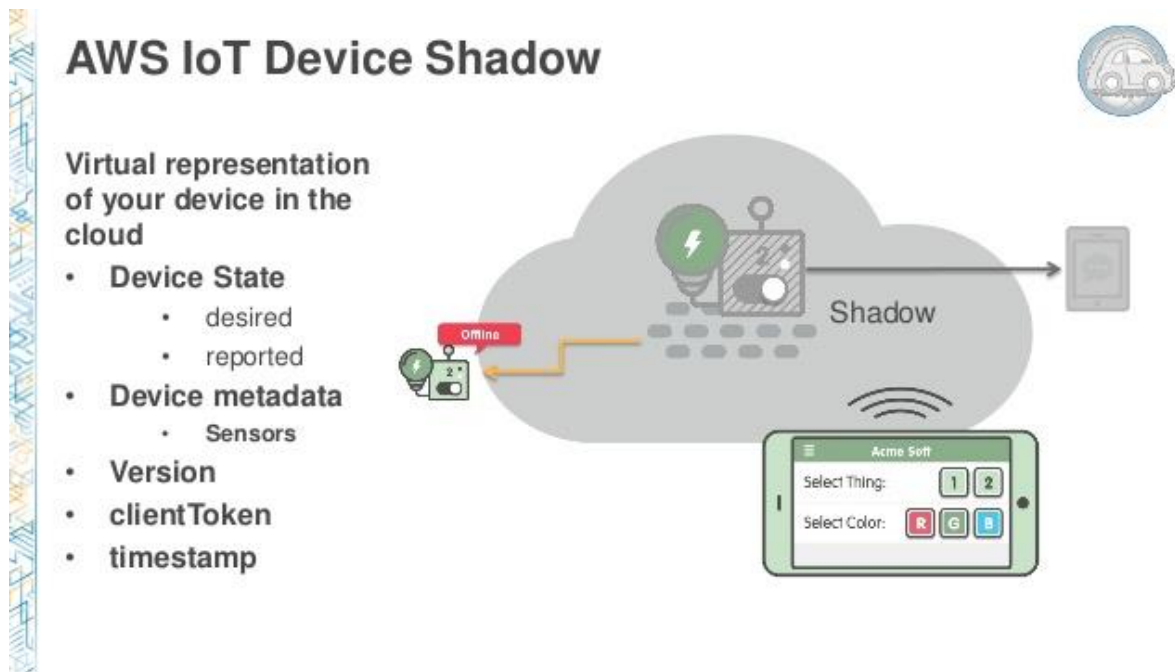
4.3 Free RTOS

I am using the NXP Freedom K64F board to complete my task which are communicating with AWS and retrieving data from the ECU Simulator. I need to use Free RTOS Queues as this will allow me to transfer data between tasks [4].



4.4 Amazon Web Services (AWS) IoT Device Shadow

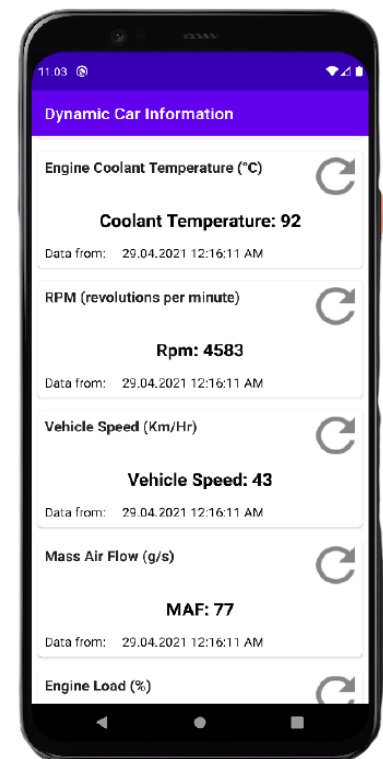
The AWS IoT Device Shadow allows the device's state to be available to apps and other services whether the device is connected to AWS IoT or not. A Shadow basically acts as a buffer for the device as the data is temporarily stored until it is overwritten with data. For example, an android application can 'Subscribe' to the IoT Device's Shadow and retrieve the data as it is being 'Published' to AWS IoT. The 'Shadow' that is created will allow the IoT Device 'Publishes' data to that particular 'Shadow' and then the android application can send updates on the state of the IoT device and via the 'Shadow' and will receive the state that they requested if available [5].



4.5 Android Application

To display the data that is being sent to AWS, I opted to go with an android application, using Android Studio to create the application.

Public key cryptography is used to allow the messages to be transferred from AWS to the android application. The message is encrypted using a public key and the only way that this message can be decrypt is by using the corresponding private key [6].



