Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

TELECOMMUNICATION ENGINEERING MASTER´S DEGREE

FINAL YEAR PROJECT

DESIGN AND IMPLEMENTATION OF CAN COMMUNICATION FOR OWASYS IOT DEVICES

owasys
Advanced Wireless Devices

Student: Orbe Deusto, Unai

Director: Jacob Taquet, Eduardo
Codirector: Huarte Arrayago, Maider

Course: 2022-2023

Date: Bilbao, 12th of September of 2023

Contents

List of Figures

List of Tables

Abstract Laburpena Resumen

This project presents the design and implementation of a CAN (Controller Area Network) communication for Owasys IoT devices. Owasys designs and creates embedded Linux devices for vehicle fleet management, DTCO, Industry 4.0… These devices offer a CAN interface but no platform to start implementing it for the clients, and this project stands as the first prototype to offer it. A Python code will be written alongside the necessary open-source libraries to connect to any commercial car´s ECU (Electronic Control Unit) and retrieve data from it. This data will be then sent to a server where the client can access the data in real time.
Key words: CAN, OBD, Python, IoT, Linux

Proiektu honek Owasys IoT gailuetzat CAN (Controller Area Network) komunikazio baten diseinua eta ezarpena aurkezten du. Owasys-ek ibilgailuen flota kudeatzeko, DTCO edota Industria 4.0-rako Linux gailu txertatuak diseinatzen eta sortzen ditu... Gailu hauek CAN interfaze bat eskaintzen dute baina bezeroei inplementatzen hasteko plataformarik ez, eta proiektu honek horretarako lehen prototipoa eskaintzen du. Python kode bat idatziko da beharrezko kode irekiko liburutegiekin batera edozein auto komertzialeko ECU-arekin (Electronic Control Unit) konektatzeko eta bertatik datuak berreskuratzeko. Ondoren, datu horiek zerbitzari batera bidaliko dira, non bezeroak datu horiek denbora errealean atzi ditzakeen.
Gako-hitzak: CAN, OBD, Python, IoT, Linux

Este proyecto presenta el diseño y la implementación de una comunicación CAN (Controller Area Network) para los dispositivos IoT de Owasys. Owasys diseña y crea dispositivos Linux embebidos para la gestión de flotas de vehículos, DTCO, Industria 4.0... Estos dispositivos ofrecen una interfaz CAN pero no una plataforma para empezar a implementarla para los clientes, y este proyecto se erige como el primer prototipo para ofrecerla. Se escribirá un código Python junto con las librerías de código abierto necesarias para conectarse a la ECU (Unidad de Control Electrónico) de cualquier coche comercial y recibir datos de la misma. A continuación, estos datos se enviarán a un servidor donde el cliente podrá acceder a ellos en tiempo real.
Palabras clave: CAN, OBD, Python, IoT, Linux

1.Introduction

In recent years the fever of wireless communication and data engineering has escalated quickly, brining to life all sorts of solutions to retrieve and compute data form various devices, vehicles, city areas etc... Nowadays we want to know what is happening in the other part of the globe, what will be the weather forecast for the next week, set the temperature of our home or check our car´s battery percentage remotely from our phone. This same phenomenon is happening for the companies, which want to know exactly the status, location and important values of their car or vehicle fleets.

Owasys is a company created in 2002 by a group of professionals coming from Ericsson (Sony company) which during the last 21 years has led the development and launching of advanced wireless devices. It designs and manufactures embedded Linux devices within the M2M (Machine to Machine) market. Many of Owasys clients use their devices to monitor and control large fleets or trucks, cars, boats or any kind of vehicle, and most of them make use of the CAN protocol. As explained before, these devices only offer a raw CAN interface in order to be used, and the final client has to develop a new or implement an existing platform.

In the need of creating the first iteration of a functional platform within the company, I started an internship and this project with Owasys. During this internship several approaches have been discussed and the desired one has been designed and implemented. The main given guidelines are that the programming language must be something more modern than the widely used C/C++ within the company, and that I would have to implement open-source libraries and modules. It is also important that the developed code and modules fit into the devices small hard disk, so the code must be written efficiently.

Throughout this document the whole process of the decision making, code writing, testbench design and result analysis will be held and described

2.Background

Automotive electronics have come a long way since the first electronic components were implemented in cars over half a century ago. In the early days, these components were mainly used for basic functions such as lighting, ignition, and radio reception. However, with the rise of computer technology, the role of electronics in automobiles has expanded significantly. Today, electronic components play a critical role in vehicle safety, performance, and efficiency.

One of the key technologies that has enabled the widespread use of electronics in cars is the Controller Area Network (CAN) protocol. CAN is a communication protocol that allows various Electronic Control Units (ECUs) in a vehicle to communicate with each other. This allows for better coordination between the different systems in the car, leading to improved safety and performance.

In addition to CAN, another important technology in modern vehicles is On-Board Diagnostics (OBD). OBD is a system that provides self-diagnostic and reporting capabilities to the vehicle owner or technician. There are two main types of OBD protocols: OBD-I and OBD-II. OBD-II is the most commonly used protocol in modern vehicles and provides a standardized way for mechanics and technicians to diagnose and repair problems with the vehicle.

The integration of CAN and OBD protocols has enabled a wide range of advanced features in modern vehicles. For example, many vehicles now come equipped with advanced driver assistance systems (ADAS), such as lane departure warning, blind spot detection, and adaptive cruise control. These systems rely on the integration of multiple sensors and ECUs, which communicate with each other through the CAN protocol.

Despite the many benefits of these technologies, there are also some challenges associated with their implementation. For example, ensuring compatibility between different ECUs and systems can be a complex task, and there is a risk of security vulnerabilities if proper measures are not taken to secure the communication channels. Nevertheless, with proper implementation and management, CAN and OBD technologies have the potential to continue transforming the automotive industry in the years to come.

2.1. Controller Area Network

The Controller Area Network (CAN) protocol was developed in the early 1980s by engineers at Bosch who were seeking a new serial bus system for passenger cars. Existing protocols did not meet the requirements of automotive engineers, so Uwe Kiencke began developing CAN in 1983. Mercedes-Benz and Intel were involved in the specification phase, and the protocol was named "Controller Area Network" by Professor Dr. Wolfhard Lawrenz. CAN was introduced at the SAE congress in 1986 and featured a non-destructive arbitration mechanism and error detection mechanisms.

In 1987, Intel and Philips Semiconductors introduced the first CAN controller chips, marking the realization of the CAN protocol. The FullCAN concept by Intel required less CPU load but had limitations in received frames, while the BasicCAN implementation by Philips required less silicon. Today's CAN controllers incorporate a mixture of both concepts.

The Bosch CAN specification was submitted for international standardization in the early 1990s, resulting in the publication of the ISO 11898 standard in 1993. The standard defined the CAN protocol and a physical layer for bit-rates up to 1 Mbit/s. However, the published specifications contained errors and were incomplete, leading Bosch to ensure CAN chips comply with their reference model.

Revised CAN specifications were standardized, including ISO 11898-1 for the CAN data link layer, ISO 11898-2 for the non-fault-tolerant physical layer, and ISO 11898-3 for the fault-tolerant physical layer. Application profiles based on the SAE J1939 network approach, such as ISO 11992 for trucks and ISO 11783 for agriculture and forestry machines, were also specified.

In 1992, the 'CAN in Automation' (CiA) [1] international users and manufacturers group was founded to promote CAN and enhance its technical development. The CiA published recommendations for using ISO 11898 compliant CAN transceivers and developed the CAN Application Layer (CAL) or 'Green Book'. The international CAN Conference (iCC) was initiated in 1994 for knowledge exchange. The Esprit project Aspic led to the development of CANopen [2], a CAN-based profile for internal networking of production cells. CANopen became the most important standardized embedded network in Europe.

Automotive manufacturers like Mercedes-Benz and BMW started implementing CAN in their cars, initially for engine management and later for body electronics. Devicenet and CANopen emerged as two standardized application layers for industrial automation, with Devicenet becoming popular in the US and CANopen in Europe. CANopen's flexibility and configurability made it suitable for various applications, and it became internationally standardized.

