Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

DriveWise

Project Engineering

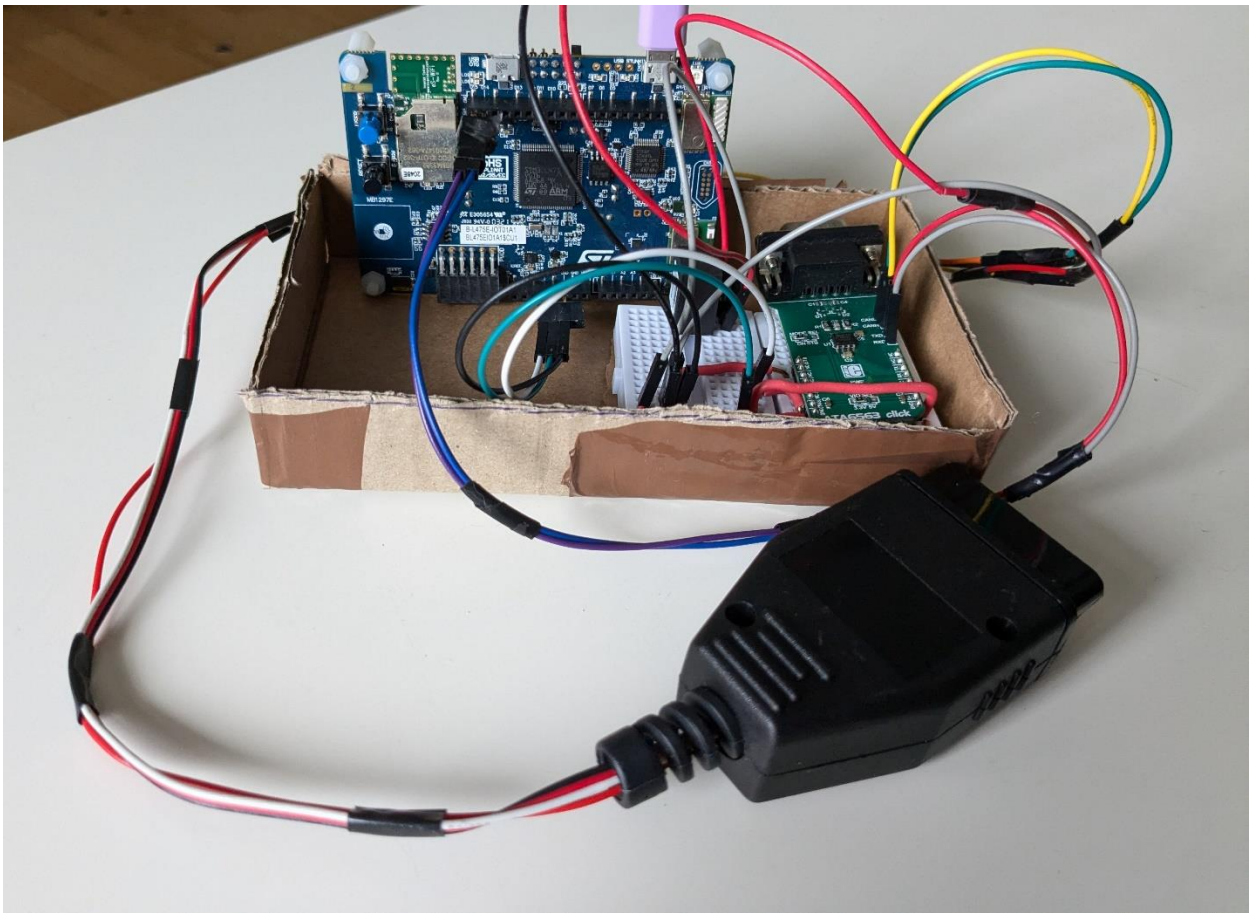
Year 4

James Albright

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025



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.

James Albright – G00379271

Acknowledgements

I would like to express my gratitude and appreciation to the academic staff at GMIT who have provided continuous support and guidance to me throughout the duration of the academic year

Specifically, my thanks go out to my project supervisor Niall O'Keeffe, who has been extremely helpful in providing technical feedback and direction to areas of my project of which required support. And of course, thanks to the course coordinator; Ben Kinsella, who had great knowledge and insight into a very unknown area to me.

Table of Contents

Table of Figures

1 Summary

DriveWise is a personal Blackbox with a mobile application interface. This monitors driver actions, safety and fuel efficiency by using data from a vehicle's OBD-II (Onboard Diagnostics) port. The STM32L475E will allow the user to do real-time data acquisition via CAN communication, an ATA6563 CAN transceiver, with the vehicle's ECU (Electronic Control Unit). Underneath, the STM will send a request frame, it will be converted by the transceiver so the ECU can understand it, in turn a response frame is then sent back to the transceiver and formatted for the STM. From here data processing can begin.

This data will fuel algorithms that analyses' driver behavior and calculate a safety and fuel efficiency score. The data used to provide these insights is also available to view, logs will be kept securely within an AWS S3 bucket for the user to access at any time.

The application will also provide live data, available to the user if they wish to view it. The speedometer, rev counter, engine temperature and more can potentially be displayed inaccurately by the onboard dials. Physical components wear down. As well as this, the car may not provide the user with all available information. Taking the data directly from the sensors is best as it removes a point of failure. In doing this project, I have met roadblocks regarding diagnostic information, which led me to believe that if you own a product that collects and processes its data, you should own it too.

2 Poster

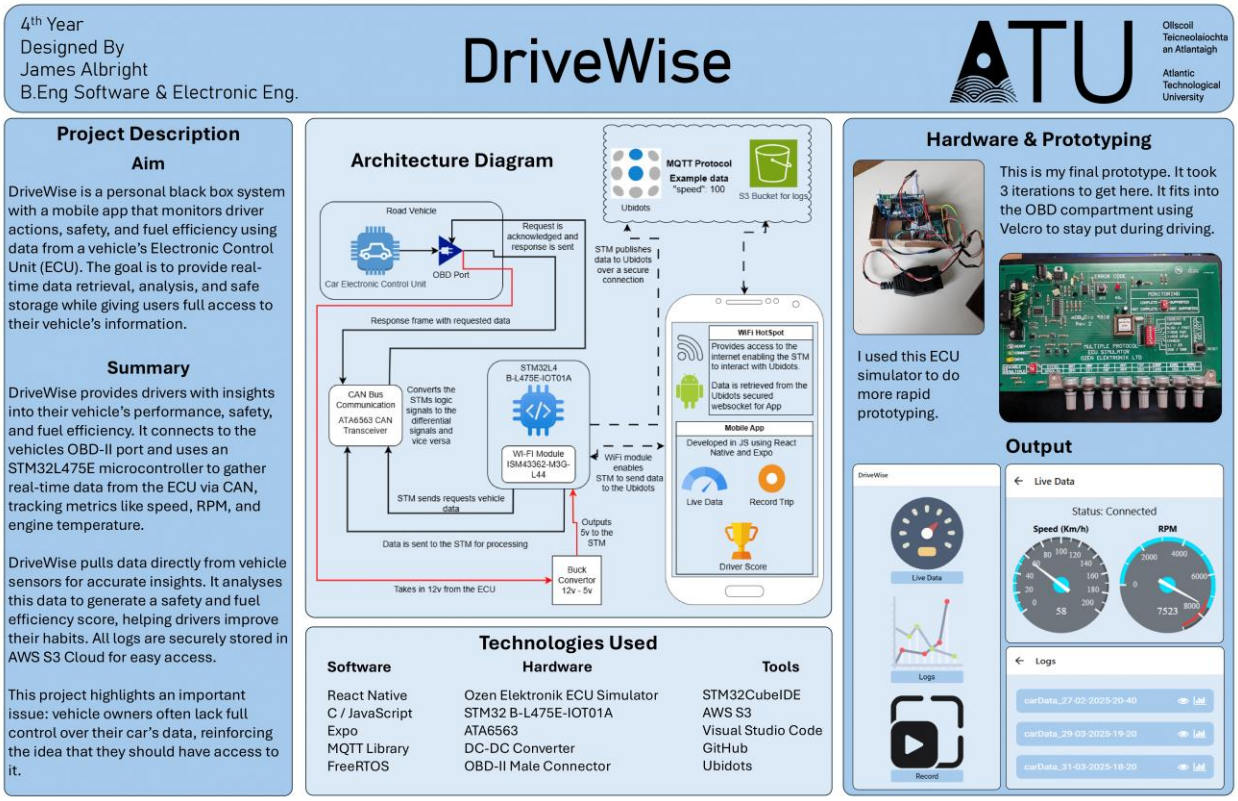


Figure 1 - Project Poster

3 Introduction

This report outlines the development of DriveWise, a personal Blackbox system designed to monitor driver behaviour, safety and fuel efficiency using data from a vehicle's OBD-II port. The project aims to help drivers with real-time insights and safety scores to improve driving habits, road safety and fuel economy.

The scope of this project includes designing a system that collects data from the vehicle's ECU via CAN, using the STM32L475E microcontroller and an ATA6563 CAN transceiver. It also incorporates algorithms for analysing driving behaviour and storing data securely on AWS S3 for easy access.

I have organised this report into sections that separately discusses the hardware and software components. The Hardware section focuses on the microcontroller, CAN communication and ECU integration. While the Software section covers the mobile app, data processing and cloud storage using AWS.

4 Research

4.1 CAN

"The Controller Area Network (CAN bus) is a message-based protocol" [1] which allows the Electronic Control Units (ECUs) within vehicles to communicate with each other. It uses a priority-based system to manage the order of messages, where each message (or frame) carries an identifier to determine this priority. When a message is sent, all devices on the network receive it and the recipient device decides whether to process or ignore the frame. CAN's communication protocol is defined and maintained by the ISO 11898 standard.

While there are alternatives to CAN, such as Ethernet [2], FlexRay [3] and LIN [4] (Local Interconnect Network), CAN remains the dominant protocol in automotive communication due to its established infrastructure and reliability. Ethernet is gaining traction for high-bandwidth applications like infotainment and ADAS, but it hasn't been as widely adopted in vehicle control systems due to its higher complexity, certification requirements and additional weight to the system (reduces driving distance for electric vehicles) [5]. FlexRay is used in specific high-performance applications but is costly and complex for broader use, while LIN is suited for simpler, non-critical tasks. When my car was designed, CAN was the most widely adopted protocol, offering the best balance of cost-effectiveness, reliability and availability. Due to my car only making use of the CAN protocol, it is why I have used it.