5 Project Architecture

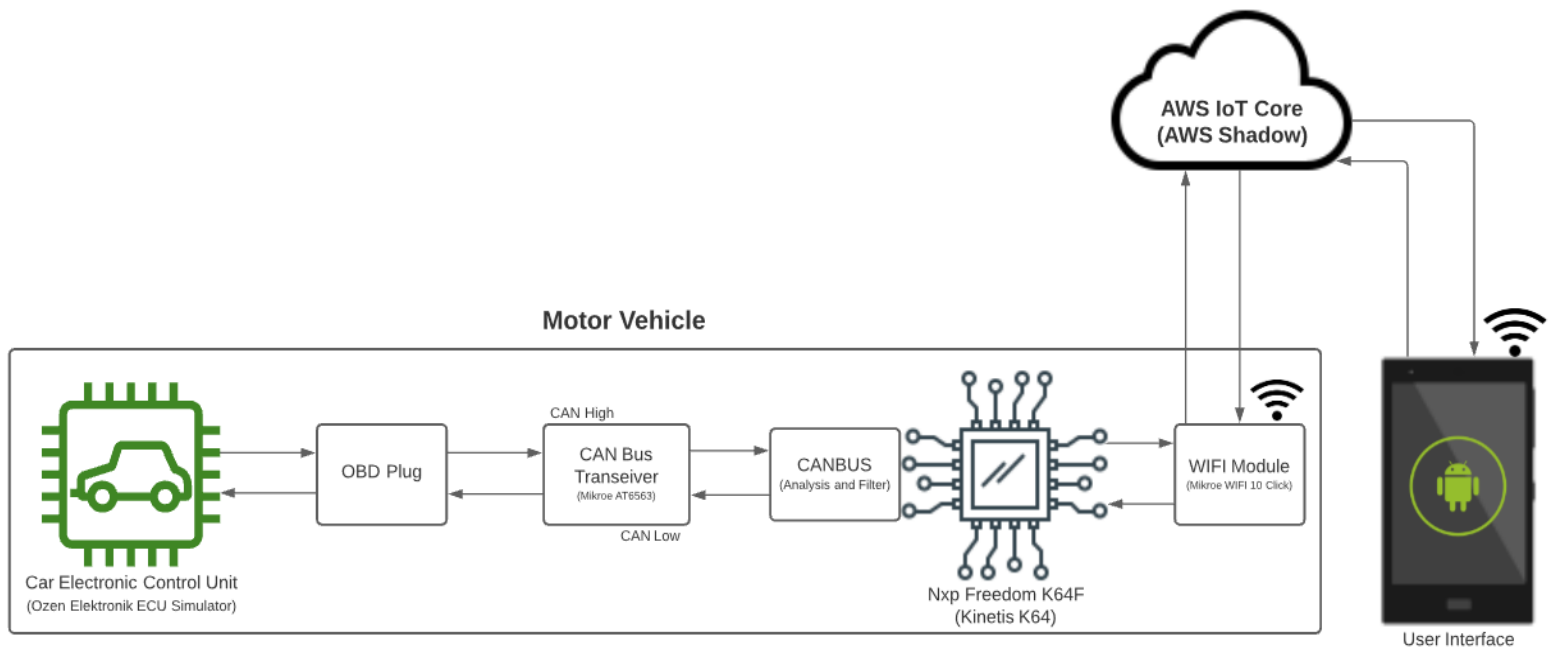


Figure 5-1 Architecture Diagram

For my final year project, I am using a NXP Freedom K64-F development board and a Ozen Electronic On-Board Diagnostic (OBD) Simulator that is used to simulate a vehicles Electronic Control Unit. I will be Using Can Bus as the form of communication, I need to use a transceiver to drive and detect data on the bus, for this task I am using the Mikroe ATA6563 transceiver. To enable the Freedom K64-F to connect to the internet I am using the Mikroe WIFI 10 Click board which allows me to use Amazon Web Services (AWS).

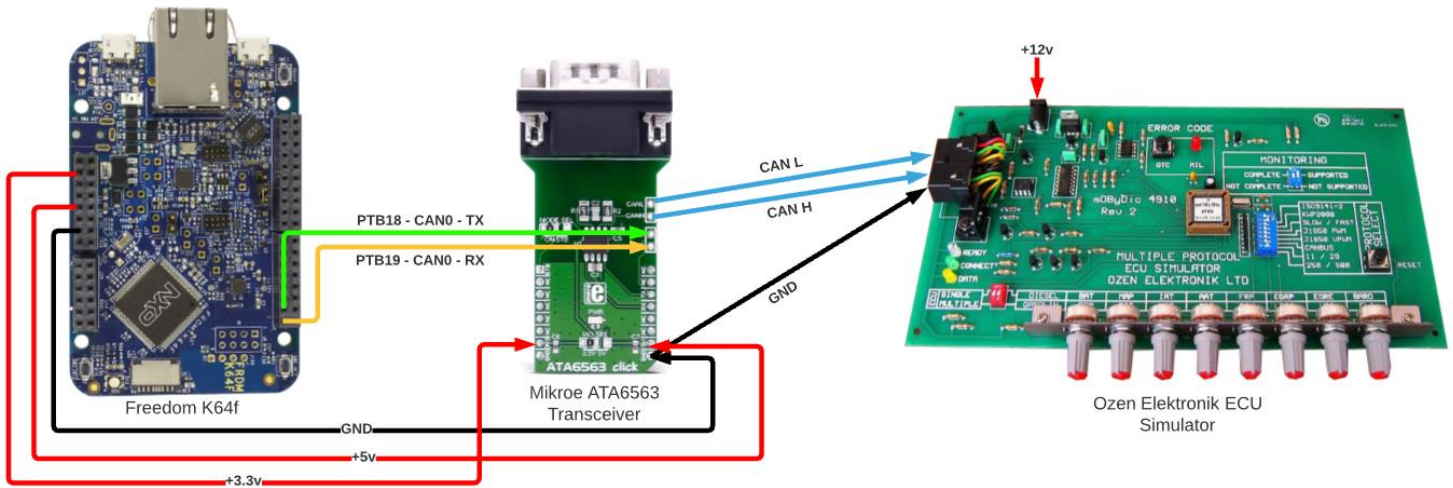


Figure 5-2 Can bus Schematic.