In the early 2000s, the TTCAN protocol was defined by an ISO task force for time-triggered transmission of CAN frames. This extension enabled time-equidistant transmission and closed-loop control, but its adoption has been limited. Several proprietary CAN-based safety protocols, including Safetybus p and CANopen-Safety, were developed.

The capabilities of the protocol were further expanded in 2012 with the introduction of CAN with Flexible Data-Rate (CAN FD). greater data exchanges between ECUs were made possible by CAN FD's [3] faster data transmission rate and greater data payload. This breakthrough allowed the creation of sophisticated applications for automotive systems, such as real-time control algorithms, high-resolution sensor data, and more comprehensive diagnostic capabilities.

The de facto standard for automobile communications is still CAN today. It is utilized in many different automotive systems, such as engine management, chassis control, infotainment, and driver assistance systems, and has been enthusiastically embraced by automakers all over the world. The protocol has cemented its position as a foundational technology in contemporary automobiles thanks to its dependability, adaptability, and interoperability with diagnostic systems like On-Board Diagnostics (OBD)[4].

CAN operates on the Physical and Data Link layers of the OSI model [5]:

- **Physical Layer (11898-2):**

The physical layer of CAN is responsible for transmitting and receiving electrical signals over the physical medium. It defines the electrical characteristics, such as voltage levels, signal timing, and connector pin assignments.

  - **Physical Medium:** CAN typically uses a twisted pair of wires, known as the CAN bus, to transmit data. The bus consists of two lines: CAN High (CANH) and CAN Low (CANL). These lines carry differential signals, which help in noise immunity.
  - **Signalling:** CAN uses a non-return-to-zero (NRZ) bit encoding scheme. The dominant bit is represented by a logical zero (CANH > CANL voltage differential), while a recessive bit is represented by a logical one (CANH = CANL voltage levels).
  - **Bit Timing:** CAN defines the rules for synchronization between transmitting and receiving nodes. It uses bit stuffing to maintain synchronization and to ensure that a sufficient number of edges are present in the bit stream.

- **Data Link Layer (ISO 11898-1):**

The data link layer of CAN manages the transmission and reception of data frames and it provides mechanisms for frame identification, error detection, and error handling.

  - **Frame Structure**: CAN frames consist of several fields, including the arbitration field, control field, data field, and CRC field.
    1. **SOF (Start of Frame):** The Start of Frame is a dominant bit (0) that signals the beginning of a CAN frame. It indicates that a CAN node is initiating communication.
    2. **ID (Identifier):** The ID is the frame identifier and determines the priority of the message. Lower ID values have higher priority, allowing critical messages to take preference over lower-priority ones.
    3. **RTR (Remote Transmission Request):** The Remote Transmission Request bit indicates whether a node is sending data (data frame) or requesting dedicated data from another node (remote frame). A dominant bit (0) means it is a data frame, while a recessive bit (1) indicates it is a remote frame.
    4. **Control**: The Control field contains two important bits:
       a. IDE (Identifier Extension Bit): For 11-bit identifiers, the IDE bit is dominant (0). For 29-bit identifiers, it is recessive (1). It distinguishes between standard and extended frame formats.
       b. DLC (Data Length Code): DLC is a 4-bit field that specifies the number of data bytes (0 to 8) in the frame.
    5. **Data:** The Data field contains the actual payload of the CAN frame. It consists of the data bytes that carry information. Each byte can represent a signal or multiple signals that can be extracted and decoded to obtain meaningful information.
    6. **CRC (Cyclic Redundancy Check):** The Cyclic Redundancy Check is a field used for error detection. It contains a checksum calculated based on the frame data. The receiving node performs the same calculation and compares the result with the received CRC to check for data integrity.

7. **ACK (Acknowledgment):** The ACK slot indicates whether the node has acknowledged and received the data correctly. During transmission, the transmitting node monitors the ACK slot. If it detects a recessive bit (1), indicating successful reception, it knows that the message was received without errors.
8. **EOF (End of Frame):** The End of Frame marks the conclusion of the CAN frame. It is a recessive bit (1) that follows the ACK slot and allows the bus to return to its idle state.

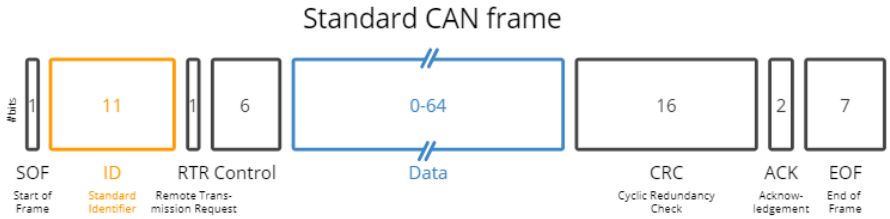


Figure 1: CAN frame structure. [5]

- **Frame Transmission:** CAN uses a non-destructive bitwise arbitration mechanism. Nodes on the bus monitor the bus during transmission and compare the dominant/recessive levels. The node transmitting a dominant bit while another node transmits a recessive bit detects the collision and stops transmitting, allowing the higher-priority message to continue.

- **Error Detection and Handling:** CAN implements error detection mechanisms to ensure data integrity. As explained before, it uses a bit-wise acknowledgement mechanism and cyclic redundancy check (CRC) to detect transmission errors. If an error is detected, the transmitting node can retransmit the frame.

- **Bus Access:** CAN employs a priority-based bus access mechanism, where messages with lower identifiers gain bus access before messages with higher identifiers. This enables the transmission of critical messages with minimal delay.

These are the most used CAN protocol implementations:

- **CAN 2.0:** The Computer Area Network (CAN) 2.0 standard, also known as CAN 2.0A and CAN 2.0B, was the CAN protocol's first iteration. Its goal was to create a dependable and effective mode of communication for electronic equipment used in industrial and automotive settings. While CAN 2.0B introduced the use of 29-bit identifiers for extended addressing, CAN 2.0A allowed 11-bit identifiers.
- **CAN FD:** The CAN FD (Flexible Data-Rate) protocol expanded the CAN protocol's capabilities by enabling faster data transmission rates and larger data payloads than CAN 2.0. Its goal was to answer the growing need for more bandwidth and flexible communication in modern industrial and automotive systems. Both traditional CAN frames and improved FD frames were supported by CAN FD.
- **CANopen:** Developed on top of the CAN physical layer, CANopen acts as an upper-layer communication protocol. It established a standardized set of communication profiles and application layer capabilities, enabling distributed control systems' interoperability and plug-and-play capability.

Medical technology, industrial automation, and numerous more industries all make substantial use of CANopen.

- **J1939:** J1939 is an advanced protocol that extends the capabilities of the CAN physical layer. It was created especially for the heavy-duty commercial vehicle sector, including buses and trucks. J1939's goal is to give these vehicles' electronic systems a standardized communication protocol that will enable interoperability and make it easier to perform diagnostic and control tasks.
- **DeviceNet** is a higher-layer protocol that is based on CAN. It specializes in industrial automation and offers a network for tying together and managing industrial equipment. The purpose of DeviceNet is to make the integration of devices simpler and to enable seamless communication between devices, including sensors, actuators, and programmable logic controllers (PLCs).
- **CANopen Safety:** Functional safety applications are the focus of CANopen Safety, an addition to the CANopen protocol. To ensure compliance with safety standards like IEC 61508 and ISO 13849, it offers additional safety-oriented features and communication systems. The goal of CANopen Safety is to make it possible to use the CAN protocol to create safety-critical systems.
- **CAN XL:** CAN XL, also known as CAN with eXtended Link, is an upcoming standard that further enhances the capabilities of CAN FD. It aims to support even higher data rates and larger payloads, enabling more demanding applications in automotive and industrial domains. The objective of CAN XL is to provide a scalable and future-proof solution for evolving communication requirements.

Overall, the CAN protocol provides a robust and efficient means of communication, particularly suited for real-time applications in challenging environments. It combines a well-defined physical layer for electrical signaling with a data link layer that manages frame transmission, arbitration, and error detection. This combination allows for reliable and deterministic communication between nodes connected to the CAN bus.