4.1.1 CAN Frame

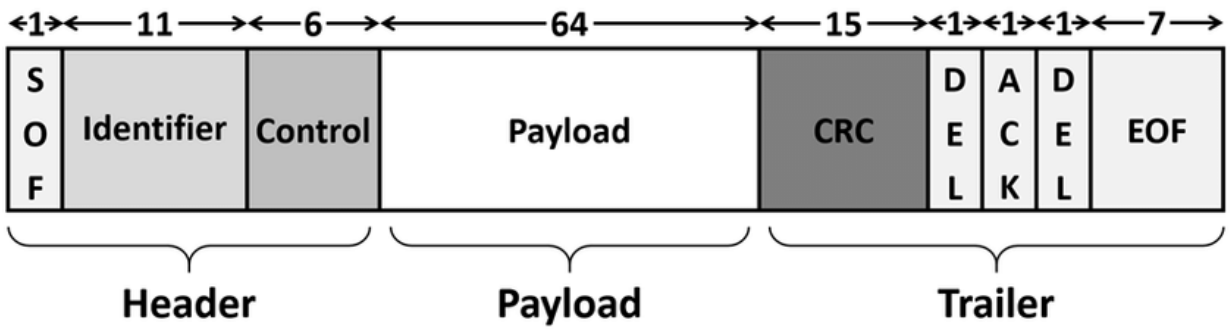


Figure 2 - CAN Frame Example

SOF (Start of Frame) - A single dominant bit (0) that marks the beginning of a CAN message and synchronises all nodes on the bus.

Identifier (IDE+RTR+ID) - The 11-bit (standard) or 29-bit (extended) message ID that determines message priority. Lower IDs have higher priority and the IDE bit distinguishes between standard and extended frames, while RTR indicates if this is a data frame (0) or remote request frame (1).

Control Field - Contains: 4 bits for DLC (Data Length Code) specifying data size (0-8 bytes) and

reserved bits (always dominant 0 in standard CAN)

Data Field - The actual payload (0-8 bytes) being transmitted.

CRC Field - A 15-bit checksum calculated from all previous fields (except CRC) plus a recessive CRC delimiter bit (1). Used to detect transmission errors.

ACK Field - Two bits where: ACK (acknowledgement) slot (transmitter sends recessive 1) and ACK delimiter (recessive 1) receivers overwrite the slot with dominant 0 to acknowledge successful receipt.

EOF (End of Frame) - 7 recessive bits (1) marking the end of the message. [6]

4.2 OBD-II

On-board diagnostics (OBD) refers to the vehicle electronics that provide vehicle self-diagnosis for the user and reporting capabilities for mechanics. An OBD gives you access to subsystem information for performance monitoring and analysing potential repairs. [7]

Within my project, I'll mention different types of data that come from the car like engine load, RPM, barometric pressure, throttle position and more. This data is pulled from Service (or Mode) 01, which provides real-time information related to the vehicle's powertrain and emissions systems. It includes standardised, publicly available parameters that give insight into how the engine is performing at any given moment.

While I only use Service 01 in my project, there are typically 10 in a system [8]. For the 'debugging' of a car you would be dealing with DTC's (diagnostic trouble codes) which I haven't used for the sake of a more focused project.

OBD is the standard protocol used mostly across cars (J1939 for heavy vehicles) to retrieve vehicle diagnostic information. Information is generated by ECUs within a vehicle. They are like the vehicle's brain or computers. Akin to CAN, this wasn't really a choice as my car has this from factory, like with most modern cars.

4.3 MQTT

MQTT (Message Queuing Telemetry Transport) is a messaging protocol made for simple and quick communication, especially in situations where network connections may be slow or potentially unreliable. It uses a publish and subscribe model, this is where devices send messages to a broker (Ubidots in my case) and other devices receive messages by subscribing to topics.

It is widely used in Internet of Things applications due to its small message size, low power demands and ease of use [9]. In this project, MQTT is used to send car data to a cloud service, which then forwards it to a web application. Its reliability and simplicity made it a suitable choice for handling vehicle data in real time. As well as this, I had experience in using it throughout previous modules making it a protocol I was familiar with and being ideal for this use case.

Other protocols like HTTP [10] (Hypertext Transfer Protocol) and CoAP [11] (Constrained Application Protocol) could have been used to transmit the data, but they tend to be less efficient for real-time applications, especially when working with constrained devices like microcontrollers. HTTP is more heavyweight and better suited for request-response communication, while CoAP is designed for constrained networks but is less widely supported. AMQP [12] (Advanced Message Queuing Protocol) is a strong contender but more resources intensive and not ideal for use in microcontrollers. MQTT offered the best balance of simplicity, performance and support for this project.

4.4 WebSockets

WebSockets is a communication protocol that provides full-duplex, persistent connections between a client and a server over a single TCP connection [13]. Unlike HTTP, which requires a new connection for every request, this keeps the connection open, allowing data to be sent and received in real time without constant handshakes or polling. This makes it ideal for my use case, where I have new values coming in every second.

One of the main advantages of WebSockets over HTTP is its low-latency, bidirectional nature, which improves performance for real-time systems [14]. However, it is slightly more complex to implement and may not be supported in more restricted environments. Here I use WebSockets alongside MQTT to provide a live data stream to the mobile app I developed.

4.5 ECU Simulators

An ECU simulator is a tool used to mimic the behaviour of a real vehicle's ECU without needing access to a car. It is almost necessary during development and testing, where live data and responses are needed but working with an actual vehicle is too impractical. These simulators can send and receive CAN messages, simulate sensor values and respond to diagnostic requests in a controlled environment.

In my case, I used the Multiple Protocol OBD ECU Simulator with a mOByDic4910 chip from Ozen Elektronik, which was available at the college. While functional, it required a lot of manual setup, especially since the microcontroller I was using had no CAN transceiver. Given the choice, I would have preferred the Freematics OBD-II Emulator MK2, as I am more interested in the software side of things and it offers the ability to emulate ECU responses using scripts. That approach would have freed me up to focus more on developing and refining the software side of my project. That said, working with the Ozen simulator pushed me to develop a much deeper understanding of how CAN communication works, including how diagnostic requests are structured, how responses are handled and how different PIDs interact with the bus. Despite the extra effort involved, it gave me a more complete picture of the system as a whole.

5 Project Architecture

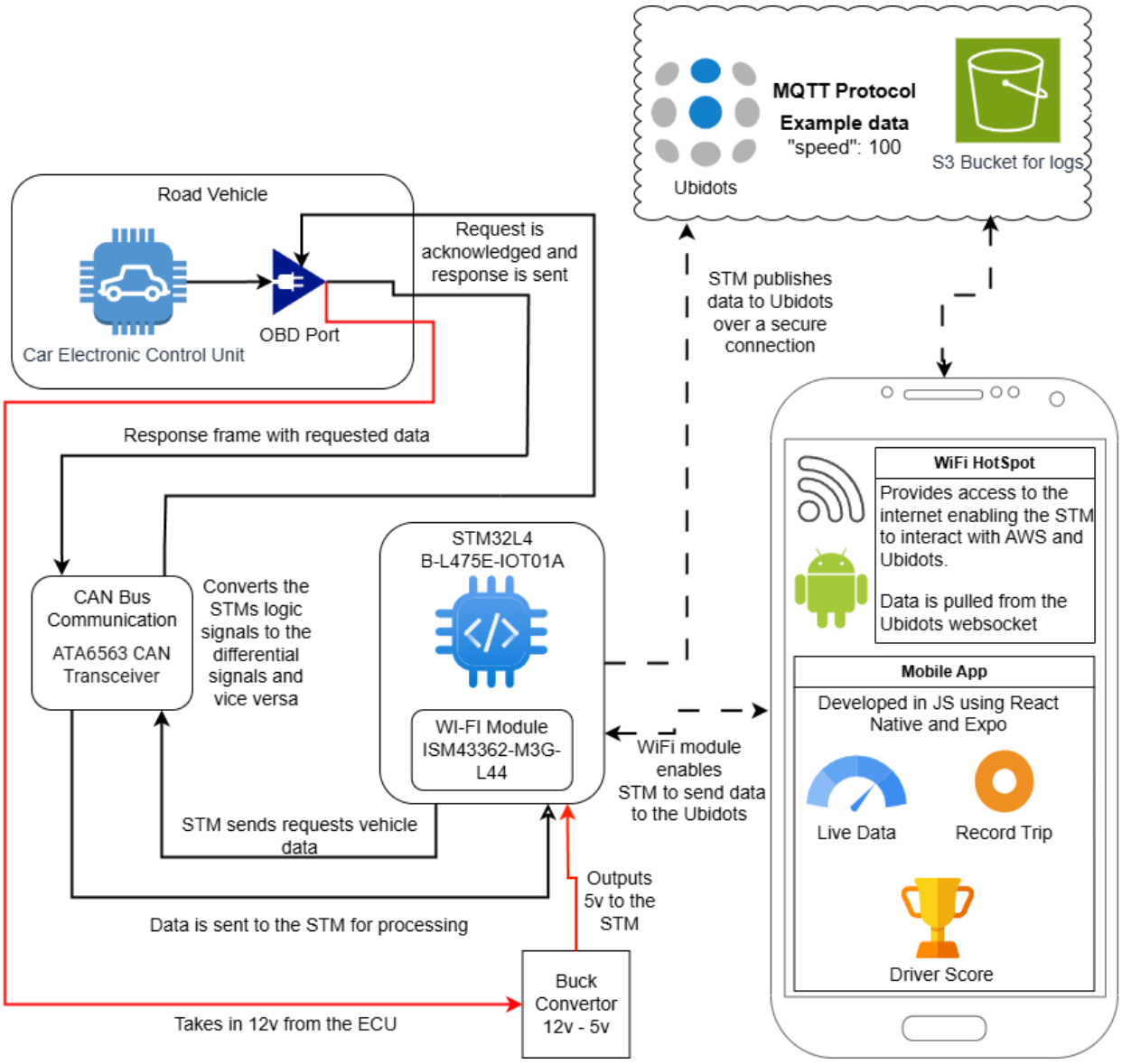


Figure 3 - Architecture Diagram

Brief overview of this diagram; the STM produces a request for data by adding it to the transmit mailbox, it is then parsed inside the CAN transceiver and converted to differential CAN signals. This is so the cars ECU is able to understand it. The message is decoded, acknowledged and a return frame is made to be sent back. The process happens backwards now, the transceiver converts it to standard logic for the STM. It then queues the data up to be sent to Ubidots, the mobile app reads this in and displays it (and saves it to a csv file, if the user is recording data). After a 'trip' it is now ready to be analysed and scored for efficiency and clean driving.