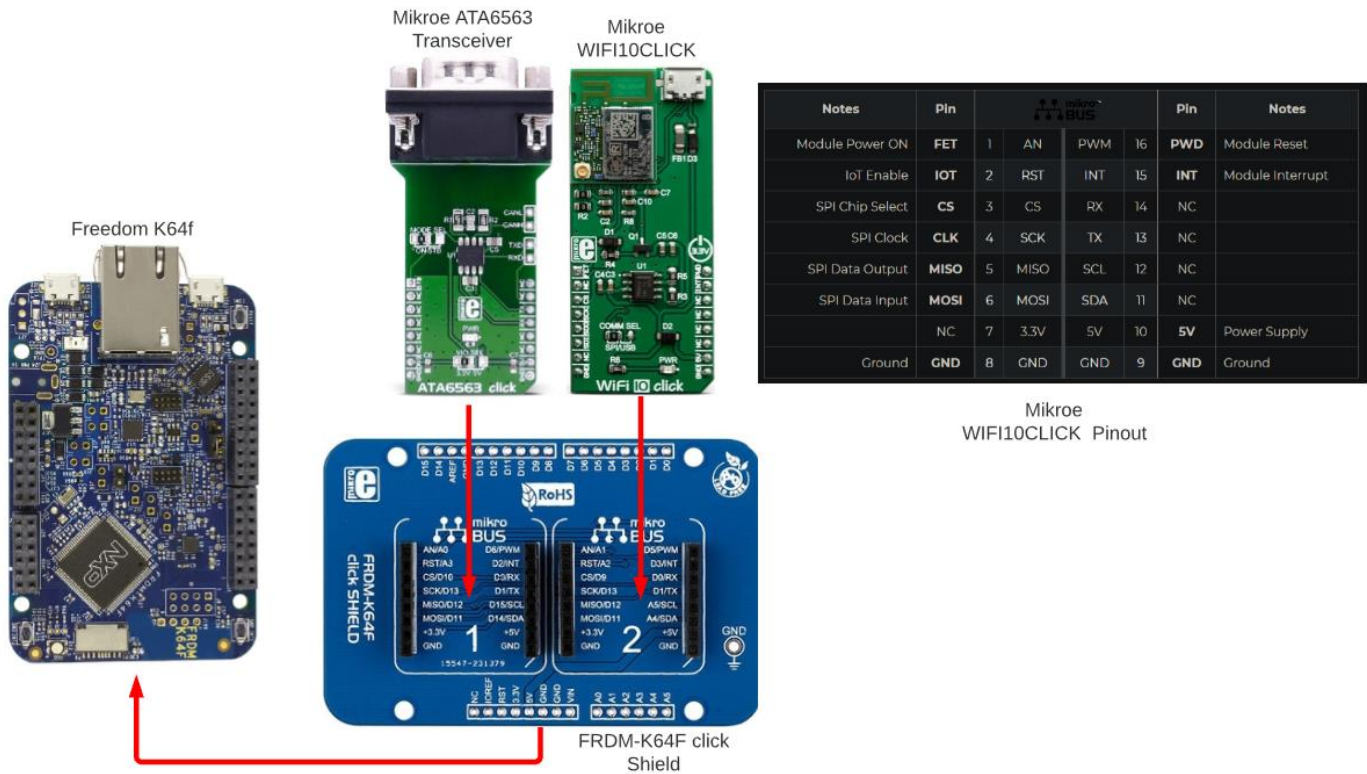


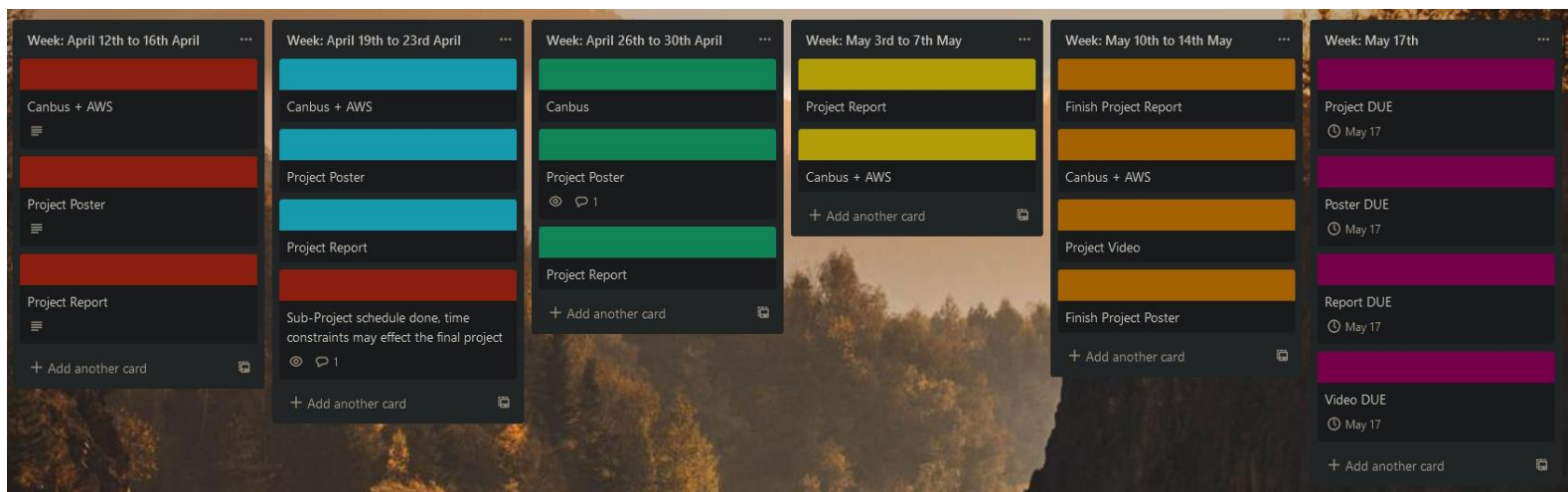
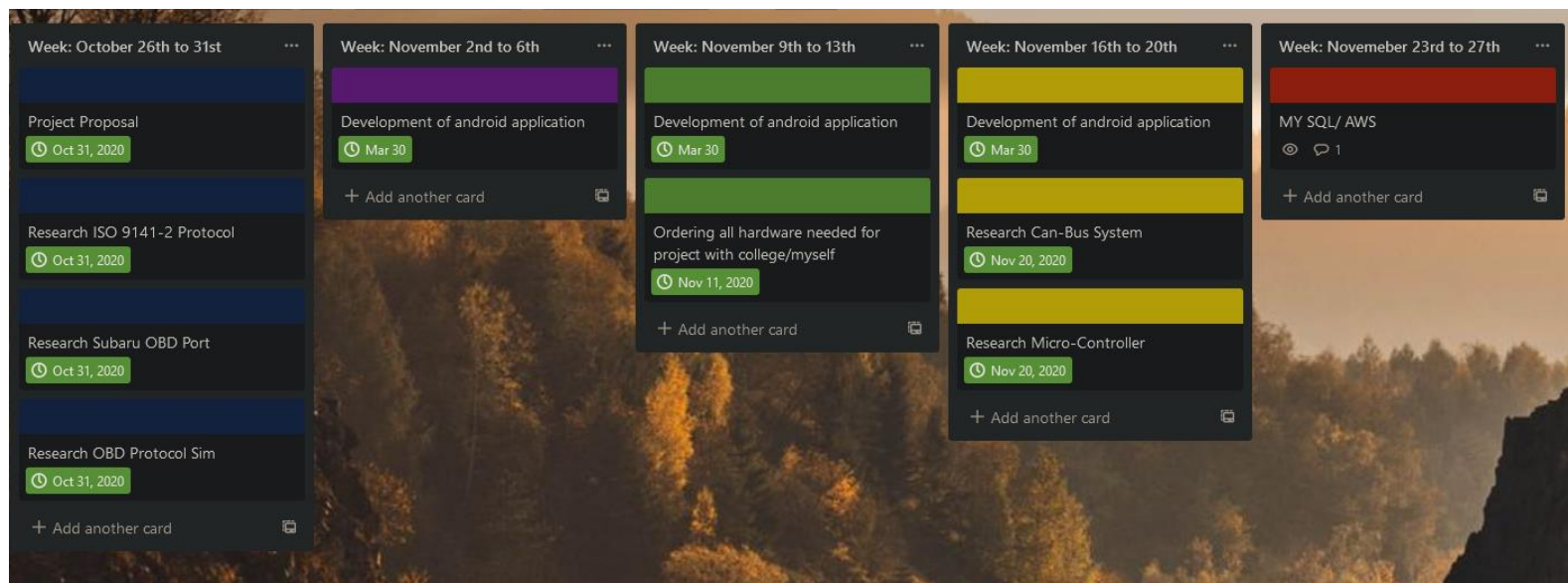
Figure 5-3 WIFI + Can bus Transceiver.

6 Project Plan

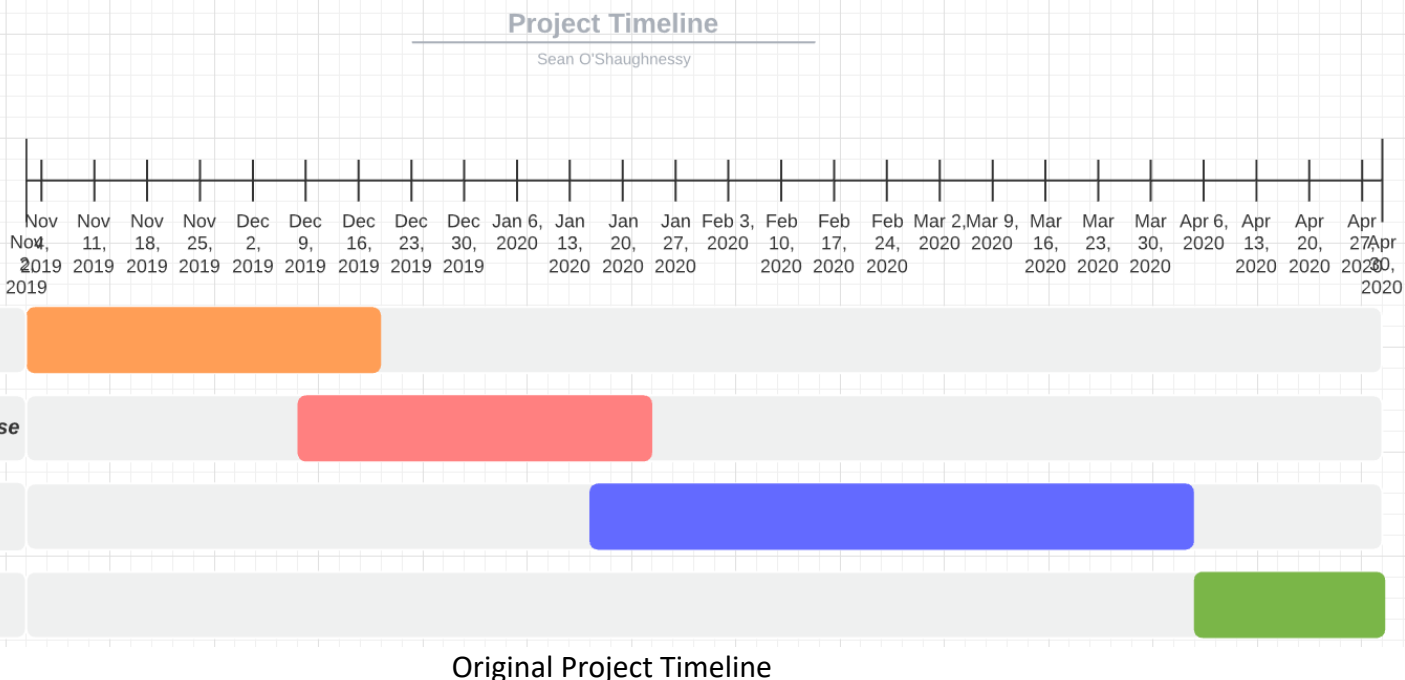
To plan my project from the beginning I used Trello as I was already used to using Trello for other projects and I found it very easy to use and keep track of important goals in the project.