2.2. On-Board Diagnostics

As explained before, CAN protocol only establishes as a communication standard for Physical and Data Link layers of the OSI model. So in order to establish a connection with the ECU and be able to retrieve data from it a higher level protocol is needed. For this project OBD has been selected due to its large compatibility and establishment within the car manufacturer industry.

The history of OBD can be traced back to the early 1980s when automotive manufacturers recognized the importance of implementing standardized diagnostic systems in vehicles. At that time, vehicle emissions and engine control systems were rapidly evolving, prompting the need for efficient monitoring and troubleshooting mechanisms.

Initially, OBD systems were relatively basic, consisting of rudimentary sensors and limited diagnostic capabilities. However, as technology progressed, so did the sophistication of OBD systems. The real breakthrough came with the introduction of the first generation of standardized OBD, known as OBD-I.

OBD-I systems were implemented in the late 1980s and early 1990s. They provided manufacturers with the ability to monitor certain emission-related components and detect malfunctions, thus aiding in emission

control and regulatory compliance. However, the diagnostic capabilities of OBD-I were limited, and the data retrieval process was often complex and manufacturer-specific.

Recognizing the need for improved diagnostic systems, regulatory bodies and automotive industry stakeholders collaborated to establish a unified and more advanced standard. This led to the introduction of the second generation of OBD, known as OBD-II. OBD-II was a significant milestone in automotive diagnostics. It was standardized across all vehicles sold in the United States from 1996 onwards and became a global benchmark for diagnostic systems. OBD-II systems featured enhanced diagnostic capabilities, a standardized diagnostic connector, and the ability to monitor a broader range of vehicle systems and components.

The establishment of OBD-II in the market brought numerous benefits for vehicle owners, technicians, and regulators. It enabled easier and more accurate diagnosis of vehicle issues, streamlined emission testing and compliance, and facilitated the development of aftermarket diagnostic tools and software.

Today, OBD continues to evolve with the advancement of automotive technology. The current generation, OBD-II, has become an integral part of vehicle maintenance and diagnostics. It provides valuable insights into the health and performance of vehicles, allowing for efficient repairs, improved fuel efficiency, and reduced emissions.

In the following points, the most important features of the the OBD-II protocol will be analyzed and explained [4]:

- **Connector and Pinout:**
  The OBD-II protocol utilizes a standardized diagnostic connector, which is typically located within the vehicle's cabin for easy access. The connector follows a specific pinout configuration to ensure compatibility across different vehicle manufacturers and models. The most common type of OBD-II connector is the 16-pin Data Link Connector (DLC), which is often located under the dashboard or steering column.

  The pinout configuration of the OBD-II connector consists of several important pins, including power and ground pins for supplying electrical power to the diagnostic interface. But the most important pins for us are the ones dedicated for the CAN bus (Controller Area Network) or other protocols, which enable data exchange between the vehicle's onboard systems and external diagnostic hardware (Owasys owa4x in this case). Is it possible to see them in the next figure, pins 6 and 14:

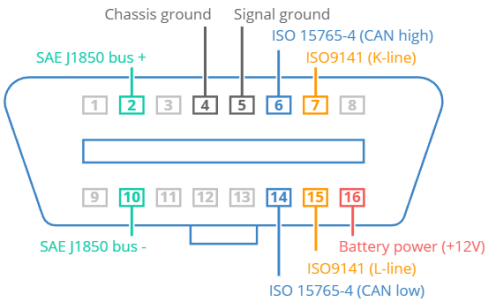


Figure 2: OBD connector pinout. [5]

- **Layer in the OSI Model:**
  In terms of the OSI (Open Systems Interconnection) model, the OBD-II protocol primarily operates in the application layer. The application layer is responsible for providing a standardized interface for communication and data exchange between the vehicle's electronic control modules and diagnostic devices. OBD-II is standardized as ISO 15765-4.
  The lower layers of the OSI model, including the physical and data link layers, are abstracted by the OBD-II protocol and the underlying communication protocols such as CAN, ISO 11898-1 (Data Link Layer) and ISO 11898-2 (Physical Layer) mentioned before. These protocols handle the actual transmission of data between the vehicle's internal systems and the diagnostic tool.

- **OBD-II Frame Structure:**
  The OBD-II protocol employs a specific frame structure for transmitting diagnostic information between the vehicle and the diagnostic tool. The frame structure consists of several components, including:
  - **Identifier:** In OBD-II messages, the identifier consists of an 11-bit code used to differentiate between request messages (ID 7DF) and response messages (ID 7E8 to 7EF). The main engine or ECU responds with ID 7E8 in most of the cases.
  - **Length:** This field indicates the length of the remaining data in terms of the number of bytes.
  - **Mode:** In requests, the mode field ranges from 01 to 0A. For our purposes the Mode 1 will be used, known as Current Data, which enables access to real-time vehicle speed, RPM, and other parameters. Other modes are used to retrieve or clear stored diagnostic trouble codes (DTCs) and display freeze frame data for example.
  - **PID:** Each mode has a list of standard OBD2 PIDs associated with it. This identifier refers to the memory address where the desired data is located or to a specific routine or request to be run. Most of the PIDs are proprietary of car manufacturers but there are some PIDs available defined by the standards [7]
  - **A, B, C, D:** These represent the data bytes in hexadecimal format, this means that the data is not readable as it arrives, so we need to use DBCs (CAN DataBase). This topic will be held in the next chapter

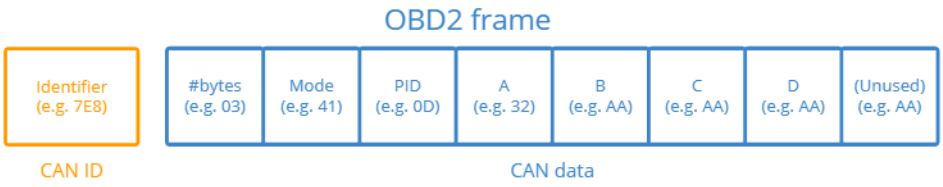


Figure 3: OBD2 frame structure. [5]

2.3. CAN DataBases (DBCs)

As explained before, the data retrieved from the OBD-II protocol comes in raw hexadecimal so it is necessary to implement a DBC file. These files are a structured database that contains information about the messages and signals exchanged between different electronic control units (ECUs) or nodes through a CAN network.

The main features and characteristics of DBC files include [6]:

- **Message Definitions:** DBC files define the messages transmitted over the CAN bus. They specify the message identifiers, message names, and associated attributes. Each message represents a specific command or information exchange between ECUs.
- **Signal Definitions:** DBC files define the signals contained within each message. Signals represent individual data elements within a message, such as temperature, speed, or engine RPM. DBC files describe signal names, scaling factors, units, and bit positions within the message frame.
- **Network Node Definitions:** DBC files include information about the network nodes or ECUs participating in the communication. They specify the node names, node addresses, and the messages transmitted or received by each node. This allows for proper routing and identification of messages within the network.
- **Signal Multiplexing:** DBC files support signal multiplexing, which allows multiple signals to be transmitted within a single message. This is achieved by assigning different multiplexor values to different signals, enabling the receiver to extract the desired signal based on the provided multiplexor value.
- **Signal Encoding and Scaling:** DBC files define the encoding format and scaling factors for signals. They specify whether a signal is represented in binary, integer, or floating-point format. Scaling factors allow the conversion of raw signal values into meaningful engineering units, for example, converting a voltage reading to temperature in degrees Celsius.
- **Attribute Definitions:** DBC files can include additional attributes for messages, signals, and network nodes. These attributes provide additional information or metadata about the data being transmitted, such as signal validity, signal resolution, or the intended use of a particular message.