6 Project Schematic

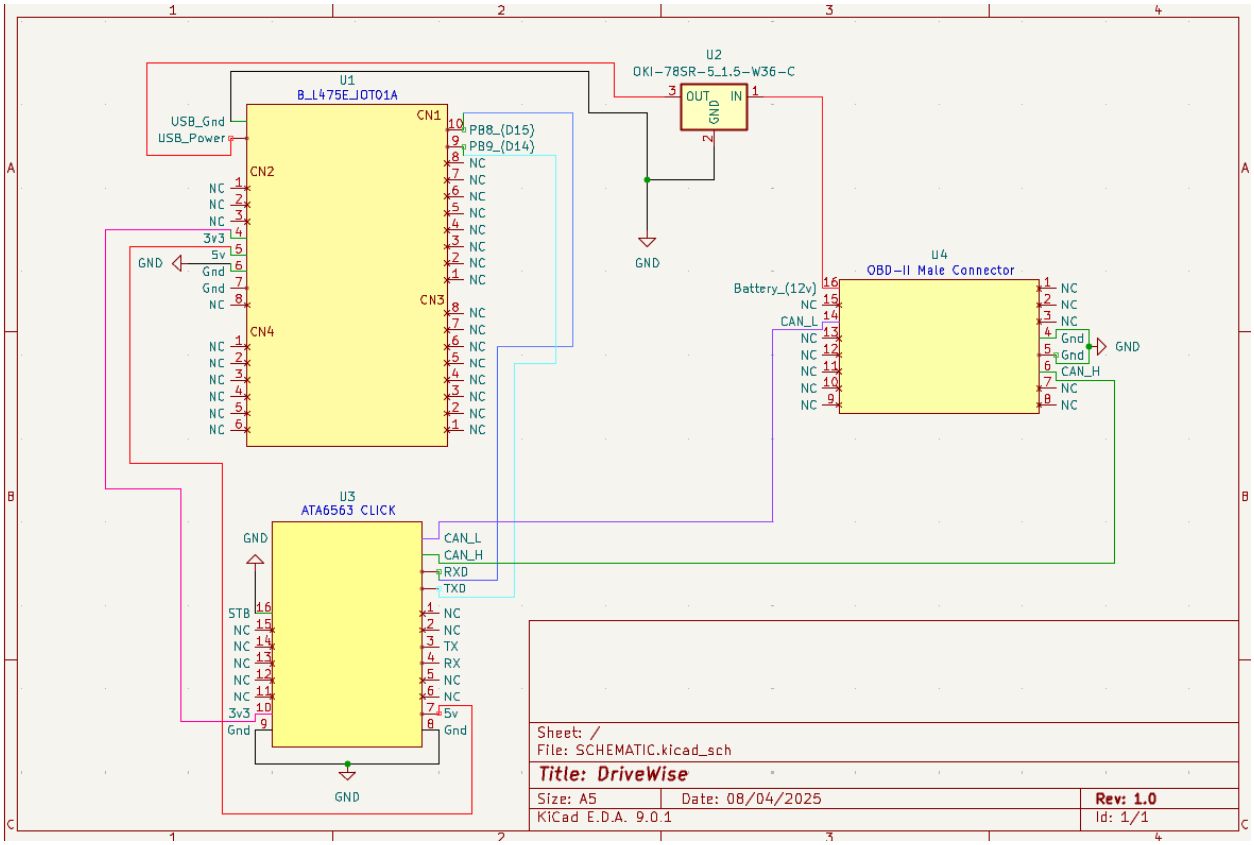


Figure 4 - Project Schematic

Brief overview of this diagram; There are four components here, the microcontroller (U1), the CAN transceiver (U3), the OBD connector (U4) and a DC-DC convertor (U2).

7 Hardware

7.1 STM32 B-L475E-IOT01A

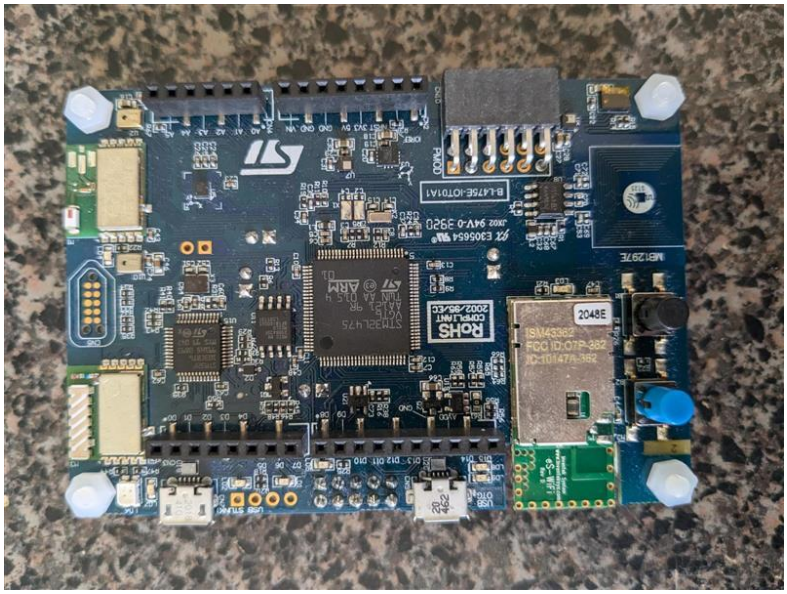


Figure 5 - STM32 L475E IOT01A

The STM32 B-L475E-IOT01A is a development board based on the STM32L475VG microcontroller, built around an Arm Cortex-M4 core. In this project, it serves as the main controller for sending/receiving CAN messages and transmitting data via Wi-Fi. I mainly used this board due to prior experience with it across different modules, which made development efficient.

The key feature used in this project is the onboard CAN controller, which is responsible for generating and managing messages sent to the CAN transceiver. The board also provides power to the external CAN transceiver. In addition, the integrated Wi-Fi module was used to send data to the cloud, removing the need for any additional communication hardware.

Regarding its specifications, the STM32L475 runs at 80 MHz and includes 1MB of Flash memory and 128KB of SRAM, which is more than sufficient for running the tasks required here.

Its compatibility with FreeRTOS (a real-time operating system) makes task scheduling fairly easy to get setup. The board is also equipped with multiple connectivity options (BLE/NFC/Wi-Fi), though only Wi-Fi (ISM43362-M3G-L44) was used here in order to communicate with Ubidots.

7.2 CAN Transceiver

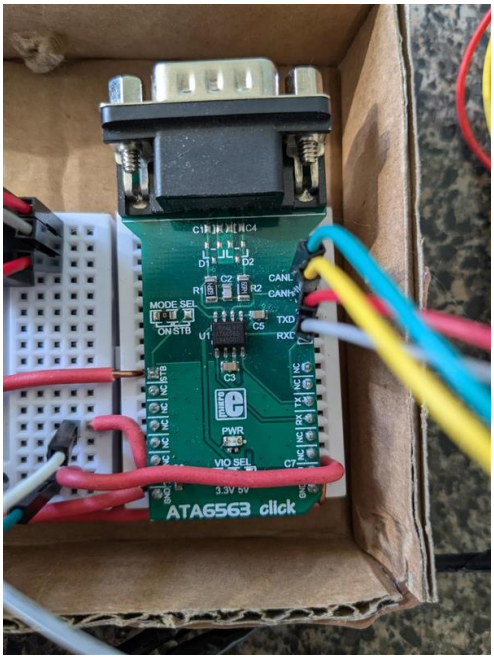


Figure 6 - ATA 6563 CAN Transceiver

This is an ATA6563 click CAN transceiver. “The ATA6563 Click is a compact add-on board that provides an interface between a controller area network (CAN) protocol and the physical two-wire CAN bus” [15]. It is running on the 5v outputted by the STM board I am using (it can use 3.3V as well by moving the onboard switch). It takes data signals from the microcontroller and makes differential signals for it to communicate over the CAN bus, allowing messages to be transmitted and received across the vehicle’s network without interference or signal degradation [16].

7.2.1 Example

Before reading this table, you need to know that a 0 is a **dominant** bit and a 1 is a **recessive** bit. These are taken as 0v (0) and 5v (1) from our microcontroller to the CAN transceiver.

If you want more information about the CAN frame, read section 4.1.1.

I’ll take a basic CAN message here and show how it goes from data to differential signals.

Data

Data Field	Description	Bits / Hex
SOF	Start of frame	0 / 0x0
ID	11 bit identifier	00100010001/0x111
RTR	Data frame	0 / 0x0
IDE	Standard CAN (11 bit ID)	0 / 0x0
DLC	Data length code	0010 / 0x2
Data	The payload	11011110 10101101 / 0xDE 0xAD
CRC	Error checking bits	1101010110010101 / 0x6ACA
ACK	ECU confirms by converting to 0	1 / 0x1
EOF	End of frame	1111111 / 0x7F

Differential

For every 0 and 1 sent from the microcontroller, these are seen as 0V and 5V on the TXD line. The CAN transceiver then converts these logic levels into differential voltages. When it receives a logic 0 (0V), this is treated as a dominant bit where the transceiver drives CANH up to around 3.5V and CANL down to around 1.5V, creating around a 2V difference. When it receives a logic 1 (5V), it's seen as a recessive bit where both CANH and CANL are set to about 2.5V, meaning there's no difference between them. A dominant bit (logical 0) is actually represented by a voltage difference (CANH > CANL) and a recessive bit (logical 1) by equal voltages. This follows the ISO 11898-2 standard.