I have included some snippets of my Trello board; the first snippet is the start of the project, and the second snippet is the end of the project. This gives you an idea of how the project progressed throughout the weeks. In the Trello board I also allowed time at the start of the project for researching CAN and Amazon Web Services. I set goals where I thought I could meet the goals but sometimes the timeline did not go to plan, and I learned a lot about time-management doing this project and it is an area that I can improve on for the future.

<https://trello.com/b/zYB45MHZ/4th-year-project>



This was the original Timeline that I set out at the start of the project. This Timeline gave me a good understanding of where I needed to be, but I did switch to Trello so I could set exact tasks for each week, but I referred to this Timeline to see how I was progressing through the project.



I reviewed the timeline again when the final data was getting closer to really understand where I was with the project and if I would have enough time to complete it what I wanted to get done for the demonstration as we had exams coming up that I had to set aside time to study for said exams. In my opinion it really helped doing this final timeline as I kept to what I said each week and got the project where I wanted it to be for the demonstration, but I did not get everything what I wanted to get done but I got the project to a demonstrable state that I was happy with but there is room for future improvements.

Reviewed Timeline:

April: 19th to 23rd:

(C++ Quiz + Cloud Computing Project Due)

- Finish Can bus side.

April: 26th to 30th:

(DSP Quiz + Cloud Computing Quiz + Enterprise & Innovation Team Presentation + Enterprise & Innovation Assignment Due)

- Finish Can bus side.
- Project Report

May: 3rd to 7th:

(C++ Exam + MAD Exam)

- Maybe Join AWS & Can bus if time permits.
- Project Report

May: 10th to 14th:

(DSP Exam & Enterprise and Innovation Business Plan Due)

- Record Videos for Project.
- Finalise Poster and Report.

May: 17th:

- All Project Deliverables Due.

7 Technologies

In this section I will go through some of the technologies that I used throughout my project. The main technologies that are present in my project are CAN bus, On-Board Diagnostics, AWS IoT Core, Android Application and Free RTOS.

7.1 On-Board-Diagnostics (OBD)

On Board Diagnostics (OBD) is an automotive electronic system that provides self-diagnostics and fault reporting capabilities. This gives the user access to the car's subsystem information allowing them to monitor and analyze the cars data. The data is generated by the Electronic Control Unit (ECU) within the vehicle.

I needed to research about the OBD parameter IDs as these are used to request data from the ECU. I referred to the listed PIDs on Wikipedia [7] that helped me find the correct address that I should be requesting depending on what parameter I want to read from. It also contained the formulas needed to calculate the correct value. Pictured below is the format of a Bit-Encoded-Notation. This is useful for when I am calculating the formulas.

A								B								C								D							
A7	A6	A5	A4	A3	A2	A1	A0	B7	B6	B5	B4	B3	B2	B1	B0	C7	C6	C5	C4	C3	C2	C1	C0	D7	D6	D5	D4	D3	D2	D1	D0

Below are just some examples of the PIDs and their formulas [7]. I have explained the formulas in the CAN Bus section.

PIDs (hex)	PID (Dec)	Data bytes returned	Description	Min value	Max value	Units	Formula[s]
------------	-----------	---------------------	-------------	-----------	-----------	-------	------------

Engine Load:

04	4	1	Calculated engine load	0	100	%	$\frac{100}{255}A$ (or $\frac{A}{2.55}$)
----	---	---	------------------------	---	-----	---	---

Engine Speed (rpm):

0c	12	2	Engine speed	0	16,383.75	rpm	$\frac{256A + B}{4}$
----	----	---	--------------	---	-----------	-----	----------------------

Vehicle Speed:

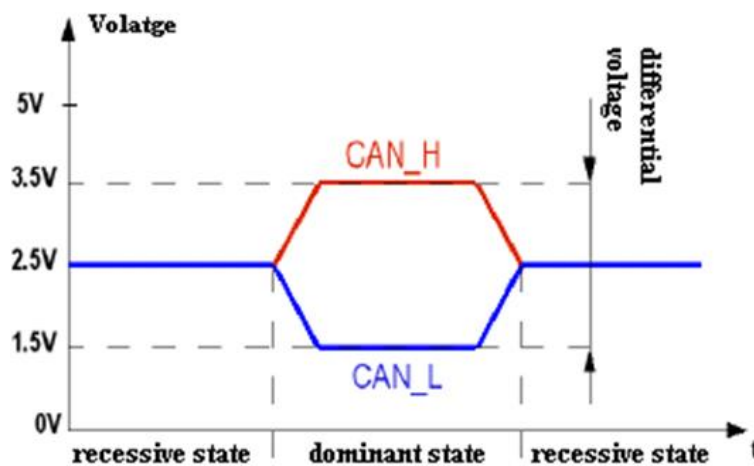
0d	13	1	Vehicle speed	0	255	km/h	A
----	----	---	---------------	---	-----	------	---

The first column (PIDs) is the ID of the corresponding parameter. The formula is the last column (formula). Also listed is the min and max values which are important for testing purposes and the units that the parameter is read in.

7.2 Can Bus

Controller Area Network is (CAN Bus) a method of communication to a vehicle's Electronic Control Unit (ECU). The CAN bus system enables ECUs to communicate with other ECU's that may be present as this allows them to share any information to other 'nodes' [8].

The CAN bus system enables each ECU to communicate with all other ECUs - without complex dedicated wiring. Specifically, an ECU can prepare and broadcast information (e.g., sensor data) via the CAN bus that consist of two wires, CAN low and CAN high. The broadcasted data is accepted by all other ECUs on the CAN network - and each ECU can then check the data and decide whether to receive or ignore it [8].



CAN uses a differential signal with two logic states, called recessive and dominant. Recessive indicates that the differential voltage is less than a minimum threshold voltage. Dominant indicates that the differential voltage is greater than this minimum threshold. The dominant state is achieved by driving a logic '0' onto the bus, while the recessive state is achieved by a logic '1'. This is inverted from the traditional high and low used in most systems.

I used an example project that is included with the NXP Freedom K64F and built/changed that example project to suit my needs.

