COMS Car Project

Duncan Kroot, Dylan Loers, Thijs Berings

November 2024

Contents

1 Introduction

The objective of this assignment is to gain a better understanding of distributed systems. This objective will be accomplished by completing this project. In this project a RTOS and several protocols will be utilized. This project consists of simulating a car by creating several ECU following requirements outlined in the assignment document.

2 Analysis

2.1 Problem statement

Testing new components and/or firmware is currently a difficult and time-consuming process. To simplify and speed up this process. A physical test environment will be designed and created.

2.2 Use cases and scenarios

Component	**Description**
Use case	Users can use the remote application to control the headlights of the vehicle.
Actor	User
Scenario	The user leaves the house at night. To be able to see anything, the user utilizes the app to turn on the headlights. The car receives this command and turns on the headlight.

Table 1: UC1 and SC1

Component	**Description**
Use case	When the User presses on the brake pedal. The engine must stop immediately.
Actor	User
Scenario	The user is driving and an obstacle appears on the road. To prevent a collision the user presses the brake pedal. This signal is sent to the Engine Control Unit. The engine control unit receives this signal and turns off the engine.

Table 2: UC2 and SC2

Component	**Description**
Use case	The user can use the app to see the current vehicle state.
Actor	User
Scenario	The user notices that some functionality is not working. To check the current vehicle status. The user uses the app to check what the status is of each individual ECU.

Table 3: UC3 and SC3

2.3 Requirements

2.3.1 Must have

- The headlights must be able to be switched on and off via a dashboard and via a remote-control app.
- The indicators must be able to be switched on and off via a dashboard and a remote-control app.
- The speed of the engine must be controllable via the dashboard and a remote-control app.
- The brakes can be controlled via the dashboard. If the brakes are pressed, then the engine should stop immediately.
- The temperature of the engine must be readable via the app.
- The temperature of the engine should be updated every two seconds.

Component	Description
Use case	A mechanic can access the data stored on the EDR using the app.
Actor	Mechanic
Scenario	The car is in a repair shop for a routine service. Part of the routine service is to check the system's logs. The mechanic accesses these logs via the app.

Table 4: UC4 and SC4

- If the temperature of the engine exceeds a certain value, then it must be made visible in the app and on the dashboard.
- Tire pressure should be able to be read via an app.
- The tire pressure should be updated every two seconds.
- If the tire pressure gets too low, it should be made visible both in the app and on the dashboard.
- If one of the ECU's fail this should be shown in the remote control app and the dashboard.
- An employee of a car repair shop needs to be able to read the contents of the EDR via the app on request.

2.3.2 Should have

- The car should have an EDR with recorded events as speed, user activities, temperature, tire pressure etc. for the last x minutes.
- CAN-bus messages should have priority over message from the remote control app.
- The system should be robust, reliable and real time.

2.3.3 Could have

- The car has cruise control that can be operated via an app and the dashboard.

2.3.4 Will not have

- The app will not be a mobile app.