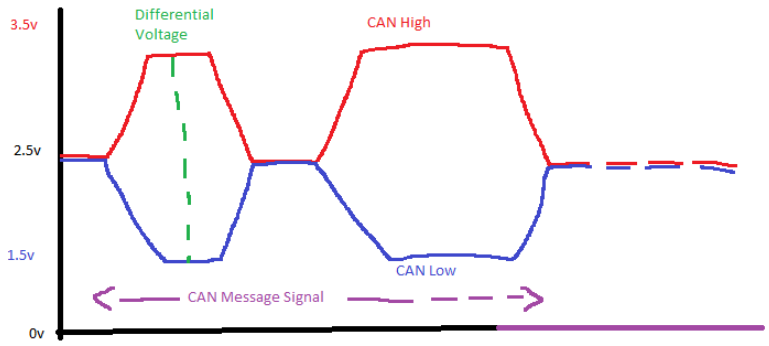


Figure 7 - CAN Bus Differential

7.3 ECU Simulator

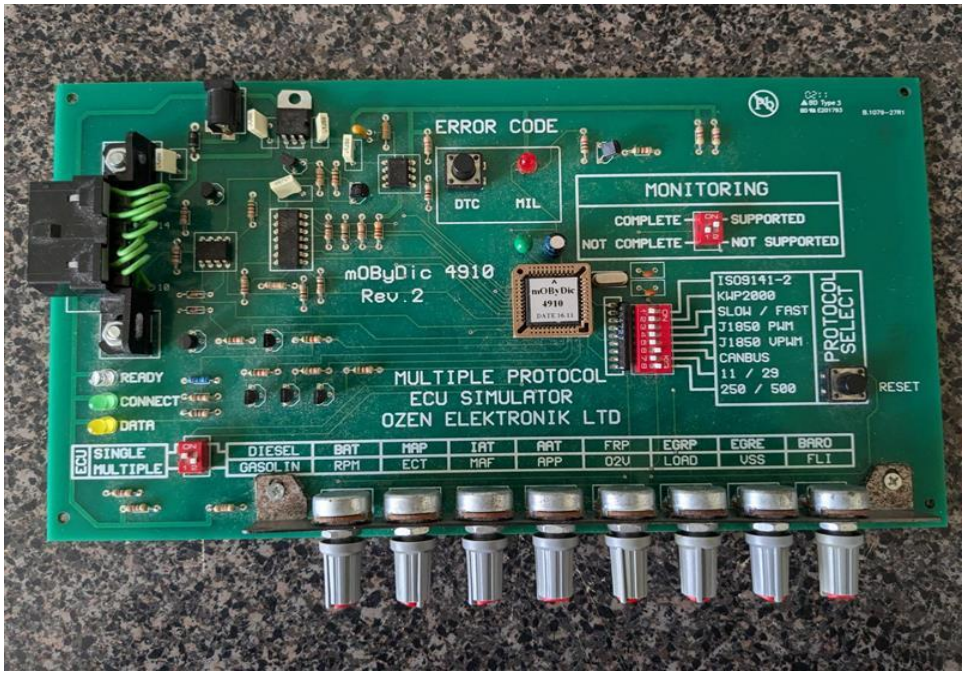


Figure 8 - OZEN Multiprotocol ECU Simulator

The OZEN Multiprotocol ECU Simulator is a development tool designed to replicate the behaviour of a real vehicle's ECUs. It allows users to send/receive diagnostic messages, control/send data using the 8 variable resistors and select either petrol or diesel PIDs (parameter identifiers) over standard vehicle communication protocols such as ISO9141-2, KWP2000, J1850 PWM, J1850 VPWM and most importantly, CAN [17]. All of this meant I did not need access to a physical car during my development stage which was gave me massive time savings.

One of the main benefits of using a simulator like this is that it removes the risks and limitations of working directly with a live vehicle. It gives a safe, controlled environment to test how data is sent, how responses are handled and how the system reacts to different requests. For development, this makes debugging much easier and allows for repeated testing without worrying about interfering with a real car's systems. It also provides quick access to the types of responses you'd typically only get while the car is running.

The simulator can emulate specific PIDs and DTC responses, making it ideal for fine-tuning request handling and understanding how the communication flow works, this is especially useful during the early stages of building a system or when you're testing against different situations.

7.4 OBD

Figure 9 - OBD-II Male Connector

OBD-II is the standard interface for ECU's within cars and light duty vehicles. This was how I connected to the OZEN ECU Simulator or to my own car, without this I had no way of communicating to either. The purpose of this is to send signals through the CAN High/Low pins to the ECU. I also took out a ground and 12v in order to power the board (which I cover in detail in the next section). The benefit of OBD-II being a standard is that the layout and signals are the same across most vehicles, making it ideal for diagnostic tools and college projects alike. It provides a universal access point for reading and sending vehicle data, which is exactly what I needed for this setup. Below is the inside of the male connector, I have 4 pins used:

16: 12v – power for microcontroller

14: CAN Low

6: CAN High

5: Signal Ground

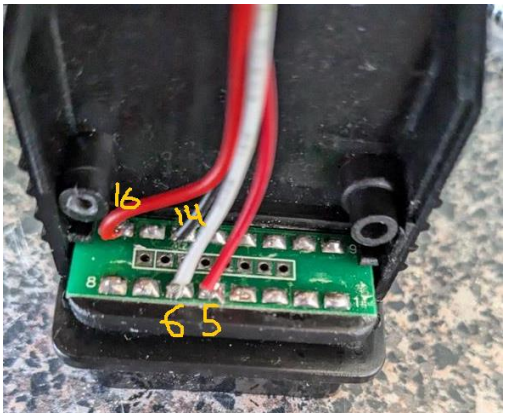


Figure 10 - OBD-II Male Connector Inside

7.5 Power

Figure 11 - DC-DC Convertor

To power the hardware in this project, I needed a source and a way to regulate it. I initially thought to use the cars battery (not a great idea), using some long cable from the engine to the

driver's cabin. Thankfully, while working on the setup of everything, I noticed there was a 12v output from the OBD-II connector which was great. I had a source, now I just needed to drop it down. That's where the DC-DC convertor comes in (pictured above). It takes in a range of 7v – 36V [18], which is perfect as we get 12v from the car and outputs 5v for the microcontroller to operate.

8 Software

8.1 FreeRTOS

"FreeRTOS stands for Free Real-Time Operating System. It is an open-source operating system targeted at embedded applications that need real-time event processing" [19]. I used it in this project to manage how different parts of the program run without clashing or blocking each other. Instead of everything running in one long loop, FreeRTOS lets us split the program into tasks that run separately, almost like different mini programs that can be paused and resumed. This helps keep everything responsive and organised, especially when dealing with things like CAN messages and MQTT communication that will happen at different times.

Here it is used to create and manage a few key tasks. There's an initialisation task that sets everything up and then deletes itself to save stack space. After that, the CAN_Task is responsible for sending PID requests over the CAN bus and pacing them out properly. The MQTT_Task takes care of reading messages from a queue and publishing them to the MQTT broker. These tasks run alongside each other and don't interfere because of how FreeRTOS schedules them.

To help these tasks work together, FreeRTOS has queues and semaphores. The queue is used to pass messages between the CAN interrupt (where data is received) and the MQTT task (which sends that data online). It acts like a buffer that keeps everything in order. The semaphore is used like a signal to say when to go. In this case, a timer acts as the signal to the CAN task to start sending its next bunch of requests. Without FreeRTOS handling all of this, the system would be messy and hard to time properly. I made great use of the FreeRTOS API reference manual provided by Niall O'Keefe during our Embedded RTOS module.

8.2 CAN

8.2.1 CAN Messages

```
void CAN_Tx_Request(uint8_t requestedPID) {
    CAN_TxHeaderTypeDef TxHeader;
    uint8_t TxData[8] = {0x02, 0x01, requestedPID, 0x55, 0x55, 0x55, 0x55, 0x55};
    uint32_t TxMailbox;

    //Configure CAN header (Data length, Broadcast Identifier, Standard 11bit, Carries Data)
    TxHeader.DLC = 8;
    TxHeader.StdId = 0x7DF;
    TxHeader.IDE = CAN_ID_STD;
    TxHeader.RTR = CAN_RTR_DATA;

    HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox);

}
```

Figure 12 - CAN Message Request

To transmit or request information over the Controller Area Network (CAN), a message must first be constructed. Each CAN frame consists of an identifier, control bits, a data length code (DLC) and up to 8 bytes of data. In my project, I used standard 11-bit IDs. To request data from the vehicle, you need OBD Parameter IDs (PIDs), which follow a set format defined by SAE J1979 [20]. These allow access to engine-related data like RPM, speed and engine temperature [21].

CAN messages are sent using the 'Mailbox system', which holds up to three pending messages [22]. I structured my main FreeRTOS task to cycle through each PID request with a delay, calling HAL_CAN_AddTxMessage() to place each message into a mailbox. Once the message is sent, the microcontroller waits for a response via interrupt.

```
//CAN Interrupt Handler
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
    if (HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader, RxData) == HAL_OK) {
        switch (RxData[2]) {
         case RPM:
            //Shift first byte to the left by 8bits, add the other byte then scale down
            uint16_t vehicleRPM = ((RxData[3] * 256 + RxData[4])/4);
            printf("Vehicle RPM: %d rpm\r\n", vehicleRPM);
            sprintf(rxPayload, "{\"rpm\":%d}", vehicleRPM);
            if (xQueueSendFromISR(mqttQueueHandle, rxPayload, NULL) != pdPASS) {
                printf("Queue send failed\r\n");
            }
            break;
```

Figure 13 - CAN Interrupt

Incoming messages are handled by the CAN RX interrupt callback function, HAL_CAN_RxFifo0MsgPendingCallback(). Within this function, I call HAL_CAN_GetRxMessage() to retrieve the frame and then extract/parse the data depending on the PID. Each parsed value

is sent to a FreeRTOS queue which is later accessed by the MQTT task for publishing. For some data (like above) there needs to be some adjustments, rpm comes in two bytes so it needs to be shifted left by 8bits to get a 16bit value, add the second byte and then scale down. I used this site for a variation of formulas to get the correct data from the ECU.

8.2.2 Parameter IDs

```
//Enums for CAN PIDs

#ifndef CAN_PIDS_H
#define CAN_PIDS_H

typedef enum {
    RPM = 0x0C,//RPM
    ENGINE_COOLANT_TEMP = 0x05,//ECT
    MASS_AIR_FLOW = 0x10,//MAF
    ENGINE_LOAD = 0x04,//LOAD
    VEHICLE_SPEED = 0x0D,//VSS
    FUEL_LEVEL = 0x2F,//FLI
    //Diesel Only
    AMBIENT_TEMP = 0x46,//AAT
    MANIFOLD_PRESSURE = 0x0B,//MAP
    BARO_PRESSURE = 0x33//BARO
} OBD2_PID;


#endif // CAN_PIDS_H
```

Figure 14 - CAN PID Enum

The PIDs I selected were the same as what I had access to on the ECU Simulator I used. These were gotten from the csselectronics website, which had a handy table for specific requests, as well aslisting all available PIDs. I then verified them using my ECU simulator to make sure the requests matched. Each request message is sent to the ID 0x7DF, which is an address recognised by all ECUs that support OBD-II. The format of the data field follows the convention: [number of additional bytes], [mode], [PID], followed by padding if needed. For example, a request for RPM takes the form ‘0x02 0x01 0x0C 0x00 0x00 0x00 0x00 0x00’ (refer to figure 12 to see its construction in code).

8.2.3 IOC Config

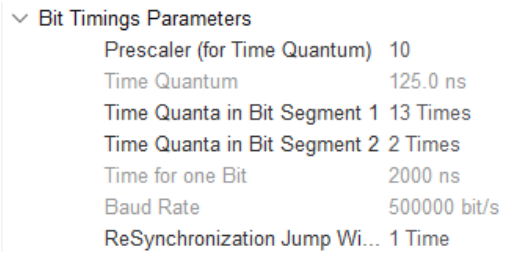


Figure 15 - CAN Config In IOC

CAN was configured using STM32CubeMX, where I enabled CAN1 in normal mode and routed the transmit and receive lines through PB9 (TX) and PB8 (RX). These connect directly to the CAN transceiver, which manages the physical layer. I set the CAN filter to accept all messages and enabled interrupts for received frames. Bit timing was set based on this website, as several options were available including 125kbps, 250kbps, 500kbps and 1Mbps. Each had their own prescaler and segment values. I selected 500kbps with a prescaler of 10, a time segment 1 of 13 and a time segment 2 of 2. This offered a good balance of speed and reliability for my setup. STM32CubeMX then generated the required init code with these timing settings, allowing the CAN peripheral to communicate with a vehicles ECU/the simulator I used.