Firstly, the receive and transmit CAN frames must be configured. The receive frame and transmit frames are simple to configure but the transmit frame contains a data frame. The ID of the frames are set in the configuration of the frames as ID's are very important in CAN as the ECU will only respond to a correct ID.

```
/* Setup Rx Message Buffer. */
const flexcan_rx_mb_config_t rxMbConfig = {
    .id = FLEXCAN_ID_STD(0x7E8),
    .format = kFLEXCAN_FrameFormatStandard,
    .type = kFLEXCAN_FrameTypeData
};
```

- Configuration for the receive message buffer that has an ID of (0x7E8).

The transmit frame is setup inside of a function as we will be sending a different data frame depending on what PID we want to receive data from. The setup of the transmit message buffer is similar, the data frame and the message buffer index is passed into the function and then we call this function and pass different data frames to it that I want to read from. A blocking function is used to send the frame and will stay blocked until an interrupt occurs on the receive message buffer.

```
uint8_t Tx_Frame(uint16_t ID, uint8_t *dataFrame, uint8_t msgBuffIndex) {
    flexcan_frame_t txFrame;
    txFrame.format = (uint8_t)kFLEXCAN_FrameFormatStandard;
    txFrame.type = (uint8_t)kFLEXCAN_FrameTypeData;
    txFrame.id = FLEXCAN_ID_STD(ID);
    txFrame.length = 8;

    txFrame.dataByte0 = dataFrame[0];
    txFrame.dataByte1 = dataFrame[1];
    txFrame.dataByte2 = dataFrame[2];
    txFrame.dataByte3 = dataFrame[3];
    txFrame.dataByte4 = dataFrame[4];
    txFrame.dataByte5 = dataFrame[5];
    txFrame.dataByte6 = dataFrame[6];
    txFrame.dataByte7 = dataFrame[7];

    CANFrameReceived = 0;

    printf("\rSending CAN Tx Frame\n\r");

    FLEXCAN_TransferSendBlocking(CAN0, msgBuffIndex, &txFrame);

    printf("CAN TX Frame Sent\n\r");
    return 1;
}
```

Pictured below is each of the data frames that have a different data frames, and the ECU will respond to that frame with the corresponding data, for example a request is sent for coolant temperature with the address 0x05, the ECU will respond with the value for the coolant temperature.

```
uint8_t coolantTemp_Request[8] = {0x2,0x1,0x05,0x55,0x55,0x55,0x55,0x55};
uint8_t RPM_Request[8] = {0x2,0x1,0x0C,0x55,0x55,0x55,0x55,0x55};
uint8_t speed_Request[8] = {0x2,0x1,0x0D,0x55,0x55,0x55,0x55,0x55};
uint8_t MAF_Request[8] = {0x2,0x1,0x10,0x55,0x55,0x55,0x55,0x55};
uint8_t engineLoad_Request[8] = {0x2,0x1,0x04,0x55,0x55,0x55,0x55,0x55};
```

```
CAN_dataFrame[5] = {coolantTemp_Request, RPM_Request, speed_Request, MAF_Request, engineLoad_Request};
```

The TX Frame function is called and the frame ID, data frame and the message buffer index is passed to the function and the ECU will respond if the frame ID and the right frame is transmitted. CAN_dataFrame[i] stores the 5 data frames and I use a loop, so it sends a request for each of the OBD parameters.

```
Tx_Frame(0x7DF, CAN_dataFrame[i], 1);
```

```
/* CAN0_Ored_Message_buffer_IRQn interrupt handler */
void CAN0_CAN_ORED_MB_IRQHANDLER(void) {
    uint8_t x;

    printf("Message Buffer Status Flag: %d\n\r", FLEXCAN_GetMbStatusFlags(CAN0, 0x01));

    if(FLEXCAN_GetMbStatusFlags(CAN0, 0x01) == 1){
        FLEXCAN_ClearMbStatusFlags(CAN0, 0x01);
        FLEXCAN_ReadRxMb(CAN0, 0 , &rxFrame);

        rxBuffer[0] = rxFrame.dataByte0;
        rxBuffer[1] = rxFrame.dataByte1;
        rxBuffer[2] = rxFrame.dataByte2;
        rxBuffer[3] = rxFrame.dataByte3;
        rxBuffer[4] = rxFrame.dataByte4;
        rxBuffer[5] = rxFrame.dataByte5;
        rxBuffer[6] = rxFrame.dataByte6;
        rxBuffer[7] = rxFrame.dataByte7;
        for(x = 0; x < 8; x++) {
            printf("%02x\t", rxBuffer[x]);
        }
        printf("\n\n\r");
        CANFrameReceived = 1;
    }
}
```

An Interrupt is used for the receive message buffer and sets a message buffer status flag to 1 when a response message is received. We then check this flag to make sure that a message was received and if it was we clear the message flag by using `FLEXCAN_ClearMbStatusFlags()`. 0x01 is the message buffer index as you can have multiple message buffers. The `FLEXCAN_ReadRxMb` reads the message received into a frame and then when can read out the data that was received using the `rxBuffer` by iterating through the buffer and printing the data in the terminal that is in the Buffer.

```
switch(rxBuffer[2]){
case 0x05:
    printf("Coolant Temperature: %d Degrees Celsius\n\n\r", (rxBuffer[3]));
    canbus_struct.sourceID = 0;
    canbus_struct.dataField = (rxBuffer[3]);
    xQueueSend(canbusQueue, &canbus_struct, 0);
    break;

case 0x0C:
    printf("RPM: %d revolutions per minute\n\n\r", ((uint16_t)rxBuffer[3]*256+rxBuffer[4])/4);
    canbus_struct.sourceID = 1;
    canbus_struct.dataField = (((uint16_t)rxBuffer[3]*256+rxBuffer[4])/4);
    xQueueSend(canbusQueue, &canbus_struct, 0);
    break;

case 0x0D:
    printf("Speed: %dkm/h\n\n\r", rxBuffer[3]);
    canbus_struct.sourceID = 2;
    canbus_struct.dataField = (rxBuffer[3]);
    xQueueSend(canbusQueue, &canbus_struct, 0);
    break;

case 0x10:
    printf("MAF: %d grams/sec\n\n\r", ((uint16_t)rxBuffer[3]*256+rxBuffer[4])/100);
    canbus_struct.sourceID = 3;
    canbus_struct.dataField = (((uint16_t)rxBuffer[3]*256+rxBuffer[4])/100);
    xQueueSend(canbusQueue, &canbus_struct, 0);
    break;

case 0x04:
    printf("Engine Load: %d %%\n\n\r", ((uint16_t)rxBuffer[3])*100/255);
    canbus_struct.sourceID = 4;
    canbus_struct.dataField = (((uint16_t)rxBuffer[3])*100/255);
    xQueueSend(canbusQueue, &canbus_struct, 0);
    break;

default:
    break;
}
```

When the data has been received, we need to do some maths on the buffer that is in the receive frame. A switch case is used on rxBuffer [2] as this section holds the address. Depending on what address is received the cases are set up and return the correct equation to display the right data. Then the data is printed out onto the terminal and then using queues the data is sent to another task that allows it to be sent via AWS.

```
uint8_t coolantTemp_Request[8] = {0x2,0x1,0x05,0x55,0x55,0x55,0x55,0x55};
rxBuffer [8] =
```

[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]

Above is an example of the data frame that contains the parameters for a coolant temperature request. I have laid out the rxBuffer underneath to help when explaining the formula to get the value.