In the next figure an example of a message and signal definition in a DBC file is shown:

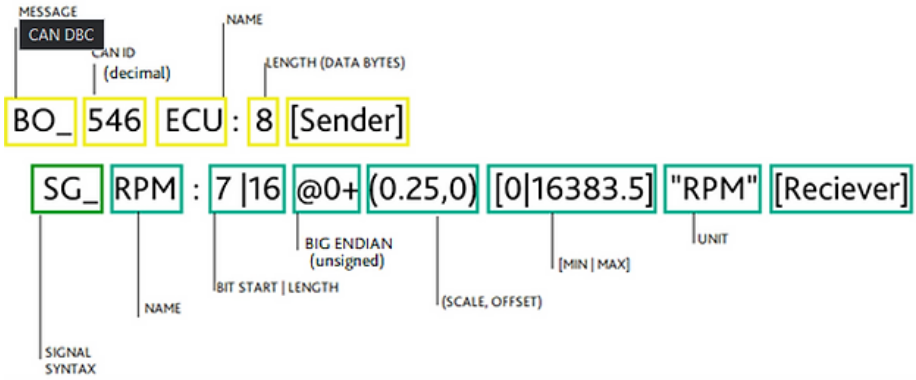


Figure 4: DBC file entry example.

Overall, DBC files serve as a standardized and structured format for describing the communication parameters and data exchange within a CAN network. They enable efficient interpretation and analysis of CAN bus data during development, testing, diagnostics, and system integration processes in the automotive industry.

It is worth pointing out that during this project all the entries of the available DBC files have been translated and implemented in python code. The implementation of different python modules in order to use DBC files could be possible, but in order to maintain a lower disk usage and achieve a simpler implementation this approach has been used.

3.Scope

Owasys is a design and manufacturing engineering company within the M2M (Machine to Machine) market which creates embedded Linux devices. It was created in 2002 by a group of professionals coming from Ericsson from Sony and during the last 21 years it has led the development and launching of advances wireless devices.

As explain during the Abstract, these devices offer a raw interface for CAN communication but the client itself has to create or implement another existing platform to correctly communicate through it. In the need of offering something to the client, the project of creating the first working prototype for this purpose was assigned to me. The company had very clear that their programming language diversity had to expand having a preference for Python language, which is very flexible and easy to learn.

It is worth pointing out that no requirements were asked for this project in terms of performance and stress resistance. The requirements asked for this project:

- Correct CAN communication between device and ECU
- Compatibility with generic commercial cars
- Lightweight solution in terms of disk usage

4.Objectives

The objective of this project is to design and implement a functional communication system that establishes a connection with commercial cars using the CAN (Controller Area Network) and OBD (OnBoard Diagnostics) protocols. The project aims to enable data retrieval and monitoring from the car's onboard systems in real time. The primary focus will be on the Owa4x hardware platform developed by Owasys.

In order to achieve the main goal, the following secondary objectives have been defined:

- Analysis of CAN (Controller Area Network) and OBD (OnBoard Diagnostics) protocols will be held. What libraries and software will be implemented in the Owa4x will be decided.
- In order to simulate the environment a test-bench will be designed, using the available hardware from Owasys and defining the different connections and metrics to be measured.
- Implementation of a functional CAN and OBD communication through the Owa4x alongside a web interface in order to monitor data in real time.
- Finally, an analysis of the obtained results will be held, as well as an explanation of the conclusions.

5.Benefits

5.1. Technical Benefits

- Enhanced Protocol Understanding: Through the analysis of the CAN (Controller Area Network) and OBD (OnBoard Diagnostics) protocols, the project provides a comprehensive understanding of their intricacies and specifications. This understanding enables the development of robust and efficient communication systems that can be effectively used by Owasys in order to communicate with commercial cars, ensuring seamless data exchange and interoperability.

- Optimized Software and Library Selection: By evaluating various software and libraries, the project helps in making informed decisions regarding the selection of appropriate tools for the implementation of the Owa4x platform.

5.2. Economic Benefits

- Cost-effective Maintenance and Diagnostics: By establishing functional communication with commercial cars through CAN and OBD protocols, the project enables efficient monitoring and diagnostics. This should lead to cost savings in terms of maintenance and troubleshooting processes within the company.

- Optimized Resource Utilization: The project's analysis and evaluation of the implemented communication systems provide insights into resource utilization, such as CPU usage, memory consumption, and power requirements. This information aids in optimizing the resource allocation for the Owa4x platform, resulting in improved efficiency, reduced operational costs, and extended device battery life.

5.3. Social Benefits

- Enhanced Safety and Performance: The functional communication with commercial cars through CAN and OBD protocols contributes to enhanced safety and performance in the automotive industry. Real-time monitoring and data exchange enable proactive identification of potential issues, allowing for timely maintenance and interventions. This makes it possible to implement safer and more efficient systems and components in the future.

- Facilitated Data-driven Decision Making: The implemented communication systems, coupled with a web interface for real-time data monitoring, provide valuable insights into vehicle parameters, performance metrics, and diagnostic information. This enables data-driven decision making for car manufacturers, fleet operators, and service providers. Informed decisions regarding maintenance schedules, optimization of vehicle performance, and resource allocation can be made, leading to improved operational efficiency and cost-effectiveness.

6.Alternative Analysis

- Programming Language:
  The choice of the programming language to be implemented during the project is probably the most important one. Depending on this choice the workflow of the project and the available modules or resources will vary from one to another, so it is important to choose wisely. Is it also important taking into account which languages are mostly used by the company or which new languages are trying to implement, this way the integration of this new project into the company will be performed more easily.
  Alternatives:
    - **C/C++ [8]:** It is one of the most if not the most implemented programming language. Created in 1970 by Dennis Ritchie is a general-purpose language implemented in operating systems, protocol stacks or Software development, which has then evolved to an object-oriented programming language (C++) in 1979. It offers a simple and portable solution which is rich in libraries and support from the community. Moreover, Owasys implements C/C++ in most of its Software developments, so its integration would be easy. On the other hand, the company aims to implement more modern languages in the short term.
    - **Python [9]:** Python is making its own place in the modern programming scene. This interpreted, open-source and object-oriented programming language offers a very light and fast solution for almost any machine. It implements a high-level built-in data structure with dynamic typing and binding, which makes it very desirable for systems interconnection. Due to its rise in market usage, Owasys is planning to implement Python in the short term.
    - **Node.js [10]:** It is a cross-platform and open-source server capable of running in different operating systems. It implements the V8 JavaScript engine offering command line tools and server-side scripting, is mainly used in web application development. It offers a good performance with great scalability and easiness to learn the language. On the other hand, there is no great community support in terms of libraries,

  Considering the three alternatives described above, the next criteria will be used to choose one of them:

    - **Development and Implementation time (40%):** This project is intended to be the first iteration of the company to bring the clients a platform for CAN communication so the time to develop and implement it is key. A fast learning and easy to put in practice programming language will be more valuable for the project.
    - **Running speed and memory usage (30%):** Considering the limited computational capacity and storage of the Hardware, which will be described in the section x, this is an important criterion for the programming language choice. If the language is interpreted or compiled and depending on their variable handling strategies, the running speed and the memory usage will vary, resulting in a performance loss.
    - **Future usage perspective (30%):** Although something already works, a prospere company should always have an open-minded perspective and see what other technologies can offer in order to improve.

	Development and Implementation time (40%)	Speed and memory usage (30%)	Future usage perspective (30%)	Total
C/C++	6	9	4	6.3
Python	9	6	9	8.1
Node.Js	9	5	7	7.2

Table 1: Programming language alternative analysis.

Despite the fact that C/C++ is mainly used within Owasys and its integration would be easier, the fast-learning curve and easiness to implement Python alongside the desire to explore new possibilities was more important in this project. It is also important taking into account that it is being developed by myself and then it will be integrated into the team's workflow. Thus, Python will be the selected programming language for the project.

7.Risk analysis

In this section, we will delve into the potential challenges that may arise throughout the project's duration. These challenges have the potential to cause delays in the project's timeline. Therefore, it is imperative to conduct a thorough analysis of these risks in order to proactively address them and prepare for worst-case scenarios.

7.1. Delays on the project

A typical risk encountered in project development involves the potential for timelines to be disrupted. When one task experiences delays, it can have a ripple effect on subsequent tasks, ultimately resulting in an extended project timeline. It is important to recognize that some tasks may be completed ahead of schedule, which can help offset possible delays.