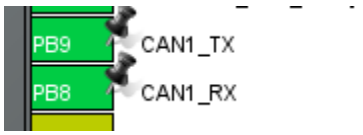


Figure 16 - Pin Config For CAN

8.3 Data Processing

The general idea of this is that once we get the data from the ECU, we want to package it up for Ubidots. It is passed to the queue like “var”:0 as we are passing up JSON. This is ideal on the frontend as handling it is super easy this way.

```
sprintf(rxPayload, "{\"rpm\":%d}", vehicleRPM);
if (xQueueSendFromISR(mqttQueueHandle, rxPayload, NULL) != pdPASS) {
    printf("Queue send failed\r\n");
}
```

Figure 17 - Send to queue function

Once the data is queued up, it is stored in a buffer (publishMsg). The MQTT_Task runs in an infinite loop and waits for data to arrive on the queue. When it receives a message, it fills a MQTTMessage struct with the data, sets the length and QOS (quality of service) level, then attempts to publish it to the MQTT broker using the /v1.6/devices/drivewise topic. If the publish is successful, a confirmation message is printed to the terminal — otherwise, an error is logged.

```
void MQTT_Task(void *pvParameters) {
    char publishMsg[50];
    while (1) {
        if (xQueueReceive(mqttQueueHandle, publishMsg, portMAX_DELAY) == pdPASS) {
            MQTTMessage mqmsg;
            memset(&mqmsg, 0, sizeof(MQTTMessage));
            mqmsg.payload = publishMsg;
            mqmsg.payloadlen = strlen(publishMsg);
            mqmsg.qos = QOS0;

            //original endpoint for ubidots /v1.6/devices/drivewise
            if (MQTTPublish(&client, "/v1.6/devices/drivewise", &mqmsg) != SUCCESS) {
                printf("MQTTPublish failed\r\n");
            } else {
                printf("Message published: %s\r\n\n", publishMsg);
            }
        }
    }
}
```

Figure 18 - MQTT Task

8.3.1 Main Task

```
void CAN_Task(void *pvParameters) {
    uint8_t rotating_index = 0;
    uint16_t demo_count = 0;

    while (1) { //try port max delay, only run when semaphore given
        //this is done to stop ubidots from limiting my requests
        const uint8_t ROTATING_PIDS[] = {
            ENGINE_COOLANT_TEMP,
            MASS_AIR_FLOW,
            FUEL_LEVEL,
            AMBIENT_TEMP,
            MANIFOLD_PRESSURE,
            BARO_PRESSURE
        };
        const uint8_t NUM_ROTATING = sizeof(ROTATING_PIDS)/sizeof(ROTATING_PIDS[0]);

        if (xSemaphoreTake(publishSemaphore, portMAX_DELAY) == pdTRUE) {
            printf("Speed Requested...\r\n\n");
            CAN_Tx_Request(VEHICLE_SPEED);
            vTaskDelay(pdMS_TO_TICKS(50));
```

Figure 19 - Main Task

In this task, I use an array to store 6 PIDs which 'rotates' by incrementing the index (this is done to avoid being limited by Ubidots for too many publishes) and request data from an ECU every second.

`**const** uint8_t NUM_ROTATING = **sizeof**(ROTATING_PIDS)/**sizeof**(ROTATING_PIDS[0])`

This is just a way to get the number of elements in the array, which is necessary here to make sure it cycles through the array over and over, without going out of bounds

`rotating_index = (rotating_index + 1) % NUM_ROTATING;`

```
// Send one rotating parameter
printf("Requested PID: 0x%02X...\r\n\n", ROTATING_PIDS[rotating_index]);
CAN_Tx_Request(ROTATING_PIDS[rotating_index]);
vTaskDelay(pdMS_TO_TICKS(50));
```

Figure 20 - Separate call for the array of PIDs

8.3.2 Timer Config for Semaphore

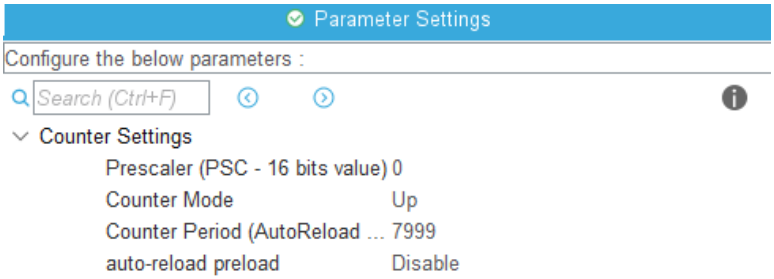


Figure 21 - Timer Config

The semaphore is given from the interrupt handler running off of TIM6 (configured above) which occurs every second. This is used for the main task (CAN_Task) where we want to request data from the ECU.

```
void TIM6_Handler() {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xSemaphoreGiveFromISR(publishSemaphore, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Figure 22 – Code for how semaphore is given

8.4 MQTT Broker

For my project, I needed an MQTT broker. My choice was Ubidots as I have had a lot of previous experience with it, it does exactly what I need and the setup is fairly simple.

8.4.1 Configuration

```
#ifndef INC_CLOUDBROKERCREDENTIALS_H_
#define INC_CLOUDBROKERCREDENTIALS_H_

#define CloudBroker_HostName "industrial.api.ubidots.com"

#define CloudBroker_Port "8883"

#define CloudBroker_Username "token_here"

#define CloudBroker_Password ""

#define CloudBroker_ClientID "unqiue_id_here"
```

Figure 23 - Cloud Broker Credential Setup

There isn't a whole lot needed to get setup for Ubidots, you have to have your host name, port, token (username) and a unique id (clientId). The host/port are from the Ubidots docs, and the token/id are for a specific device – which you get after creating one.

8.4.2 Connecting To The Broker

This section is trickier, I was fortunate enough to have had multiple modules based around IoT (internet of things) using the STM microcontroller that's used in this project. So a lot of the code here was provided by Niall O'Keefe. Firstly, bring in your cloud broker credentials

```
//Initialise MQTT broker structure
//Fill in this section with MQTT broker credentials from header file
MQTT_Config.HostName = CloudBroker_HostName;
MQTT_Config.HostPort = CloudBroker_Port;
MQTT_Config.ConnSecurity = "2";
MQTT_Config.MQUserName = CloudBroker_Username;
MQTT_Config.MQUserPwd = CloudBroker_Password;
MQTT_Config.MQClientId = CloudBroker_ClientID;
```

Figure 24 - Broker Variables

You'll then need to initialise the WiFi network, get the IP address, MAC address, set the RTC (this is needed to validate the TLS cert) and then create the network socket.

```
if (net_get_ip_address(hnet, &ipAddr) != NET_OK) {
    printf("\n\rError 2");
}
else
{
    switch(ipAddr.ipv) {
        case NET_IP_V4:
            printf("\n\rIP address: %d.%d.%d.%d\n\r", ipAddr.ip[12], ipAddr.ip[13], ipAddr.ip[14], ipAddr.ip[15]);
            break;
        case NET_IP_V6:
        default:
            printf("\n\rError 3");
    }
}

if (net_get_mac_address(hnet, &macAddr) == NET_OK) {
    printf("\n\rMac Address: %02x:%02x:%02x:%02x:%02x:%02x\r\n",
            macAddr.mac[0], macAddr.mac[1], macAddr.mac[2], macAddr.mac[3], macAddr.mac[4], macAddr.mac[5]);
}

if (setRTCTimeDateFromNetwork(true) != TD_OK) {
    printf("Fail setting time\r\n");
}
```

Figure 25 - Connect to broker code

There are a bunch of options you can set but for basic connectivity it is just create the socket, pass in a specified protocol and open the socket for connection.

```
//Create network socket with TLS
ret = net_sock_create(hnet, &socket, NET_PROTO_TLS);
if (ret != NET_OK) {
    printf("\n\rCould not create the socket.\r\n");
    printf("Check MQTT broker configuration settings.\r\n");
    while(1);
} else {
    ret |= net_sock_setopt(socket, "tls_server_name", (void *)MQTT_Config.HostName, strlen(MQTT_Config.HostName) + 1);
    ret |= net_sock_setopt(socket, "tls_ca_certs", (void *)ca_cert, strlen(ca_cert) + 1);
    ret |= net_sock_setopt(socket, "sock_noblocking", NULL, 0);
}
printf("cert length: %d\n\r", strlen(ca_cert));
ret = net_sock_open(socket, MQTT_Config.HostName, 8883, 0);
```

Figure 26 - Create network socket

8.5 Mobile App

My mobile app is developed in JavaScript using the React Native framework and runs through Expo Go for easy testing and deployment.

8.5.1 What It Does

The DriveWise mobile application is a user interface for this project. It is able to display vehicle data to the end user, record a trip by logging incoming values and give the user the choice of viewing their trips as a whole through a CSV file.

8.5.2 Displaying Data

This was possibly the most difficult aspect of the app, UI/UX are not something I have had to consider in either work (as I mostly do backend) or in college until this year. With help from the Mobile App Development module I finished, I believe that I made something accessible, user friendly and useful.

To show data I had to retrieve it first, so I set up a WebSocket to connect to Ubidots using the ‘mqtt’ library. Once configured it was simple to connect, get messages and close the connection.

```
mqttClient.on('message', (topic, message) => {
  try {
    //strip back the string in topic to get the actual topic name we are subscribed to
    const mqttData = JSON.parse(message.toString());
    const topicSplit = topic.split('/');
    const topicName = topicSplit[topicSplit.length-1];

    switch (topicName) {
      //check the case
      //setData to new value if its changed, if not leave as the same
      //set the ref for each topic value, increment count to keep track of data received
      case 'speed':
        setData((prevData) => ({ ...prevData, speed: mqttData.value }));
        logData.current.speed = mqttData.value;
        ++cnt.current.s;
        break;
```

Figure 27- -On Message Function

Here I get the topic name, use a switch case to set the data for display as well as assign it to a React reference to pass into a function to write to a csv. If I did not use ‘useRef’, I would lose values between component renders with variables or need unnecessary re-renders with state variables.

With the use of the ‘react-native-cool-speedometer’ library, I was able to make simple but good looking speedometers for information like speed, rpm, engine load, fuel level and even engine temperature. The rest are displayed in a card format, as seen below.

Figure 28 - Displayed Data

8.5.3 Recording Data

To ‘record’ the data I had to do a few things. Firstly I added a button to start/stop the process, once started it will initialise a CSV file with a header, write to the file with data captured in the display code and when the stop button is pressed, it sends the file to my S3 bucket.