3 System Overview

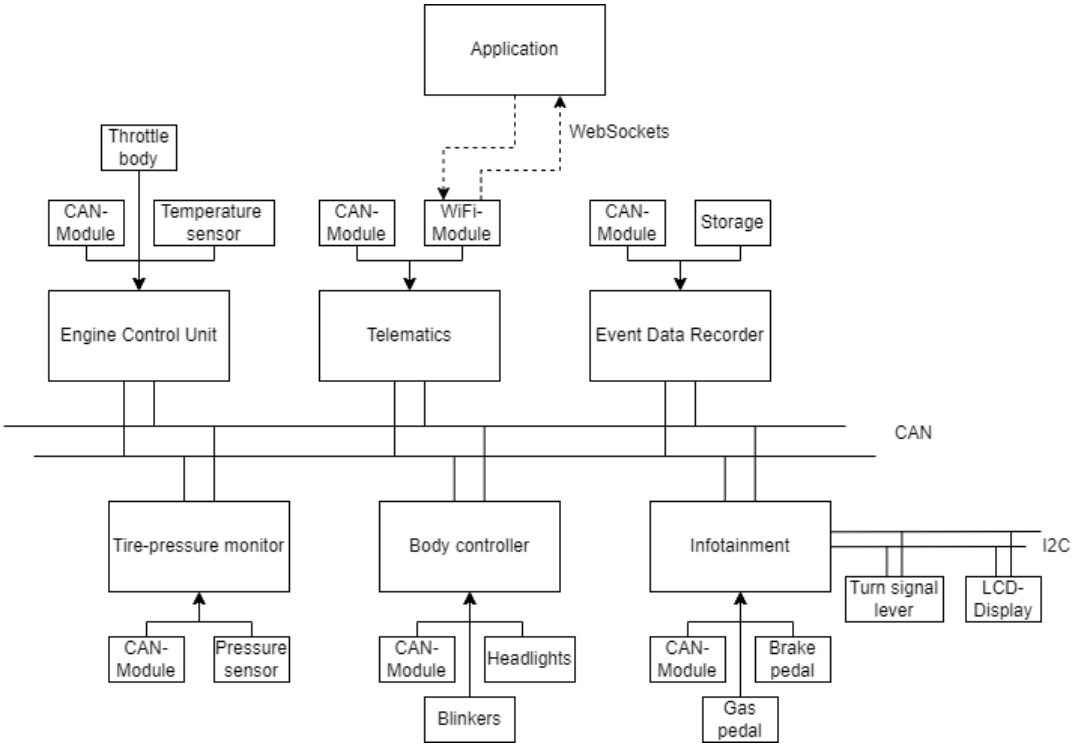

Figure 1: System overview

3.1 Description

As shown in Figure 1. The system consists of six modules. These modules communicate with each other via the CAN bus. Currently, CAN communication is handled with external modules. Later, this may be changed to the built-in TWAI (Two Wire Automotive Interface).

The engine control unit is responsible for managing the engine. For this project, that means handling throttle input, engine temperature, and measuring engine speed. The telematics module allows the vehicle to receive messages from the remote application. It does this by using web-sockets. Using web-sockets for this allows for seamless info updates with less overhead that using just http. The EDR (Event Data Recorder) keeps a record of all data sent over the CAN bus. If something were to break. The mechanic can retrieve these records and use them for troubleshooting. It stores these records on an SD card or the built-in flash memory. This is something that will be decided later in the project. The tire-pressure monitor keeps track of the tire pressures for all tires. It accomplishes this by using a pressure sensor for each tire. For this project only one sensor used. The body controller is in charge of managing the headlights and blinkers. The infotainment is consists of all the components used for human interaction. This includes the turn signal lever, headlight button, brake pedal, gas pedal and a display. The display will allow visual representation of information. Both the turn signal lever and the LCD-Display will be controlled via the I2C bus. This simplifies the software.

To simulate the functionality of this system. Different components will be used to simulate the sensors. This allows simple control when demonstrating and testing. In section 7 Hardware description the specific used components are outlined.

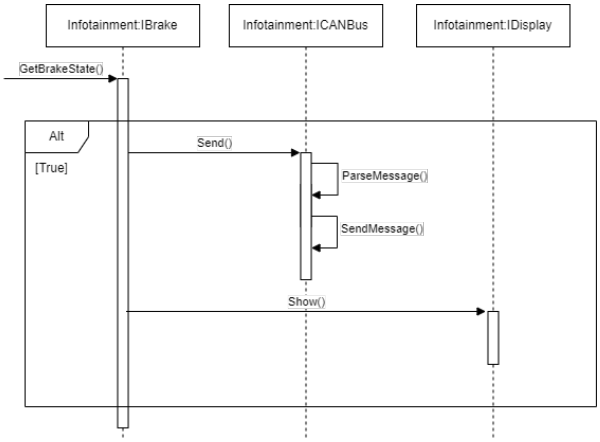

Figure 2: UML Sequence diagram showing braking

3.2 System interaction

3.2.1 Diagram Description

Figure 2 depicts the interactions between objects when the user presses the brake pedal. First the GetBrakeState() method in the Brake object is called. This checks wether the brake is being used. If the brake is being used. A message will be created. This message is then sent to the CANbus object. The CANbus object then formats the message to fit into a CAN data frame. After this is sends it via the CANbus with the sendMessage() method. After sending this data over the CAN bus. The brake status on the display is updated with the Show() method.