Probability of happening: 50%
Overall impact: 50%

Mitigation measures: To counter this risk, a flexible and realistic planning approach will be adopted. Extra time will be allocated to tasks that are more likely to experience delays, allowing for better time management and adjustment of resources as necessary.

7.2. Security Risks

When establishing communication with commercial cars, security risks need to be addressed to protect against unauthorized access or potential vulnerabilities in the protocols. Failure to address these security risks could compromise the integrity of the system, the data exchanged and the life of the vehicle's occupant..

Probability of happening: 30%
Overall impact: 90%

Mitigation measures: Robust security measures, such as authentication and encryption, will be implemented to safeguard the communication channel and ensure data confidentiality and integrity. Adherence to established security best practices and standards, as well as continuous monitoring for potential vulnerabilities, will be prioritized throughout the development process.

7.3. Overpass the budget

The budget is structured with an assumption of low likelihood regarding hardware failures or delays. However, unforeseen circumstances, such as hardware issues or delays, could potentially lead to budget overruns. While the hardware in use is not particularly costly, it's worth noting that human resource expenses might increase due to planning delays, exceeding the initial budget estimates.

Probability of happening: 10%
Overall Impact: 70%

Mitigation measures: In order to maintain the agreed budget, the used hardware must be physically well protected and if more security is needed an insurance can be purchased for the devices, this way in case of failure the impact would not be so big. As for the employees, more hours might be needed if the experience in the field is not enough, but to contrarrest this a more specialized person can be hired.

7.4. Integration Challenges

Integrating the CAN and OBD protocols with the Python program poses a risk of encountering integration challenges. Ensuring correct communication between the software components and the protocols requires careful coordination and implementation.

Probability of happening: 30%
Overall impact: 50%

Mitigation measures: Thorough testing and verification will be performed to validate the integration of the CAN and OBD protocols with the Python program. Protocols and libraries with established compatibility and a strong community support will be used to minimize integration challenges.

7.5. Comparison

In the next table a comparison between the exposed risks is shown in order to compare them visually. The worst-case scenarios are when the risk has an important impact in the project and at the same time has a high probability to happen.

		IMPACT				
		10	30	50	70	90
	10				R3	
	30			R4		R2
PROBABILITY	50			R1		
	70					
	90					

Table 2: Project`s risk comparison.

8.Solution design and implementation

As explained in the Scope section, the main goal of this project is to design and implement a functional CAN communication through the OBD interface. In order to achieve this goal, it is necessary to fit the solution and implementation to the requirements of the project. The next section will explain this process.

8.1. Owa450 specifications

The Owa450 is the most important Hardware device in the project and it establishes the basis for the rest of the project. It is a powerful IoT gateway based on Linux which is capable of processing data coming from wired and wireless sensors, devices and vehicles. Usually, these devices are used as monitoring tools for company fleets of trucks, excavators or any kind of vehicle that needs to be monitored or managed remotely. Owasys offers a range of devices adapted to many scenarios depending on the needs for computational power and power consumption.

In this case, as Python was selected for the programming language of this project, the Owa450 has been the choice due to its computational power. But at the same time, it is worth mentioning that it consumes 47 mA of current on normal usage.

- Main specifications of the Owa450 [11]:
- ARM Cortex A8 800MHz Processor
- 512MB DDR3 RAM
- 1 GB NAND Flash
- Linux Kernel v4.19.94
- Debian 10 Distribution File System
- 2 CAN interfaces
- Many wireless protocols: GNSS/WiFi/Bluetooth/LTE/UMTS/HSPA+/GSM/GPRS…

Figure 5: owa450 device.

8.2. Owasys Developers Board

The CAN interfaces available in the Owa450 are in the most basic form and it is not the best approach to connect them directly to the OBD interface of the car. In order to achieve a more elegant and professional result, Owasys offers a Developers Board with all the available connections and digital and analog inputs and outputs.

In this project it will be used to connect the raw CAN interfaces from the device to the exact pins needed in a generic OBD male connector. This way, the male connector can be directly plugged to the car and the device receives only the necessary data through the CAN pins, achieving an elegant and safe setup.

The schematic of the board is shown in Annex1. The physical description of the board is shown in the next figure:

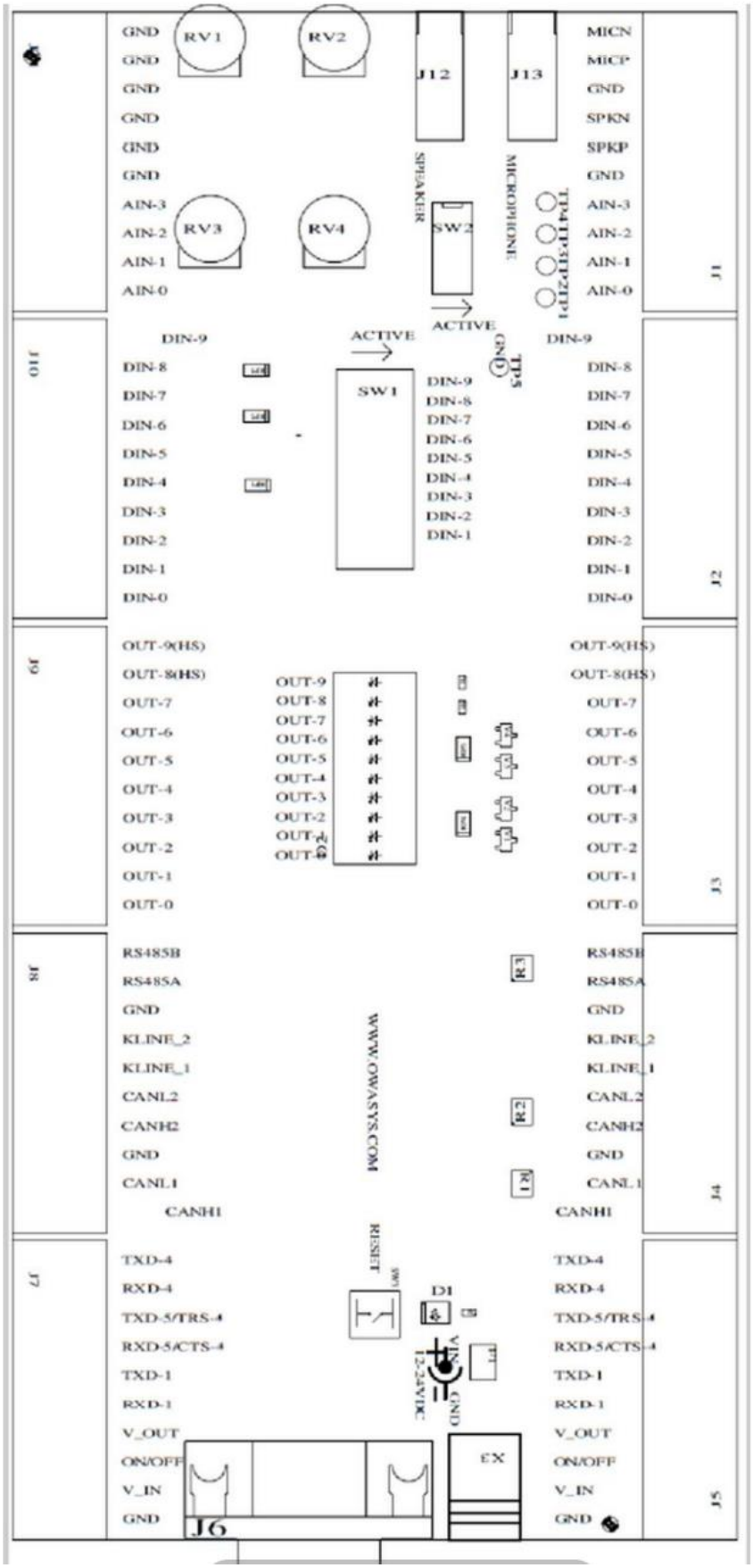


Figure 6: Owasys developers board physical description.