When reading coolant temperature and speed they both just read from the rxBuffer [3] and do not need any formula applied to it.

```
(rxBuffer[3]).
```

$$A$$

When reading the data for calculating the revolutions per minute (rpm) the data is type casted to a uint16_t and read from rxBuffer [3]. The data from rxBuffer [3] is then multiplied by 256 and the data from rxBuffer [4] is added onto it. It is then all divided by 4.

```
(uint16_t)rxBuffer[3]*256+rxBuffer[4])/4)
```

$$\frac{256A + B}{4}$$

When reading the data for calculating mass air flow (MAF) the data is type casted to a uint16_t and read from rxBuffer [3]. The data from rxBuffer [3] is then multiplied by 256 and the data from rxBuffer [4] is added onto it. It is then all divided by 4. It is then all divided by 100.

```
(uint16_t)rxBuffer[3]*256+rxBuffer[4])/100).
```

$$\frac{256A + B}{100}$$

When reading the data for calculating engine load the data is type casted to a uint16_t and read from rxBuffer [3]. The data from rxBuffer [3] is then multiplied by 100 and the whole thing is then divided by 255.

```
(uint16_t)rxBuffer[3])*100/255)
```

$$\frac{100}{255} A \text{ (or } \frac{A}{2.55} \text{)}$$

Finally, I am using queues to send the data from the CAN bus task, a retrieve data task is used. When a queue is sent this task will wait until it receives the queue and take the data from the queue and store it in a variable and then AWS can send the data. IDs are used in a switch case to decipher which parameter it is. Depending on what ID it is the data from the data struct is saved in the corresponding variable.

```
static void retrieveData_task(void *pvParameters) {
    datastruct data_struct;

    while(1) {
        if(xQueueReceive(canbusQueue, &data_struct, portMAX_DELAY) == pdTRUE){
            printf("ID: %d Data: %d \n\r", data_struct.sourceID, data_struct.dataField);

            switch(data_struct.sourceID){

                case coolTemp_ID:
                    coolantTemp = data_struct.dataField;
                    break;

                case RPM_ID:
                    rpm = data_struct.dataField;
                    break;

                case speed_ID:
                    speed = data_struct.dataField;
                    break;

                case MAF_ID:
                    maf = data_struct.dataField;
                    break;

                case engineLoad_ID:
                    load = data_struct.dataField;
                    break;

                default:
                    break;
            }
        }
    }
}
```

7.3 Free RTOS

To pass the data between tasks I decided to use queues. Queues are used as a method of data communication between tasks. I did not fully get it working the way I wanted to due to time constraints, but I have queues implemented into my project. I have three tasks, CAN bus, AWS and retrieve data task. The CAN bus task reads the data from the ECU and uses a data struct to send the data. A data struct is used as we can add a source ID to the data that is being sent and identify it when we receive the data.

```
typedef struct {
    uint8_t sourceID;
    uint32_t dataField;
}datastruct;
```

- Example of the data struct in my project.

When the data is requested from the ECU and we get the response we use the data struct to assign a source ID and, in the data, struct we have a field for storing the data (dataField). I have defined the source IDs in my project and are pictured below. We use the xQueueSend function to send the data struct with the source ID and data, and we pass in the queue handle and the ticks to wait. We do this for each OBD parameter.

```
xQueueSend(canbusQueue, &canbus_struct, 0);

#define coolTemp_ID 0
#define RPM_ID 1
#define speed_ID 2
#define MAF_ID 3
#define engineLoad_ID 4

switch(rxBuffer[2]){
case 0x05:
    printf("Coolant Temperature: %d Degrees Celsius\n\n\r", (rxBuffer[3]));
    canbus_struct.sourceID = 0;
    canbus_struct.dataField = (rxBuffer[3]);
    xQueueSend(canbusQueue, &canbus_struct, 0);
    break;
```

When the queue is sent, we then use the xQueueReceive function that will block other tasks and run the retrieve data task. The data struct can then be read and using the source ID we can use a switch case to decipher where the data belongs to as read the data from the dataField.

```
(xQueueReceive(canbusQueue, &data_struct, portMAX_DELAY) == pdTRUE)
```



```
static void retrieveData_task(void *pvParameters) {  
    datastruct data_struct;  
  
    while(1) {  
        if(xQueueReceive(canbusQueue, &data_struct, portMAX_DELAY) == pdTRUE){  
            printf("ID: %d Data: %d \n\r" data_struct.sourceID, data_struct.dataField);  
  
            switch(data_struct.sourceID){  
  
                case coolTemp_ID:  
                    coolantTemp = data_struct.dataField;  
                    break;  
            }  
        }  
    }  
}
```

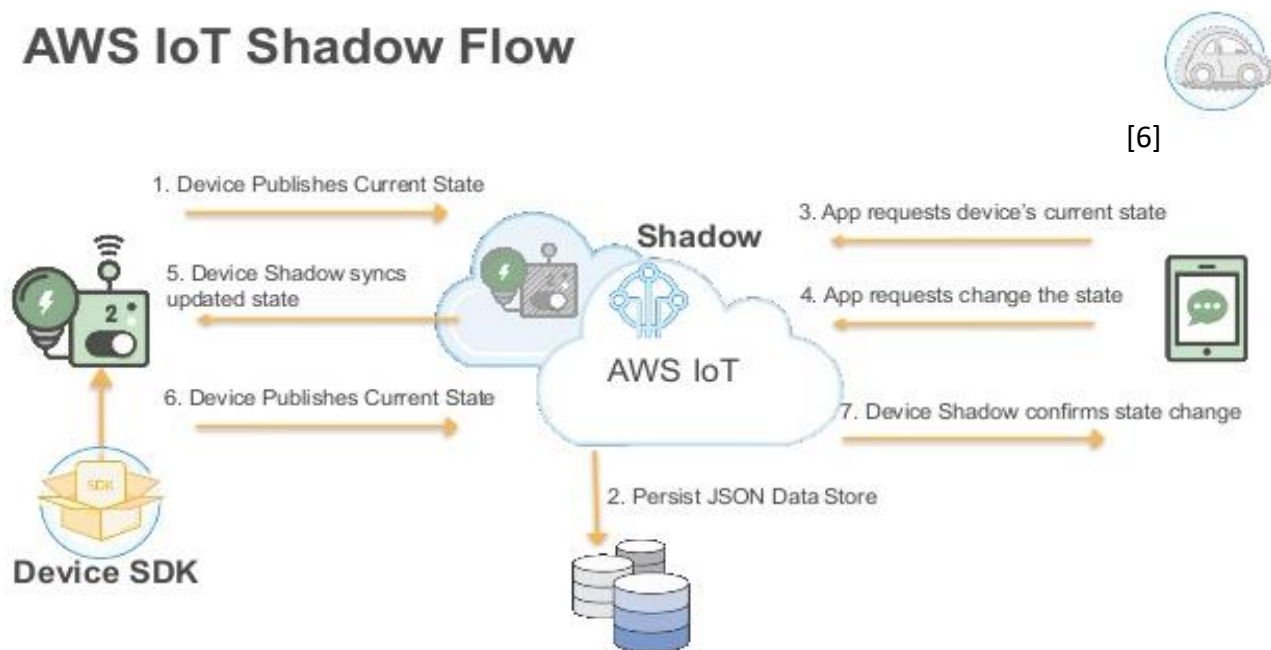
For future progression of this project regarding Free RTOS, I would remove the third task and only have the two task that pass data to each other. This was a problem I encountered in my project, and I would like to amend this problem in the future.