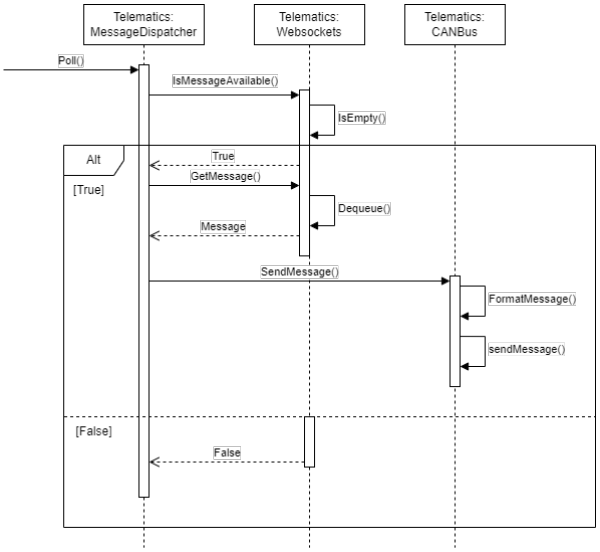

Figure 3: UML Sequence diagram showing receiving data from the app

3.2.2 Diagram Description

Figure 3 show the interactions between objects when data is received from the remote application. First the MessageDispatcher object checks wether a new message has arrived. If a new message did arrive, it retrieves is with the GetMessage() method. The message is then send to the CANbus module via the SendMessage() method. This method takes the message as a parameter. The SendMessage() method first formats the message into a CAN data frame. Then sends it via the CAN bus using the sendMessage() method.

4 Protocol Design

The communication between the ECUs will follow a strict protocol to ensure that the meaning of messages is clear. CAN bus provided an arbitration field which will be used for 2 things. The arbitration of message dictates it's priority on the CAN bus, a lower arbitration value meaning higher priority. Additionally, it serves as a message identifier. On a CAN bus, all messages are broadcast to all connected nodes, and addressed based on message ids as opposed to node ids. This is what allows the prioritization in the first place and what keeps the network topography simple. This means that ECUs will have to decide whether a message is important for them to interpret or not. Next to this arbitration field is an 8 byte data field that will hold the data itself, e.g. for tire pressure the pressure in PSI/bar or the engine temperature in celsius.
The arbitration field is split into 3 different fields to make priority easy to handle, show what message is sent, and show what ECU has sent this message. This mostly serves to make the field more human-readable. The most significant bits represent the priority. This is done to make sure that this has the most impact on the prioritization of the message on the bus. The next group of bits represents the source ECU; this both gives some information about the importance of the message, as the engine controller has higher priority than the blinkers, for example, but also makes it easy to detect where a message is coming from during debugging/testing and in the event data recorder. Lastly, the message type should have little impact on the priority of the message, while being very important to decipher the meaning of the message, this is why it's a bigger range than the other fields while also being at the end of the arbitration field.

Bit Range	Meaning
[10:8]	Priority
[7:5]	Source ECU
[4:0]	Message type

Table 5: Sub-fields of the arbitration field

The messages themselves are shown in table 6 below:

Priority	**Source ECU**	**Message Type**	**Message Description**
001	000 (Engine Controller)	00011 (Engine Overheat Warning)	Engine Overheat Warning
001	011 (Infotainment)	10101 (Brake Pressed Event)	Brake Pressed
001	001 (Body Controller)	00100 (Brake Command)	Brake Command (Engine Cut)
010	000 (Engine Controller)	00010 (Engine Power Control)	Engine Power Control (Throttle)
010	001 (Body Controller)	01000 (Headlights Toggle)	Headlights Toggle
011	010 (Tire Pressure ECU)	10000 (Tire Pressure Warning)	Tire Pressure Low Warning
011	010 (Tire Pressure ECU)	10001 (Tire Pressure Update)	Tire Pressure Update
101	011 (Infotainment)	10100 (User Input for Blinkers)	Blinkers Control (Joystick)
110	011 (Infotainment)	11001 (Gas Pedal Input)	Gas Pedal Value
111	000 (Engine Controller)	11110 (Engine Temperature Update)	Engine Temperature Update
100	100 (Telematics ECU)	Any message type needed	Any data related to message type