8.3. OBD connector

The OBD connector was one of the most basic pieces of the setup, a generic male connector was selected, shown in the next figure:

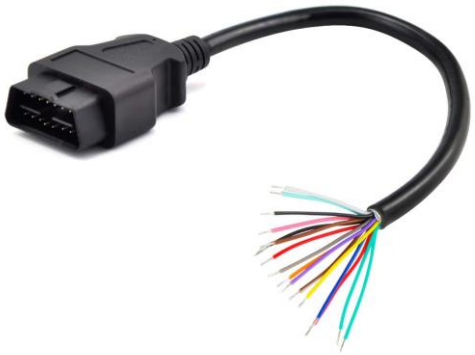


Figure 7: Generic OBD male connector.

In order to connect the cables to the Developers Board correctly, checking the OBDs pinout is necessary:

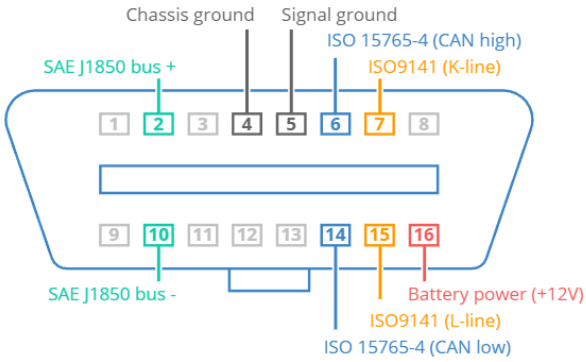


Figure 8: OBD connector pinout. [5]

As this project is intended to work in commercial cars, the ISO 15765-4 protocol pins have been used. If the usage would be for trucks for example, the SAE J1850 protocol is used, but the only change to be done would be to reconnect the desired cables from the connector to the board. Then, the pins 6 and 14 connect with the corresponding CAN High and Low from the Owa450 and the pins 16 and 5 are connected to the Power Supply and GND connections of the board. This way, the car itself will power the Owa device.

8.4. Setup graphic representation

In this section the graphic representation of the implemented setup will be presented and described. First is needed to find the OBD2 interface in the desired car to be tested and the generic OBD male connector is attached. As mentioned before, in order to achieve a more elegant and secure setup Owasys Developers Board is used as a gateway for the owa450.

Once the data is requested and received through CAN communication, the owa device processes it with the later explained Python code and sends it through the Redis network. Thanks to Owasys, a predefined encrypted pipe is used on this project to connect directly to their servers through a secure communication line, this way the data retrieved from the car will be safe sending it through 4g/LTE or WiFi hotspot.

In the Figure 10 the graphic representation of the setup is shown:

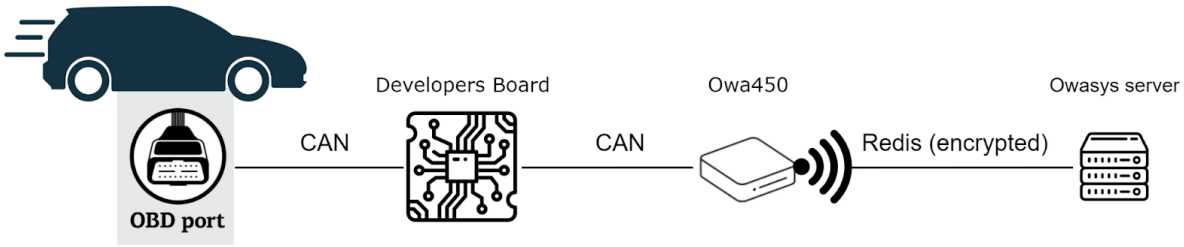


Figure 9: Graphic representation of the implemented setup

8.5. Employed packages and libraries

Keeping the open-source and community support mindset of the project, the implemented libraries are all free of use rights. During this section installed Debian packages and Python modules will be described and presented.

- **Debian packages:**
  For the operative system, Debian in this case, a preloaded module used by Owasys has been employed in order to achieve the maximum memory size reduction. The basic packages in order to achieve a normal running of the operative system have been installed alongside Python`s package and its dependencies, pip being the most important one for the project in order to install Python modules. This way a decent amount of space has been saved keeping the necessary packages and modules to develop the project.

- **Python libraries:**
  Python`s community of developers offers a great variety of libraries for any kind of use, and this project will take advantage of that. Many libraries have been used but, in this section, the main 3 are going to be described.

    - **Python-can:** This library provides a Python environment for Controller Area Network protocol. Through this library the can messages will be created, sent and received, creating the perfect bus communication for the purpose of this project.
    - **Redis:** This open-source library is an in-memory data structure store used as a database, message broker or streaming engine. It provides all variety of data structures and queries, being the perfect environment to use in order to send the information to the previously mentioned server. This way, the clients will be able to access all the data remotely and monitor its vehicles.
    - **Shared Memory reader and writer (shmreader):** This is the only library developed inside Owasys and it enables writing and reading from the shared memory of the Linux system. This way any other application or package that the client wants to install is able to read the data directly from Linux shared memory, without the need of creating a pipeline. The code is available in the Annex 3.

The secondary packages or dependencies and the version of the Python libraries are shown in the Annex 2

8.6. Code description

During this section the written code will be shown and the corresponding explanations will be given. Many functions from a variety of packages are implemented so only the necessary ones will be explained. In the next figure the imported python libraries and the initialization of the Redis pipe are shown:

```
1   #!/usr/bin/python3
2
3   from cmath import tan
4   import redis
5   import can
6   import time
7   import os
8   import queue
9   import shmreader
10  from threading import Thread
11
12
13  #Redis pipe initialization
14  r = redis.Redis()
15  r.ping()
16
```

Figure 10: Python packages import and Redis pipe start.

The program implements two dictionaries, the first one is used to translate the PID number (hexadecimal) in the receiving message to the function to be run and its name, while the other one Translates each variable name to the variable type needed during the Linux shared memory writing. In the next figure a section of the first dictionary is shown:

```
17  #Dictionay for PID-function number traduction
18  dyctionary = {
19          0x00 : ["PIDs_SUPPORTED_[1-20]", 1],
20          0x01 : ["MONITOR_STATUS_DTCLEAR", 2],
21          0x02 : ["FREZEE_DTC ", 3],
22          0x03 : ["FUEL_SYSTEM_STATUS ", 4],
23          0x04 : ["CALCULATED_ENGINE_LOAD", 5],
24          0x05 : ["ENGINE_COOLANT_TEMP", 6],
25          0x06 : ["SHORT_TERM_FUEL_TRIM_1", 7],
26          0x07 : ["LONG_TERM_FUEL_TRIM_1", 7],
27          0x08 : ["SHORT_TERM_FUEL_TRIM_2", 7],
28          0x09 : ["LONG_TERM_FUEL_TRIM_2 ", 7],
29          0x0A : ["FUEL_PRESSURE", 8],
30          0x0B : ["INTAKE_MAINFOLD_ABS_PRESSURE", 9],
31          0x0C : ["ENGINE_RPM", 10],
32          0x0D : ["VEHICLE_SPEED", 11],
33          0x0E : ["TIMING_ADVANCE", 12],
34          0x0F : ["INTAKE_AIR_TEMP ", 6],
35          0x10 : ["MAF_SENSOR", 13],
36          0x11 : ["THROTTLE POSITION", 5],
```

Figure 11: First dictionary.

Figure 12 shows a piece of the second dictionary:

```
# Dictionary
# key = shared memory tag (MAX length 32)
# value = shared memory value C type INT = 0, FLOAT = 1, BOOL = 2, LONG_LONG = 3, DOUBLE = 4
variables_dict= {
        "CALCULATED_ENGINE_LOAD" : 1,
        "ENGINE_COOLANT_TEMP" : 0,
        "SHORT_TERM_FUEL_TRIM_1" : 1,
        "LONG_TERM_FUEL_TRIM_1" : 1,
        "SHORT_TERM_FUEL_TRIM_2" : 1,
        "LONG_TERM_FUEL_TRIM_2 " : 1,
        "FUEL_PRESSURE" : 0,
        "INTAKE_MAINFOLD_ABS_PRESSURE" : 0,
        "ENGINE_RPM" : 1,
        "VEHICLE_SPEED" : 0,
        "TIMING_ADVANCE" : 1,
        "INTAKE_AIR_TEMP " : 0,
        "MAF_SENSOR" : 1,
        "THROTTLE_POSITION" : 1,
        "SECONDARY_AIR_STATUS" : 0,
        "02_SENSORS" : 0,#IN 2 BANKS
        "02_SENSOR_1" : 1,
        "02_SENSOR_2" : 1,
        "02_SENSOR_3" : 1,
        "02_SENSOR_4" : 1,
        "02_SENSOR_5" : 1,
        "02_SENSOR_6" : 1,
        "02_SENSOR_7" : 1,
        "02_SENSOR_8" : 1,
        "OBD_STANDARDS" : 0,
        "02_SENSORS": 0,#IN 4 BANKS
```

Figure 12: second dictionary.