7.4 Amazon Web Services (AWS) IOT Core

Shadows provide a reliable data store for devices, apps, and other cloud services to share data. They enable devices, apps, and other cloud services to connect and disconnect without losing a device's state [9].

While devices, apps, and other cloud services are connected to AWS IoT, they can access and control the current state of a device through its shadows. For example, an app can request a change in a device's state by updating a shadow. AWS IoT publishes a message that indicates the change to the device. The device receives this message, updates its state to match, and publishes a message with its updated state. The Device Shadow service reflects this updated state in the corresponding shadow. The app can subscribe to the shadow's update, or it can query the shadow for its current state [9].

AWS IoT Shadow Flow



[6]

When a device goes offline, an app can still communicate with AWS IoT and the device's shadows. When the device reconnects, it receives the current state of its shadows so that it can update its state to match that of its shadows, and then publish a message with its updated state. Likewise, when an app goes offline and the device state changes while it is offline, the device keeps the shadow updated so the app can query the shadows for its current state when it reconnects [9].

On the Android application the refresh button basically will request the IoT device's current state by setting the desired shadows state to '1'.

```
bTempRefresh.setOnClickListener((v) -> {
    AwsShadow shadow = new AwsShadow();
    shadow.state.desired.tempUpdate = 1;

    // reported and metadata must not be sent along with desired shadow state
    shadow.state.reported = null;
    shadow.metadata = null;
```

This update is then 'published' to AWS IoT and the 'Thing' will keep checking for messages and will parse the Json that was published from the android application and, when it receives a '1', it will perform a shadow update so the android application will receive this update on the state that was requested. The app can subscribe to the shadow's updates so if the IoT 'Thing' publishes an update the app will automatically get it, but the refresh button is used to query the IoT 'Thing' on the OBD PID's.

```
/* process item from queue */
processShadowDeltaJSON(jsonDelta.pcDeltaDocument, jsonDelta.ulDocumentLength);

if (parsedtempState == 1)
{
    count ++;
    configPRINTF(("Update Coolant Temperature.\r\n"));
    printf("%s\n\r", pcUpdateBuffer);
    xOperationParams.ulDataLength = buildJsonTEMP();
    xReturn = SHADOW_Update(xClientHandle, &xOperationParams, shadowDemoTIMEOUT);
    if (xReturn == eShadowSuccess)
    {
        configPRINTF(("Successfully performed update.\r\n"));
    }
    else
    {
        configPRINTF(("Update failed, returned %d.\r\n", xReturn));
    }
    parsedtempState = 0;
```

The buildJsonTEMP() is a function that will build a json document with the reported state. For each of the OBD PID's there is a matching function to report the state of that parameter when it is requested by the android application. This function correctly parses the JSON document as there is a specific format for AWS.

```

/* Build JSON document with reported state of the "Coolant Temp" */
int buildJsonTEMP()
{
    char tmpBufcoolantTemp[128] = {0};

    sprintf(tmpBufcoolantTemp, "{\"temp\":{\"temp\":%d}}", coolantTemp);

    int ret = 0;
    ret = snprintf(pcUpdateBuffer, shadowBUFFER_LENGTH,
        "{\"state\":{\""
        "\"desired\":{\""
        "\"tempUpdate\":null"
        "\"reported\":%s},"
        "\"clientToken\": \"token-%d\"}"
        "\"tempUpdate\":%s}",
        tmpBufcoolantTemp, (int)xTaskGetTickCount());

    if (ret >= shadowBUFFER_LENGTH || ret < 0)
    {
        return -1;
    }
    else
    {
        return ret;
    }
}

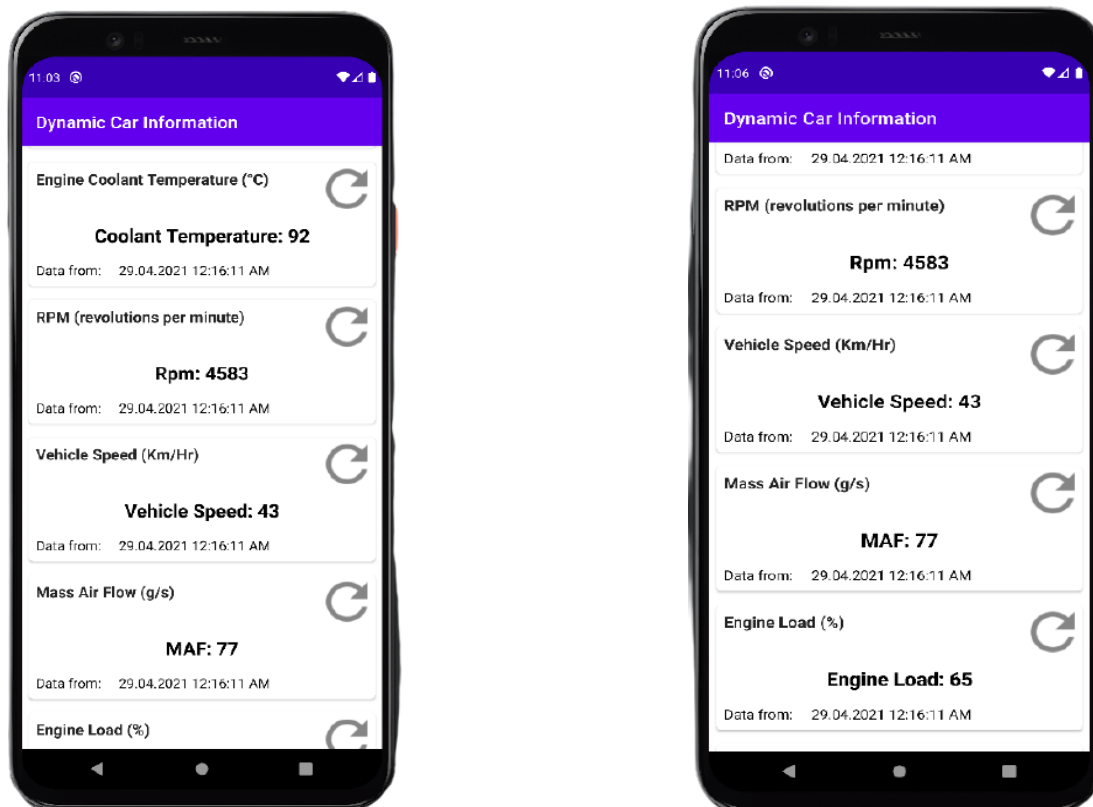
```

For future progression of this project regarding AWS IoT Core, I would like to store the data using AWS Dynamo DB and use the data stored to chart the data on the android application but due to time constraints I did not get this implemented but this would be great feature to implement in the future.

7.5 Android Application

I used an example project that is included with the NXP Freedom K64F and built/changed that example project to suit my needs.

For the user to view the data coming from the ECU I decided to use an android application that is connected to AWS to retrieve the data from. I used Android Studio as I was comfortable using android studio as I used it before during my internship in third year. The application itself is simple but displays the data when the refresh button is pressed. I have five of the ODB parameters displayed on the application. Pictured below is the android application running on the android emulator.