```
//do we have our first run?
if (isFirstIteration.current) {
  if (Object.values(cnt.current).every(value => value > 0)) { //has each count var incremented once?
    console.log("All data received");
    isFirstIteration.current = false;
  }
} else { //have we incremented the main 3 topics once? if so, push data to csv and reset
  if (cnt.current.s > 0 && cnt.current.r > 0 && cnt.current.eL > 0) {
    console.log("Data ready for CSV:", logData);
    writeCSV(logData.current, speedLim.current);
    cnt.current.s = 0;
    cnt.current.r = 0;
    cnt.current.eL = 0;
  }
}
```

Figure 29 - Write To CSV Call

Above we see that I'm keeping track of what data has come in and what hasn't. I really only care about RPM, speed and engine load (s, r, eL) as they determine the driver score, as well as being the only data that's guaranteed to be pushed every second.

For this to all work though, I use Context API instead of prop drilling (passing data through nested components). Instead, it's a way to share data across the whole app without having to pass it through each screen or component manually. It acts like a central place where any part of the app can get the data it needs. So I can get the state (are we recording or now) and pass that into my 'fileHandler.js' file to determine if we should be writing to the CSV file.

8.5.4 Accessing Logs

In the previous section you see how data is pushed up to the S3 bucket. But now we want to look at it. I considered storing files locally, but I thought it would be good to get some experience with AWS and save the user from having to have them all on the phone. If the app was deleted, the data wouldn't persist either. How we get it is covered in the 'AWS' under 'Functions'. To display it, I use a modal to save myself from creating a new screen. It acts as an overlay or pop-up screen, as you can see below.

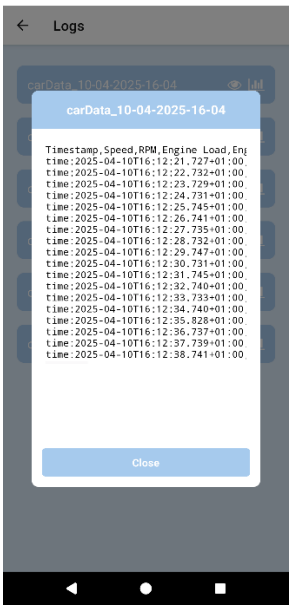


Figure 30 - Log View

There are icons along the same ‘block’ as the log name. Click on the eye the modal shows up. Click on the graph and your log is sent off to the backend to be processed. Once a score has been determined it is sent back along with a success property. The score is then displayed using an alert.

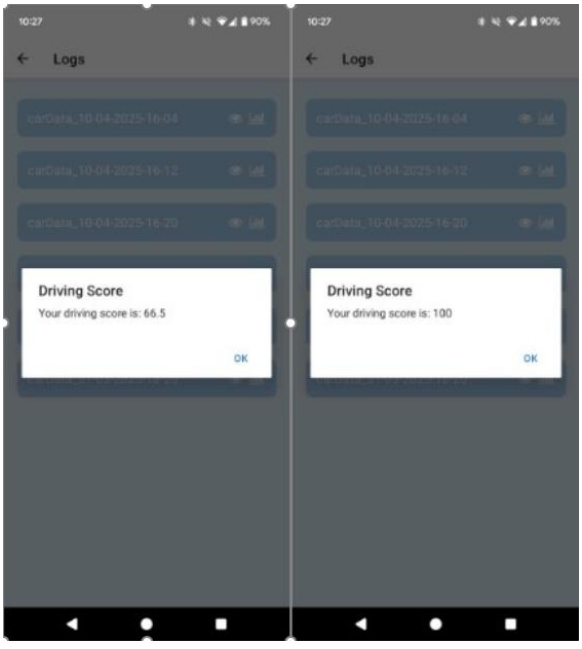


Figure 31 - Driver Score

8.5.5 Expo

As mentioned, I had a Mobile App Development module. To cover both iOS and Android, we didn't cover either specifically. Instead we used React Native to create cross platform apps and Expo Go to deploy them. Expo itself is a framework built on React Native so we can build for either operating system. It allows you to preview the app on your phone through Expo Go by scanning the QR code given when you build. It has built in support for any phones usual features, the camera, your location and notifications.

8.5.6 Speed Limits

Since I wanted to be able to give users a fair score, I thought it would be good to know the speed limit of the road they were on. This was done by using the 'expo-location' library and the overpass-api. A query is constructed with the users longitude and latitude, which allows us to get all nearby roads. From that I can get the speed limit attached (if defined) or infer from the road type.

```
const speedFromRoadType = (roadType) => {
    switch (roadType) {
        case 'motorway': return 120;
        case 'trunk':
        case 'primary':
        case 'secondary': return 100;
        case 'tertiary':
        case 'unclassified':
        case 'residential': return 50;
        case 'service': return 30;
        default: return 80; // fallback
    }
```

Figure 32 - Inferred Speed

8.6 Driver Score Algorithm

Driver behaviour is scored using a Node.js script that looks at raw driving data sent from the STM device. Each line of data is turned into an object with values like speed, RPM, engine load, and speed limit. These values are used to figure out how the user drove during the trip.

The score starts at 100 and changes as the trip goes on. If the driver speeds more than 10% over the limit, points are taken off based on how much they go over. The more they speed, the bigger the penalty, which can be up to 10 points per time.

```
//speed rules
if (speedLimit > 0 && speed > speedLimit) {
    const excessSpeed = speed - speedLimit;

    //10% allowance for speeding
    if (excessSpeed > speedLimit * 0.1) {
        //progressive penalty scale
        const buffer = speedLimit * 0.05; //scale down
        const adjustedExcess = excessSpeed - buffer;
        const rawPenalty = Math.pow(adjustedExcess, 1.3) * 0.02; //make the excess ^1.3 then scale down
        const penalty = Math.min(10, Math.floor(rawPenalty)); //round down to nearest whole number with a cap of 10

        score -= penalty;
    }
}
```

Figure 33 - Speed Rules

RPM is also checked. If the driver is redlining or holding high RPMs for too long, they lose points. The system gives some leeway based on how fast they're going.

```
//rpm rules
const rpmThreshold = (speed > 80) ? 3500 : 2500; //going onto a national? threshold increases

if (rpm > 7000) {
    score -= 3; //immediate penalty for redlining
    highRpmCount = 0;
} else if (rpm > rpmThreshold) {
    highRpmCount++;
    //penalty for sustained high RPM
    if (highRpmCount >= 10) {
        score -= 1 + Math.floor((rpm - rpmThreshold) / 500); //for every 500 rpm over, additional penalty
        highRpmCount = 0;
    }
} else {
    highRpmCount = Math.max(0, highRpmCount - 2); //account for gear changes
}
```

Figure 34 - RPM Rules

Engine load is handled the same way, if it stays high too long, the score drops.

```
//engine load rules
const loadThreshold = (speed > 60) ? 95 : 90;

if (engineLoad >= 100) {
    score -= 0.5; //aggressive driving
    highLoadCount = 0;
} else if (engineLoad > loadThreshold) {
    highLoadCount++;
    if (highLoadCount >= 8) {
        score -= 0.3 * (highLoadCount - 7);
        highLoadCount = 4; // Partial reset
    }
} else {
    highLoadCount = Math.max(0, highLoadCount - 2); //account for gear changes
}
```

Figure 35 - Engine Load Rules

Drivers can earn points back by driving smoothly. If they stay under the speed limit, keep RPMs low, and don't push the engine too hard, they get a small bonus after some time.