Table 6: Message Types and Descriptions

The data formatting for each is explained further in the following table (table 7):

Message Description	**Data Format**
Engine Overheat Warning	Byte 0: 1 = Overheat, Byte 1-2: Temperature (uint16)
Brake Command (Engine Cut)	Byte 0: 1 = Apply Brake
Engine Power Control (Throttle)	Byte 0: Power Level (0-255)
Headlights Toggle	Byte 0: 1 = On, 0 = Off
Tire Pressure Low Warning	Byte 0: Tire ID (1-4), Byte 1: Pressure Status (1 = Low Pressure)
Tire Pressure Update	Byte 0-1: Tire #1 Pressure (kPa), Byte 2-3: Tire #2 Pressure, ...
Dashboard Telemetry	Byte 0: Status Code, Byte 1: Temperature (uint16), Byte 2: Speed (uint16)
Remote Control Command	Byte 0: Command Type (1 = Power, 2 = Light), Byte 1: Value
Blinkers Control (Joystick)	Byte 0: Direction (1 = Left, 2 = Right)
Brake Pressed	Byte 0: 1 = Pressed, 0 = Released
Gas Pedal Value	Byte 0: Gas Pedal Position (0-255)
Engine Temperature Update	Byte 0-1: Temperature (uint16)

Table 7: Message Data Formats

5 Module Designs

5.1 Class Diagrams

A common theme between all class diagrams below is the ICANBus interface, which is used to abstract away the used CAN bus library. The other interfaces used provide some levels of abstraction, which will be useful when hardware needs to be changed as well as during testing through mock objects.

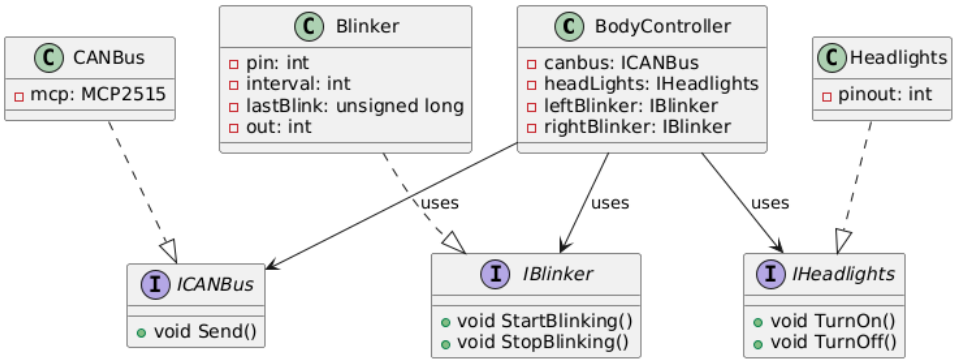

Figure 4: Body Controller UML Diagram

Figure 4 shows the class diagram for the body controller. This provides connections to the headlights and blinkers, using interfaces to abstract away the exact hardware, whether it is an LED for demo purposes or a relay in a real world application.