After the definition of these two libraries the initialization of the global variables, constant values and writing directories is done. At the same time, the PythonSwitch class is defined which will be used to iterate between the necessary functions whenever a specific message arrives. Its main function get() is also defined which adds the function number to a string in order to run the corresponding function. This piece of the code is shown in Figure 13:

```
data1 = data2 = data3 = data4 = data5 = 0
gName = ""
PID_REQUEST = 0x7DF
PID_REPLY = 0x7E8
outfile = open('/var/log.txt','w')
list_PID = []
engRPM = 0
vehicleSpeed = 0
throotlePos = 0
ambientTemp = 0
oilTemp = 0
coolantTemp = 0
tankLevel = 0


class PythonSwitch:
        global engRPM, vehicleSpeed, throotlePos, ambientTemp, oilTemp, coolantTemp, tankLevel
        def get(self, functionN):

                default = "Incorrect function"

                return getattr(self, 'case_' + str(functionN))()

        def case_1(self):
```

Figure 13: Global variables and PythonSwitch class definition.

- **dataN:** These variables will be used to store the raw data directly arriving from the CAN message and then be transformed to human readable data in its corresponding function.
- **gName:** String to save the name of the function/variable.
- **PID_REQUEST/REPLY:** Define the hexadecimal PID numbers in order to set or identify a CAN message as request or reply.
- **outfile:** Defines the path to save the log file for the owa450.
- **list_PID:** As it will be shown later, the very first thing to be done when the connection with the car's ECU is correct is to ask which PIDs are available to read. This way only the available PIDs will be requested and the bus will not be loaded with requests that will be denied.
- **Rest of variables:** The purpose of these variables is to save the data to then send it to the server via redis network, after they are transformed to human readable data inside its corresponding function. Is it worth pointing out that these variables have to be defined as global inside the PythonSwitch class in order to achieve the desired performance.

After defining all the variables and functions (which an example will be shown later), the CAN interface of the owa450 is started alongside the CAN bus. Then, the allocation for the Linux shared memory is done through the shmreader class using the previously shown variables dictionary. When transmitting or receiving messages a thread will be used to run in parallel with the main program. The definition of the sending function is very simple, it will be listening to the CAN bus while the program is alive and it will add to the queue the ones marked as a reply message.

This process is shown in the Figure 14:

```
1248  print('\n\rCAN connection test')
1249  print('Bringing up CAN1....')
1250
1251  # Bring up can1 interface at 500kbps
1252  try:
1253          os.system("sudo /sbin/ip link set can1 up type can bitrate 500000")
1254  except OSError:
1255          print('An error was encountered while bringing up CAN interface')
1256          print('An error was encountered while bringing up CAN interface',file = outfile)
1257          exit()
1258
1259  time.sleep(0.5)
1260  print('CAN1 interface ready')
1261
1262  #Bring up CAN bus
1263  try:
1264          bus = can.interface.Bus(channel='can1', bustype='socketcan_native')
1265  except OSError:
1266          print('An error was encountered while bringing up CAN bus')
1267          print('An error was encountered while bringing up CAN bus',file = outfile)
1268          exit()
1269
1270  print('CAN1 bus ready')
1271
1272  #Create shared memory allocation in the owa450
1273  print('Creating shared memory')
1274  hw = shmreader.ShmReader()
1275  hw.create_shm(10007,variables_dict)
1276
1277  def can_rx_task():      # Receive thread while program is running
1278          while True:
1279                  message = bus.recv()
1280                  if message.arbitration_id == PID_REPLY:
1281                          q.put(message)  # Put message into queue if it is a reply
```

Figure 14: Interface, bus and shared memory starting and sending thread definition.

Going through the sending thread, firstly it will ask which PIDs has the ECU available through the messages 0x00, 0x20, 0x40 and 0x60. Then, if one of the PIDs sent through the redis network is found, a periodic sending of the message is set for the CAN bus and it is removed from the list to avoid collisions.

```
def can_tx_task():      # Transmit thread
        print('Requesting available PIDs...')
        msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,0x00,0x00,0x00,0x00,0x00,0x00],extended_id=False)
        bus.send(msg)
        time.sleep(0.05)
        msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,0x20,0x00,0x00,0x00,0x00,0x00],extended_id=False)
        bus.send(msg)
        time.sleep(0.05)
        msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,0x40,0x00,0x00,0x00,0x00,0x00],extended_id=False)
        bus.send(msg)
        time.sleep(0.05)
        msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,0x60,0x00,0x00,0x00,0x00,0x00],extended_id=False)
        bus.send(msg)
        time.sleep(0.05)

        #Set peridic sending messages for variables sent via redis network
        if 0x05 in list_PID:
                list_PID.remove(0x05)
                msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,0x05,0x00,0x00,0x00,0x00,0x00],extended_id=False)
                bus.send_periodic(msg,300)
                time.sleep(0.05)

        if 0x0C in list_PID:
                list_PID.remove(0x0C)
                msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,0x0C,0x00,0x00,0x00,0x00,0x00],extended_id=False)
                bus.send_periodic(msg,3)
                time.sleep(0.05)

        if 0x0D in list_PID:
                list_PID.remove(0x0D)
                msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,0x0D,0x00,0x00,0x00,0x00,0x00],extended_id=False)
                bus.send_periodic(msg,3)
                time.sleep(0.05)
```

Figure 15: Receiving thread definition.

Alongside the periodicity of these messages, the variation threshold from which this new value is sent must be also set (in order to avoid unnecessary network load). These are the values for each variable:

	Periodicity (s)	Variation threshold
engRPM (0x0C)	3	100 rpm
vehicleSpeed (0x0D)	3	3 km/h
throotlePos (0x11)	3	3%
ambientTemp (0x46)	300	0.9 ºC
oilTemp (0x5C)	300	0.9 ºC
coolantTemp (0x50)	300	0.9 ºC
tankLevel (0x2F)	300	3%

Table 3: Redis variables Periodicity and variation threshold.

The advantage of using Python on this project is that it allows the clients to easily change the setting of these variables and even add new variables for their Redis network very easily.

To end with the sending thread, an infinite while is defined in which the messages that are only registered in the log file and saved into Linux shared memory are sent through the CAN bus. This process is shown in the next figure:

```
1338        while True: #Send the messages only registerend in the log file and saved into linux shared memory
1339                for x in list_PID:
1340                        msg = can.Message(arbitration_id=PID_REQUEST,data=[0x02,0x01,x,0x00,0x00,0x00,0x00,0x00],extended_id=False)
1341                        bus.send(msg)
1342                        time.sleep(0.05)
1343
1344                time.sleep(0.1)
```

Figure 16: Main loop for receiving thread.