```
//reward for good driving
if (speed <= speedLimit &&
    rpm < 2500 &&
    engineLoad < 80) {
    smoothDrivingCount++;
    if (smoothDrivingCount >= 15) {
        score = Math.min(100, score + 1);
        smoothDrivingCount = 0;
    }
} else {
    smoothDrivingCount = 0;
}
```

Figure 36 – Reward For Good Driving

At the end of the trip, the score is slightly adjusted based on how long the trip was. Short trips stay about the same, and longer trips can get a small boost for consistent driving. The final score is rounded and sent back to the app, where the user can track their driving and see how they're doing over time.

It took a bit of adjustment between the three rules to get a balance of fairness and penalising where needed. That's why I have scaled the speed and count the number of times an action is done before a heavy deduction. As well as adding a reward for good driving, otherwise it runs the score even for a few mistakes over a long drive.

This script runs on an AWS EC2 instance, instead of inside the mobile app. This helps keep the app lightweight, since scoring can be handled in the background without putting any extra load on the user's device.

8.7 AWS

I make use of AWS (Amazon Web Services) to store a driver's logs (S3) and to host a server (EC2) that processes/analyses said logs. However, there is a risk of going over the free tier limitations, so I have had to setup CloudWatch to monitor these services to ensure that they don't exceed them.

8.7.1 S3

Briefly, Amazon S3 is a cloud storage service that I use to store small CSV files, usually only a few kilobytes in size. It acts like an online folder where I can upload, organise and retrieve these files from anywhere, making it ideal for saving data logs generated by recording a user's driving.

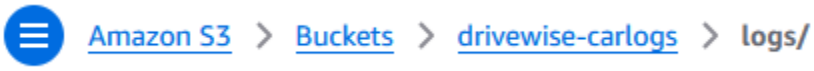


Figure 37 - AWS Bucket

Here I have a Bucket called 'drivewise-carlogs' which holds a folder of logs. Below is what is inside this folder.

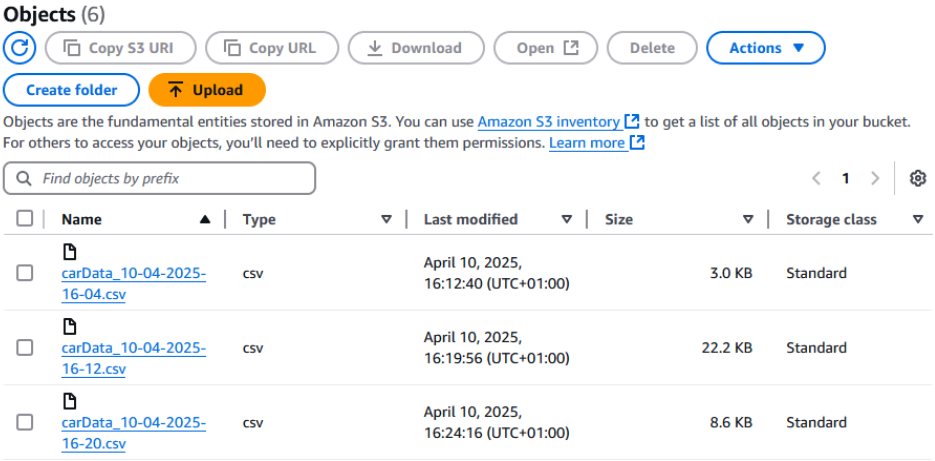


Figure 38 - What is inside the S3 Bucket

8.7.2 EC2

Amazon EC2 is a cloud-based virtual machine [23] that I use to run a Node.js application. This app takes in the CSV files stored in S3, processes the data and calculates a score, which is then sent to the mobile app for display. It allows me to run my backend logic remotely without needing any actual hardware. This helps me take some of the load off of my mobile app.

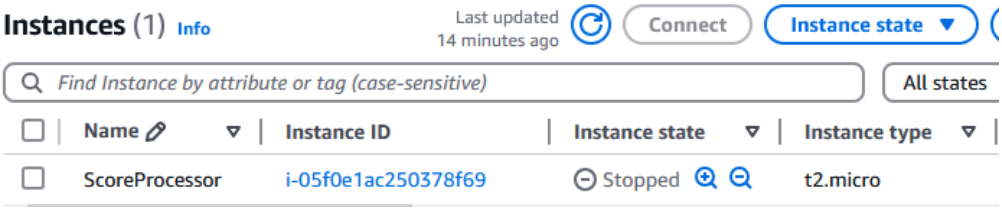


Figure 39 - EC2 Instance

Here is my instance, left off as I am not using it currently. From here I can remote in by starting it up and connecting. To do this securely, AWS provides a key pair, which consists of a public key (stored on the instance) and a private key (downloaded when the pair is created). This key pair acts like a password, allowing secure authentication when connecting. If I wanted to upload files (like my server code), I would use the private key with software like WinSCP to securely transfer files to and from the instance over SSH.

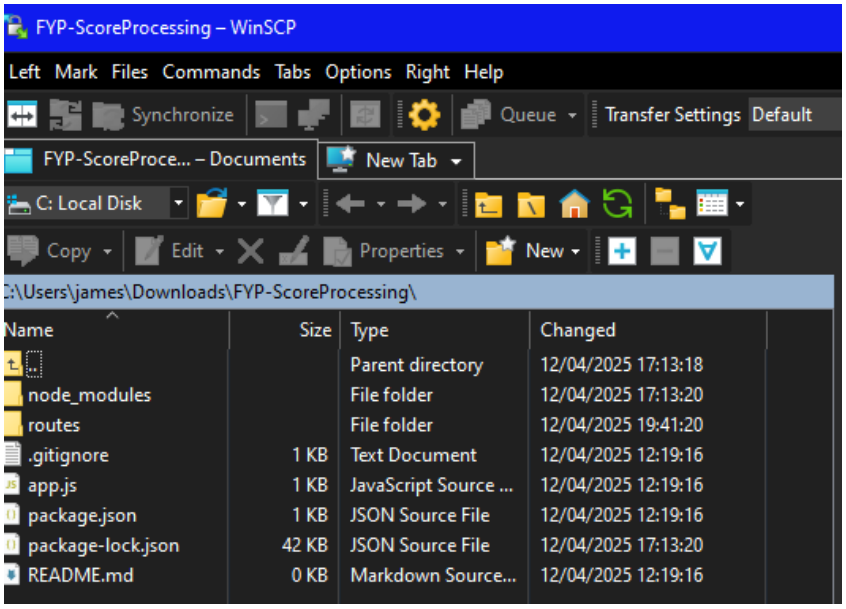


Figure 40 - WinSCP

This is what WinSCP looks like, here I can simply just drag/drop code from my local machine to the EC2 instance, it isn't ideal during development as small changes take time to constantly upload but once the code is complete it is simple to do.

8.7.3 CloudWatch

CloudWatch is a monitoring and logging service provided by AWS [24]. I only set it up as a precaution, since I had heard of past students getting caught out by unexpected usage bills. During configuration, I set up alarms to monitor any service I had used (S3, EC2 and itself). I also configured email and SMS alerts so that if any of these metrics went over a certain threshold, 2 euro, I'd get notified straight away. This gave me some peace of mind that if anything started using too many resources, I'd know about it before it became any sort of a problem.

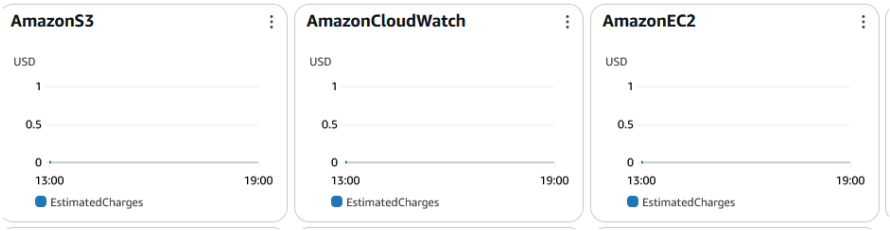


Figure 41 - Services used

8.7.4 Accessing AWS through code

8.7.4.1 Functions

The following functions are used to handle file interactions with my S3 bucket. They cover uploading CSV logs from the mobile app, retrieving a list of saved logs, and reading the contents of selected log files. These are used by the mobile and server applications to manage data efficiently in the cloud.

```
> export async function parseTextFile(fileName) {···
  }

> export async function getLogNames() {···
  }

> export async function uploadCsvToS3(fileUri, fileName) {···
  }
```

Figure 42 - Functions for AWS

The parseTextFile function is charge of retrieving a specific file from my S3 bucket and returning its contents as a string. It builds a GetObjectCommand using the file name provided, targeting

the logs folder within the bucket. The function sends this command to AWS using the S3 client. Once the file is received, the body of the response is converted into a string using the transformToString() method. This content is then returned to be either displayed for user consumption or to pass to the backend code that is hosted on the EC2 instance.

```
try {
  const command = new GetObjectCommand({
    Bucket: awsConfig.bucket,
    Key: `logs/${fileName}`,
  });

  const data = await s3Client.send(command);

  const body = data.Body;
  const stream = body?.transformToString();

  const fileContent = await stream;
  return fileContent;
```

Figure 43 – parseTextFile

The uploadCsvToS3 function handles uploading a CSV file from the device to the S3 bucket. It begins by reading the contents of the file from the local file system as a string. This ensures the data is in the proper format for uploading. It then creates a PutObjectCommand, setting the bucket, the file path within the logs folder, the file content, and the content type as text/csv. The command is sent to AWS using the S3 client, and if successful, a confirmation is logged

```
const command = new PutObjectCommand({
  Bucket: awsConfig.bucket,
  Key: `logs/${fileName}.csv`,
  Body: fileContent,
  ContentType: 'text/csv',
});

const result = await s3Client.send(command);
console.log('Successfully uploaded CSV to S3:', fileName);
```

Figure 44 - uploadCsvToS3

8.7.4.2 Credentials

There are many secure ways to connect to AWS, including using IAM roles, AWS Cognito or the AWS Secrets Manager. For DriveWise, I chose to store my credentials in a .env file. This approach is simple to implement, keeps sensitive keys out of the main source code and makes it easier to manage credentials during development. As long as the .env file is excluded from

version control (which it is, via gitignore), it gives me a secure and lightweight solution for small-scale apps like mine.

To access the credentials in the code, I used the @env package. This library allows me to privately pull environment variables defined in the .env file into my frontend. These values are then used to configure the AWS SDK within the functions mentioned above, such as uploading files to S3 or retrieving data from it, without hardcoding any sensitive information.