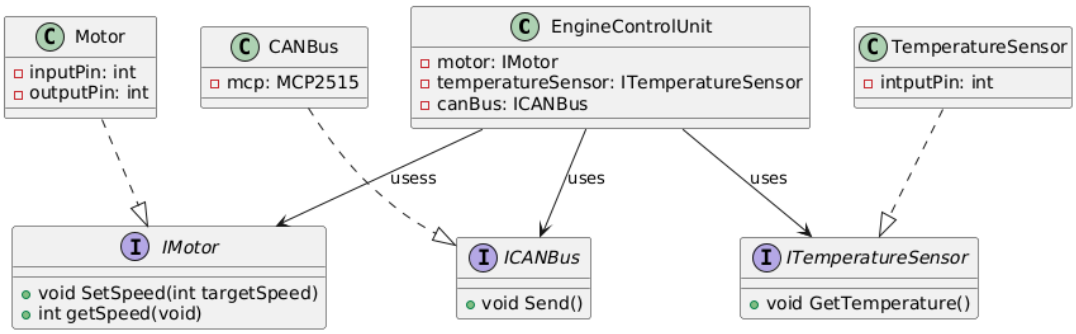

Figure 5: Engine Control unit UML Diagram

Figure 5 shows the class diagram of the engine control unit. It has a motor interface that can both write a speed to the motor and retrieve it. It uses the same ICANBus interface as the other diagrams. The temperature sensor is also abstracted away into an interface, so a potentiometer can be used during the demo and testing.
The event data recorder class diagram shown in figure 6 shows the ICANBus interface and a second IStorage interface. This gives the Event Data Recorder class the supported functionality of storing data, even when the system is powered down.
The infotainment class diagram in figure 7 shows the human interfacing of the system. It provides the gaspedal, brake, joystick for the turning indicators and headlight button.
The telematics controller shown in figure 8 acts as the bridge between the CAN bus and the application. It uses the CANBus class and WebSockets class to achieve this.
Lastly the tire pressure class diagram shown in figure 9 shows an interface that communicates with a pressure sensor.