The android application uses the AWS credential security keys to allow it to connect to AWS and pass messages. The android application will subscribe for accepted updates from AWS Shadow.

```
bTempRefresh.setOnClickListener((v) -> {
    AwsShadow shadow = new AwsShadow();
    shadow.state.desired.tempUpdate = 1;

    // reported and metadata must not be sent along with desired shadow state
    shadow.state.reported = null;
    shadow.metadata = null;
```

When a refresh is pressed, using the coolant temperature refresh for example, an update of '1' is published to AWS which is passed onto the NXP Freedom K64F and it constantly checks for an update and if a '1' is received it will pass on the data of that parameter via the devices shadow.

```
if (shadow != null && shadow.state.reported != null) {
    // enable GUI items
    enableClickableGUIItems(true);

    // coolant temperature
    updateTempValuesAfterShadowUpdate(shadow);

    // remove timeout callback
    timeoutHandler.removeCallbacks(displayTimeoutToast);

    // last known shadow state received
    if (topic.equals(awsConstants.SHADOW_GET_ACCEPTED)) {
        runOnUiThread(() -> {
            Toast.makeText(
                getApplicationContext(),
                text: "Last known device shadow state has been received.",
                Toast.LENGTH_LONG
            ).show();

            // hide if no data has been received
            cardViewTemp.setVisibility(shadow.state.reported.temp == null ? View.GONE : View.VISIBLE);
        });
    }
}
```

When the state is reported from AWS the android application will parse the JSON and pull the data that is relevant and display it on the application. Again, each PID has its own function that is called when the state that is corresponding to that PID is reported.

The android application will subscribe to AWS successful updates and load its last known state and display it when you open the application. It also displays the timestamp for when that data was received.

```
// subscribe for all accepted changes
mqttConnection.subscribe(awsConstants.SHADOW_UPDATE_ACCEPTED, subscriptionCallback);

// get device shadow after successful connection
mqttConnection.subscribe(awsConstants.SHADOW_GET_ACCEPTED, subscriptionCallback);

// make publish
mqttConnection.publish(awsConstants.SHADOW_GET, AwsConstants.EMPTY_MESSAGE);

private void updateAccelValuesAfterShadowUpdate(final AwsShadow shadow) {
    if (shadow.state.reported != null){
        if (shadow.state.reported.temp != null) {
            final AwsShadow.State.Reported.Temp temp = shadow.state.reported.temp;

            runOnUiThread(() -> {
                cardViewTemp.setVisibility(View.VISIBLE);

                // update coolant temperature values
                tvTempValues.setText(String.format("Coolant Temperature: %d", temp.temp));

                // update timestamp
                tvTempTimestamp.setText(formatUnixTimeStamp(shadow.metadata.reported.temp.temp.timestamp));

                // hide progress bar
                progressBarTemp.setVisibility(View.INVISIBLE);
            });
        }
    }
}
```

When AWS reports the current state of the parameter it uses a function to display the data on to the screen. There is a function like this for each parameter and it basically checks the state to see if 'reported' is set. If so, it will check if the 'temp' parameter is the one that is reported if so, it updates the data that is on the application to the current data that has been reported.

For future progression of this project regarding the android application, I would like to have the data coming from AWS charted on the application for the user to view.

8 Ethics

Some ethical considerations that I considered in my project are as follows [10]:

Public –

Software engineers shall act consistently with the public interest.

Product –

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

Judgement –

Software engineers shall maintain integrity and independence in their professional judgement.

Profession –

Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

Self –

Software engineers shall participate in lifelong learning regarding the practice of their profession and promote an ethical approach to the practice of the profession.

In my opinion these are the ethical considerations that I applied to my project and that I thought were applicable to my project.

9 Conclusion

In my opinion, I am very pleased with the overall outcome of the project as I have learned a lot about CAN and how it works as I have always wanted to investigate this topic deeper as I come across it quite often. I finally understand how it all works and how it ties into the vehicles system. I also learned a lot about using AWS IoT Core as I had never used any of AWS's services before and how the IoT Core makes it easy to retrieve the data. Building the android application was very enjoyable as it is nice to see it progress through phases from the beginning to the end of the project.

The project also gave me a chance to apply my learning from modules that I was completing in college as some aspects of the project tied into these modules. FreeRTOS Queues that I implemented into my project was applied from the module, Embedded Real Time Operating Systems whilst my learning from Mobile App Development module helped me build on the android application. Cloud Computing module also help me to work with AWS as I did find this very challenging in the beginning.

During my time doing the project, it has taught me that there are certain aspects that I can improve on for the future, such as time management and organisation. I felt that I could have managed my time better throughout the project as I could have worked on multiple sections at the same time instead of just focusing on one. It has also thought me that I need to plan better before starting a project and make it clearer what the end goal for the and make sure that it is a realistic end goal.

10 References

- [1] ACEA, "Size and Distribution of the EU vehicle fleet" [Online]. Available:
<https://www.acea.be/statistics/tag/category/size-distribution-of-vehicle-fleet>
- [2] Components 101, "OBD-II Connector" [Online]. Available:
<https://components101.com/connectors/obd2>
- [3] Wikipedia, "CAN Bus" [Online]. Available:
https://en.wikipedia.org/wiki/CAN_bus
- [4] N. O'Keefe, "Free RTOS Queues" [Online], 2021. Available:
<https://learnonline.gmit.ie>.
- [5] Amazon Web Services, "AWS IoT Device Shadow service" [Online]. Available:
<https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html>
- [6] Amazon Web Services, "Understanding the AWS IoT Security Model" [Online]. Available:
<https://aws.amazon.com/blogs/iot/understanding-the-aws-iot-security-model/>
- [7] Wikipedia, "OBD II PIDs" [Online]. Available:
https://en.wikipedia.org/wiki/OBD-II_PIDs.
- [8] CSS Electronics, "Can Bus Explained" [Online]. Available:
<https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en>
- [9] Slide Share, "The Lifecycle of an AWS IoT Thing" [Online]. Available:
<https://www.slideshare.net/AmazonWebServices/the-lifecycle-of-an-aws-iot-thing>
- [10] UIO, "Software Engineering Ethics" [Online]. Available:
<https://www.uio.no/studier/emner/matnat/ifi/INF3700/v12/undervisningsmateriale/Software%20engeneering%20ethics.pdf>

[10] Car Auto Repair, "Automotive CAN Bus" [Online]. Available:

<https://www.car-auto-repair.com/automotive-can-bus-system-explained-instruction-diagnosis/>

[11] McuXpresso NXP, "API Documents" [Online]. Available:

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwj4j9rLhL_wAhXFTRUIHR4uAJsQFjAAegQIAxAD&url=https%3A%2F%2Fmcuxpresso.nxp.com%2Fapi_doc%2Fdev%2F116%2Findex.html&usg=AOvVaw0kklUnil1tR-jamhA20WGD

[12] NXP, "Kinetis Data Sheet" [Online]. Available:

<https://www.nxp.com/docs/en/data-sheet/K64P144M120SF5.pdf>

[13] Mikroe, "Mikroe AT6563" [Online]. Available:

<https://download.mikroe.com/documents/datasheets/ATA6563.pdf>

[14] Mikroe, "Mikroe WIFI 10 Click" [Online]. Available:

<https://download.mikroe.com/documents/datasheets/sx-ulpan-sb-2401.pdf>

[15] Ozen Elektronik, "Multiple Protocol OBD Simulator" [Online]. Available:

<https://www.ozenelektronik.com/download.php?id=5013>