```
1  import { AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_REGION, AWS_BUCKET } from '@env';
2
3  export const awsConfig = {
4    region: AWS_REGION,
5    accessKeyId: AWS_ACCESS_KEY_ID,
6    secretAccessKey: AWS_SECRET_ACCESS_KEY,
7    bucket: AWS_BUCKET,
8  };
```

Figure 45 - Pull in values from .env

8.7.5 Limitations

The free tier for Amazon S3 and EC2 comes with several limitations. For S3, users get 5GB of standard storage, 20,000 GET requests and 2,000 PUT requests per month. EC2 offers 750 hours per month of a t2.micro instance (this depends on the region but accurate for us-east-1 which is what I use). This enough to run one instance continuously over a month. However, resources like CPU, RAM and bandwidth are limited, and exceeding any of these thresholds will result in charges [25]. Thankfully, these limits are suitable for small-scale projects like mine, just not ideal for production use or heavy workloads.

8.8 Security

8.8.1 TLS

“Transport Layer Security (TLS) is a protocol that encrypts communication over the internet”. In this project, TLS is used to secure the connection between the microcontroller and Ubidots. It ensures that the data published from the device cannot be read or changed while it is being transmitted. The STM device also uses a certificate (embedded in code) to verify the identity of the broker before sending any data. This adds an extra layer of trust and helps protect against man in the middle attacks or unauthorised access [26].

8.8.2 WSS

"WebSockets allow for two-way communication between a client and a server over a single connection" [27], which stays open so data can be sent and received in real time. WSS is the secure version of WebSockets that uses TLS to encrypt the connection, protecting the data from being read or changed during transmission [28]. I use this to make sure the data being passed from Ubidots is secure and safe for use in my mobile app.

9 Challenges & Solutions

I have had many issues pop up and be resolved throughout this project but these are the most relevant, or what I believe to have been the most difficult to deal with.

9.1 Memory Issue

This particular issue had stumped me for a few days, I was attempting to connect to Ubidots over TLS, it was working with no security before this, and I had never actually came across memory management problems. What happened was, for me to connect to the broker, the CA certificate needed to be stored in memory to be passed into the socket options, but the stack size of the task was too small. All I had to do was increase it and it worked after.

9.2 Ubidots Limitations

Ubidots has a rate limit of 5 requests per second for its free trial accounts, these last for 30 days. I was publishing a consistent 9 requests with some delays to prevent limiting, but the data was coming in too slow. I implemented an array of PIDs that were to be pushed up, the index would increment and different data would be sent to the broker. For me to keep doing this, I also must create a new account every 30 days for me to keep using it at the rate of 5/s.

9.3 Websocket Crashing

A small bug I should have noticed sooner, the clientId for a connection to Ubidots needs to be unique, other it will assume it is the same device trying to connect twice and close one. My websocket was continuously crashing, I looked over both configurations and until I read the documentation, I assumed it was fine to do this. After I changed the clientId it had no issues after.

10 Project Plan

I have used JIRA to manage my project, the main reason I have chosen it is due to my experience on work placement. My team would follow a Kanban framework which is like a board of work items, where you just take tickets when you're able to do more work. I liked this method, it felt freeing getting to choose what I worked on and not having heavy deadlines for tasks.

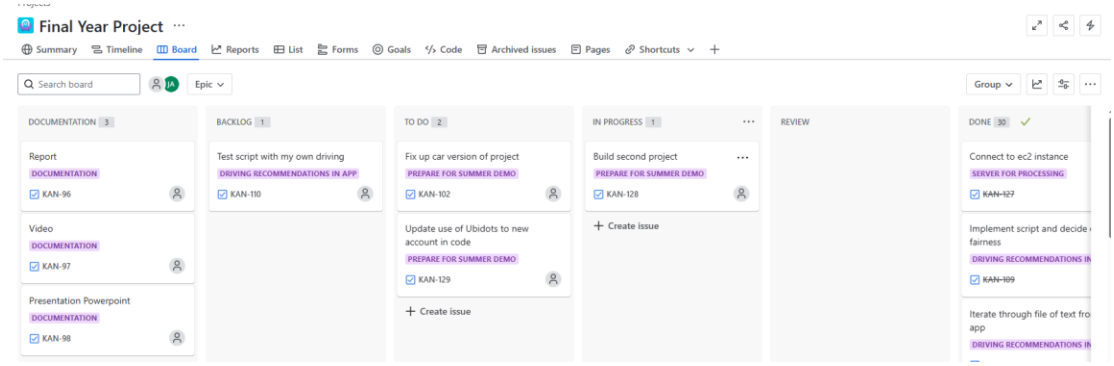


Figure 46 - Kanban Board

This is my one, it's a bit empty in this image as I am nearing the end of the project. The way I created my Epics (big tasks) was to really break down what had to be done. From there I would do the same again and those became tasks sorted into their relevant Epics. The main thing I did for consistency was adding the link to my board as a tab that immediately popped up whenever I opened my browser. This small thing really helped me keep on top of it. As a way to show work done, I use a 'Cumulative Flow Diagram' which just shows the status of my work over time, so you can see below the movement from TODO -> DONE.

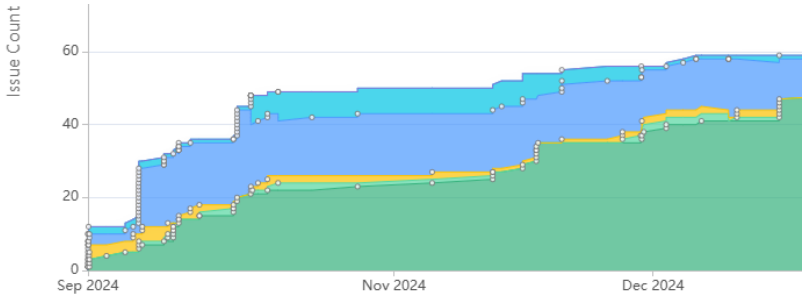


Figure 47 - Cumulative Flow Diagram

11 Ethics

My project tries to push users to be safer and more fuel efficient driving by showing them how they drive and helps them become aware of it. This can reduce aggressive driving and lower their emissions over time. It lines up with three of the UN's Sustainable Development Goals (SDG) like SDG 3, which focuses on health and safety, SDG 11, which aims to improve how we live and move around in cities and SDG 13, which is about taking action on climate change [29]. With this, my project can play a small part in making transport safer and more sustainable for everyone.

12 Conclusion

After roughly 8 months of consistently working on DriveWise, I have completed everything I had set out to do, I developed a working product that connects directly to a vehicle's OBD port, processes real-time data and scores a user based on their driving through a mobile app. The app is passed in live car data, records it locally and stores it securely in an AWS S3 bucket. A server hosted on an EC2 instance then uses this data to analyse driving behaviour and returns a score based on their speed, RPM and load on the engine to the app. Throughout, I overcame challenges around CAN communication, hardware interfacing, prototyping, mobile app development and secure communications. The result is a full-stack product that gave me experience with embedded systems, backend development, AWS and app development. All of this works together to give users more control over their own driving data.

13 References

[1] G. M. Smith, "What Is CAN Bus (Controller Area Network) and How It Compares to Other Vehicle Bus Networks," DEWESoft.com, 13 Feb 2024. [Online]. Available: https://dewesoft.com/blog/what-is-can-bus. [Accessed 12 Apr 2025].

[2] Siemens, "Automotive ethernet," Siemens, 2025. [Online]. Available: https://www.sw.siemens.com/en-US/technology/automotive-ethernet/. [Accessed 12 Apr 2025].

[3] National Instruments, "FlexRay Automotive Communication Bus Overview," ni.com, 01 July 2024. [Online]. Available: https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/flexray-automotive-communication-bus-overview.html. [Accessed 12 Apr 2025].

[4] Martin Flach, "LIN Bus Explained - A Simple Intro [2025]," CSSElectronics.com, Jan 2025. [Online]. Available: https://www.csselectronics.com/pages/lin-bus-protocol-intro-basics. [Accessed 12 Apr 2025].

[5] H. Y. Yeo, "Automotive Ethernet: The In-Vehicle Networking of the Future," Keysight.com, 12 Feb 2024. [Online]. Available: https://www.keysight.com/blogs/en/tech/educ/2024/automotive-ethernet. [Accessed 12 Apr 2025].

[6] ni-xnet, "CAN Frames," ni.com, 12 June 2024. [Online]. Available: https://www.ni.com/docs/en-US/bundle/ni-xnet/page/can-frames.html. [Accessed 13 Apr 2025].

[7] Geotab Team, "What is OBDII? History of on-board diagnostics," geotab.com, 25 Jan 2023. [Online]. Available: https://www.geotab.com/blog/obd-ii/. [Accessed 13 Apr 2025].

[8] A. Moore, "The 10 modes of OBD-II," vehicleservicepros.com, 01 Jan 2020. [Online]. Available: https://www.vehicleservicepros.com/service-repair/the-garage/article/21181910/the-10-modes-of-obd-ii. [Accessed 13 Apr 2025].

[9] M. K. Pratt, "Top 12 most commonly used IoT protocols and standards," techtarget.com, 12 July 2023. [Online]. Available: https://www.techtarget.com/iotagenda/tip/Top-12-most-commonly-used-IoT-protocols-and-standards. [Accessed 13 Apr 2025].

[10] I. Craggs, "Understanding the Differences Between MQTT and HTTP," hivemq.com, 25 Mar 2024. [Online]. Available: https://www.hivemq.com/blog/mqtt-vs-http-protocols-in-iot-iiot/. [Accessed 13 Apr 2025].

[11] EMQX Team, "MQTT vs CoAP: Comparing Protocols for IoT Connectivity," emqx.com, 23 Apr 2024. [Online]. Available: https://www.emqx.com/en/blog/mqtt-vs-coap. [Accessed 13 Apr 2025].

[12] N. Clement, "AMQP vs MQTT: Messaging protocols compared," cloudamqp.com, 20 June 2023. [Online]. Available: https://www.cloudamqp.com/blog/amqp-vs-mqtt.html. [Accessed 13 Apr 2025].

[13] Developer Relations Team, "What are WebSockets ws:// & wss:// connections," pubnub.com, 03 Sept 2023. [Online]. Available: https://www.pubnub.com/guides/websockets/. [Accessed 13 Apr 2025].

[14] navixy, "WebSocket integration for real-time vehicle sensor data visualization," navixy.com, 21 Oct 2024. [Online]. Available: https://www.navixy.com/blog/websocket-integration/. [Accessed 13 Apr 2025].

[15] Mikroe, "ATA6563 CLICK," mikroe.com, 2025. [Online]. Available: https://www.mikroe.com/ata6563-click. [Accessed 13 Apr 2025].

[16] B. Watson, "CAN Transceiver vs CAN Controller – What are the Differences and How are They Used?," totalphase.com, 20 Nov 2024. [Online]. Available: https://www.totalphase.com/blog/2024/11/can-transceiver-vs-can-controller-what-are-differences-how-are-they-used/. [Accessed 13 Apr 2025].

[17] OZEN Elektronik, "Multiple Protocol OBD ECU Simulator mOByDic4910," ozenelektronik.com.tr, 2025. [Online]. Available: https://www.ozenelektronik.com.tr/multiple-protocol-obd-ecu-simulator-p.html. [Accessed 13 Apr 2025].

[18] Murata Power Solutions, "Encapsulated Non-Isolated Switching Regulator DC-DC," 2025. [Online]. Available: https://docs.rs-online.com/912f/0900766b81634806.pdf. [Accessed 13 Apr 2025].

[19] B. Gunasekaran, "FreeRTOS, Everything You Need To Know..!," embeddedinventor.com, 02 Aug 2019. [Online]. Available: https://embeddedinventor.com/freertos-a-complete-beginners-guide/. [Accessed 15 Apr 2025].

[20] sae, "E/E Diagnostic Test Modes J1979_201202," sae.org, Feb 2012. [Online]. Available: https://www.sae.org/standards/content/j1979_201202/. [Accessed 14 Apr 2025].

[21] CSSelectronics, "OBD2 PID Overview [Lookup/Converter Tool, Table, CSV, DBC]," 2025. [Online]. Available: https://www.csselectronics.com/pages/obd2-pid-table-on-board-diagnostics-j1979. [Accessed 14 Apr 2025].

[22] st.com, "vlegalwaymayo.atu.ie," 20 Sept 2024. [Online]. Available: https://vlegalwaymayo.atu.ie/pluginfile.php/1279564/mod_resource/content/3/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. [Accessed 14 Apr 2025].

[23] AWS, "What is Amazon EC2," docs.aws.amazon.com, 2025. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. [Accessed 14 Apr 2025].

[24] AWS, "What is Amazon CloudWatch," docs.aws.amazon.com, 2025. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html. [Accessed 14 Apr 2025].

[25] AWS, "AWS Free Tier," aws.amazon.com, 2025. [Online]. Available: https://aws.amazon.com/free/. [Accessed 14 Apr 2025].

[26] cloudflare, "What is TLS?," cloudflare.com, 2025. [Online]. Available: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/. [Accessed 14 Apr 2025].

[27] http.dev, "Encrypted WebSocket," http.dev, 02 Aug 2023. [Online]. Available: https://http.dev/wss. [Accessed 14 Apr 2025].

[28] A. Dizdar, "WebSocket Security: Top 8 Vulnerabilities and How to Solve Them," brightsec.com, 25 Mar 2025. [Online]. Available: https://www.brightsec.com/blog/websocket-security-top-vulnerabilities/. [Accessed 14 Apr 2025].

[29] United Nations, "The 17 Goals," sdgs.un.org, 2025. [Online]. Available: https://sdgs.un.org/goals. [Accessed 14 Apr 2025].

14 Code

GitHub

Note: Any repository with FYP as the start of its name is relevant to this project.