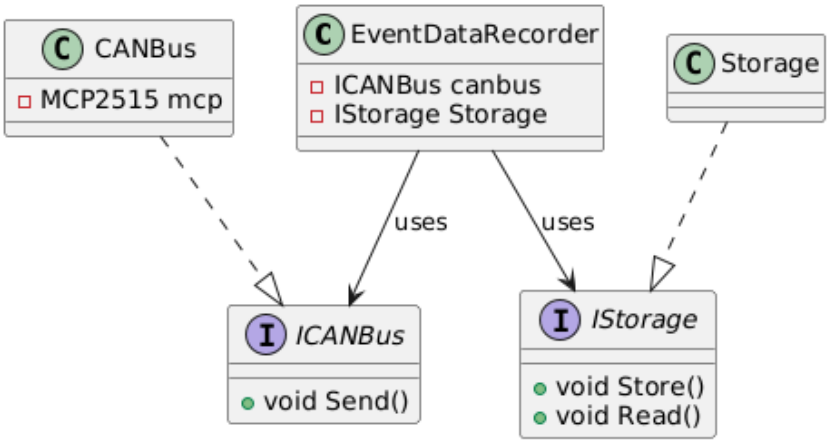

Figure 6: Event Data Recorder Class Diagram

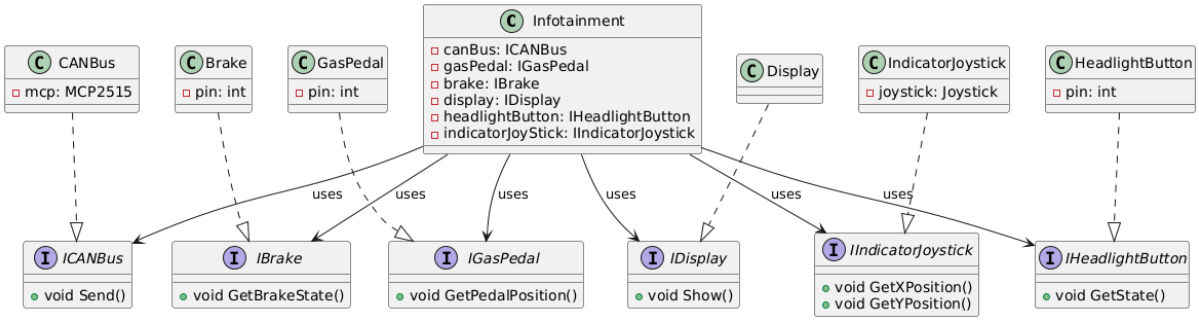

Figure 7: Infotainment Class Diagram

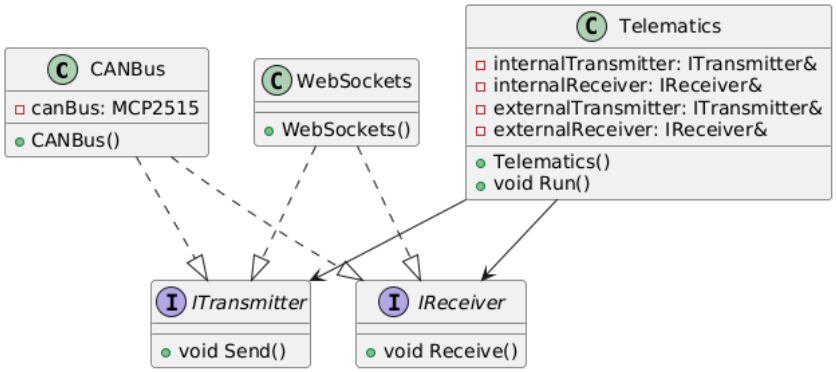

Figure 8: Telematics Class Diagram

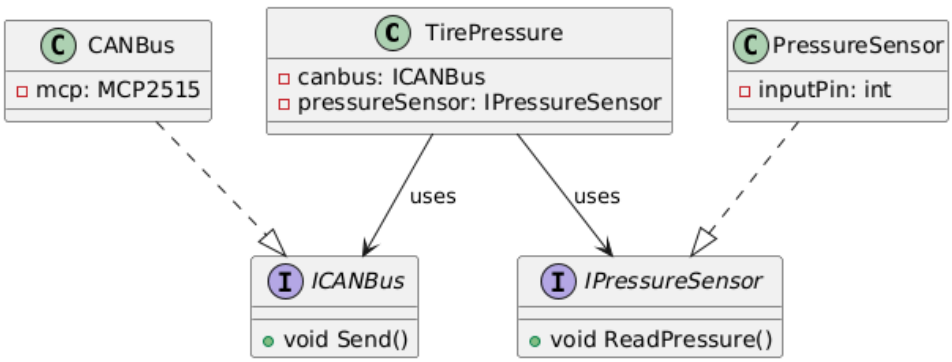

Figure 9: Tire Pressure Class Diagram

6 Testing strategy

6.1 Introduction

In order to ensure code quality and adherence to requirements, a solid testing strategy is necessary. This testing strategy will describe what aspects of the project will be tested, how they will be tested, and what actions should be taken when a test fails.

6.2 Overview

6.2.1 Unit testing

Because of nature of this project, having multiple units connected through a communication bus, each module will be tested separately. By unit tests. This ensures that each individual module behaves as expected. These tests will be written using the Google test framework.

6.2.2 Integration testing

To test if all modules are well integrated and the system as a whole is functional, integration tests must be carried out. Automating these tests is not a priority due to the size and scope of this project. Therefore, these tests will consist of walking manually through the scenarios as described in the design.

6.3 Testing responsibility and defect management

Whoever is responsible for writing a section of code is also responsible for ensuring that this section is fully tested. The Gitlab issue board will be used to keep track of this. In order for an issue to be marked as closed, a second person must review if the work is up to standard.

7 Hardware description

7.1 Shared components

Every ECU communicated over the CAN bus. This is achieved by utilizing the QYF-983 module. This may be changed later in the project in favor of TWAI. This cuts down on hardware components and thus reducing cludder and improving reliability. All ECUs will be utilizing the ESP32 as the MCU.

7.2 Body Controller

For this project the body controller will only control the lights. To simulate this, LED's will be utilized.

7.3 Engine control Unit

The engine control unit controls as the name implies everything related to the engine. For this project this includes the engine speed and the engine temperature sensor. To simulate this a Parallax feedback 360 will be utilized. This allows us to visualize the throttle control and measure engine the speed in RPM.

7.4 Event Data Recorder

The event data recorder stores historical data of the state of the car. It does this by monitoring the CAN and recording states of different ECUs. This controller will use the same hardware as the rest, but could include a micro sd card module as the storage solution. This could have two benefits, it has more storage capacity than the on board flash of the mcu and, if formatted properly, can be read out on a computer directly.

7.5 Infotainment

The infotainment system provides human interfacing for the network of ECUs. It does so by presenting a joystick for the blinkers on one axis and the headlights on the other axis. It has a button to simulate the brake. A potentiometer is used to simulate the gas pedal's variable pressure. Lastly, an LCD display shows any important information to the user, such as ECU failures/warnings, speed and what blinkers are active.

7.6 Telematics

The telematics module is responsible for transmitting and receiving data to and from the app.

7.7 Tire-Pressure monitor

The tire-pressure monitor is responsible for taking tire pressure measurements. To simulate the tire pressure, a potentiometer will be utilized. This allows use to quickly change pressure tires.