Finally, the last variables and functions definitions alongside the main loop of the program are reached. The queue, the sending and receiving threads and the function switch are started next. In the main loop, until the program catches a keyboard interrupt (Ctrl + C) it will run an infinite loop where it gets a message from the queue if there is one and it checks if it is a reply message and if it is in the dictionary. If this check is correct, it will save the raw data to the global variables, ask the dictionary for the function/variable name and the function number and run the get() function of the switch to activate the corresponding task.

```
1347  q = queue.Queue() #Creation of the queue
1348  rx = Thread(target = can_rx_task) #Transmiting thread
1349  rx.start()
1350  tx = Thread(target = can_tx_task) #Receiving thread
1351  tx.start()
1352
1353  my_switch = PythonSwitch() #Creation of the function swithc
1354
1355  # Main loop
1356  try:
1357          while True:
1358                  while(q.empty() == True): # Wait until there is a message
1359                          pass
1360                  message = q.get() #Get message from the queue
1361
1362                  if message.arbitration_id == PID_REPLY:
1363                          PID = message.data[2]
1364                          if PID in dyctionary:
1365                                  data1 = message.data[3]
1366                                  data2 = message.data[4]
1367                                  data3 = message.data[5]
1368                                  data4 = message.data[6]
1369                                  data5 = message.data[7]
1370                                  name,functionN = dyctionary.get(PID)
1371                                  gName = name
1372                                  my_switch.get(functionN)
1373
1374
1375
1376
1377  except KeyboardInterrupt:
1378          #Catch keyboard interrupt
1379          outfile.close()         # Close logger file
1380          os.system("sudo /sbin/ip link set can1 down")
1381          print('\n\rKeyboard interrtupt')
```

Figure 17: Start of queue and threads and main loop of the program.

In order to explain every part of the code, an example of a function inside the switch will be shown. Not all the PIDs are treated the same way, when asking the list of PIDs for example, each bit must be examined to extract the data, some of the functions are specific for a variable and some, as in this example, are used for multiple variables.

```
402         def case_6(self):
403                 temp = data1 - 40
404                 prhase = "The " + gName + " is "  + str(temp) + "ºC"
405                 print(prhase)
406                 print(prhase,file = outfile)
407                 hw.update_shm_key(10007,gName,temp)
408                 if gName == "ENGINE_COOLANT_TEMP":
409                         difference = abs(coolantTemp - temp)
410                         if difference > 0.9:
411                                 coolantTemp = temp
412                                 data = "{\"ts\":" + str(time.time()) + ",values:{\"coolant.temp\":" + str(coolantTemp) + "}}"
413                                 #Send data
414                                 r.rpush("q-out", data)
415                 if gName == "AMBIENT_AIR_TEMP":
416                         difference = abs(ambientTemp - temp)
417                         if difference > 0.9:
418                                 ambientTemp = temp
419                                 data = "{\"ts\":" + str(time.time()) + ",values:{\"air.temp\":" + str(ambientTemp) + "}}"
420                                 #Send data
421                                 r.rpush("q-out", data)
422                 if gName == "ENGINE_OIL_TEMP":
423                         difference = abs(oilTemp - temp)
424                         if difference > 0.9:
425                                 oilTemp = temp
426                                 data = "{\"ts\":" + str(time.time()) + ",values:{\"oil.temp\":" + str(oilTemp) + "}}"
427                                 #Send data
428                                 r.rpush("q-out", data)
429
430
431                 return 1
```

Figure 18: Example of a function inside the switch.

In the figure above, the function to manipulate and send the data for engine coolant temperature, ambient air temperature and engine oil temperature is shown. This is possible because the raw data comes in the same format for all three variables and thus, the transformation to be done is also the same. The manipulation here is just to rest 40 to the raw data. Then, a phrase with the data is defined and saved into the log file and shared memory, and printed through the terminal to the user. The name of the variable is checked, and if the verification is correct the variation threshold is checked. In this case if the temperature change is bigger than 0.9 ºC the new temperature is updated and the string format to send through the Redis network is defined and sent.

With this the code description is finished, for further inspection the full code is in the next GitHub repository:

https://github.com/Orbesito/OwasysTFM/blob/main/obd.py

8.7. Virtual and real environment testing

During this section the results from the virtual and real environments testing will be shown. In order to achieve a better Software development time, the first iterations of the testing will be held in a virtual environment. This way, it will not be necessary to mount a real testing workbench with a car for each time a new feature is tested and it will be possible to fix small errors before arriving to the workbench.

8.7.1 Virtual testing

In order to perform the virtual testing, the only thing needed is a virtual machine with a Linux distribution running. In this case the implemented virtualization software is VMware Workstation 16 Pro in the version 16.1.0 build-17198959 and the selected Linux image is Ubuntu 20.04.6 LTS with 64-bit configuration. Once the virtual environment is set up, the python libraries shown before are installed through pip3.

After everything is installed, the *modprobe* module will be used in order to charge a virtual can module into the Linux kernel and then the virtual CAN interface added as a device and brought up through *ip link* tool from Linux. This process is shown in the next figure:

```
$ sudo modprobe vcan
$ sudo ip link add dev vcan0 type vcan
$ sudo ip link set up vcan0
```

Figure 19: Virtual can interface set up.

After setting the virtual interface is it possible to check its status:

```
mario@ubuntu:~/Desktop/pruebaPedro$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:8a:bc:7b brd ff:ff:ff:ff:ff:ff
    altname enp2s1
    inet 192.168.238.128/24 brd 192.168.238.255 scope global dynamic noprefixroute ens33
       valid_lft 1594sec preferred_lft 1594sec
    inet6 fe80::20d2:29f9:55ce:a2b2/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
5: vcan0: <NOARP,UP,LOWER_UP> mtu 72 qdisc noqueue state UNKNOWN group default qlen 1000
    link/can
```

Figure 20: Check that interface is created.

In order to achieve the correct communication between the python program and the interface is it necessary to perform some changes in the code as shown in the next figure:

```
# Bring up can1 interface at 500kbps
try:
        os.system("sudo /sbin/ip link set vcan0 up type can bitrate 500000")
except OSError:
        print('An error was encountered while bringing up CAN interface')
        print('An error was encountered while bringing up CAN interface',file = outfile)
        exit()

time.sleep(0.5)
print('CAN1 interface ready')

#Bring up CAN bus
try:
        bus = can.interface.Bus(channel='vcan0', bustype='socketcan_native')
except OSError:
        print('An error was encountered while bringing up CAN bus')
        print('An error was encountered while bringing up CAN bus',file = outfile)
        exit()
```

Figure 21: Code modifications for virtual testing.

In order to simulate a car´s ECU, previously recorded CAN readouts have been used through *can-utils* Linux package. Two main functions of this package will be used:

- **Candump:** In order to show the raw data received through the virtual can interface
- **Canplayer:** In order to reproduce the previously recorded traces through the same interface.

So, for the sending part in one of the terminals the next command will be run in order to replicate the initial log recorded from the car in the virtual interface vcan0:

```
$ canplayer vcan0=can1 -I initialLog.log
```

Figure 22: Canplayer usage to reproduce the recordings.

Notice that it must be specified that the previously recorded can1 interface is not the desired one, it is the vcan0 instead.

In another terminal the raw data received from this recorded is shown:

```
mario@ubuntu:~$ candump vcan0
  vcan0  7DF   [8]  02 01 3B 00 00 00 00 00
  vcan0  7DF   [8]  02 01 3D 00 00 00 00 00
  vcan0  7DF   [8]  02 01 3F 00 00 00 00 00
  vcan0  7DF   [8]  02 01 21 00 00 00 00 00
  vcan0  7DF   [8]  02 01 23 00 00 00 00 00
  vcan0  7DF   [8]  02 01 26 00 00 00 00 00
  vcan0  7DF   [8]  02 01 27 00 00 00 00 00
  vcan0  7DF   [8]  02 01 28 00 00 00 00 00
  vcan0  7DF   [8]  02 01 30 00 00 00 00 00
  vcan0  7DF   [8]  02 01 32 00 00 00 00 00
  vcan0  7DF   [8]  02 01 33 00 00 00 00 00
  vcan0  7DF   [8]  02 01 34 00 00 00 00 00
  vcan0  7DF   [8]  02 01 37 00 00 00 00 00
  vcan0  7DF   [8]  02 01 39 00 00 00 00 00
  vcan0  7DF   [8]  02 01 3B 00 00 00 00 00
  vcan0  7DF   [8]  02 01 3D 00 00 00 00 00
```

Figure 23: Raw data received in the vcan interface.

And in the last terminal the python program will be run. As shown in the next figure, firstly creates and sets up the CAN interface and then it creates the needed variables in the Linux shared memory:

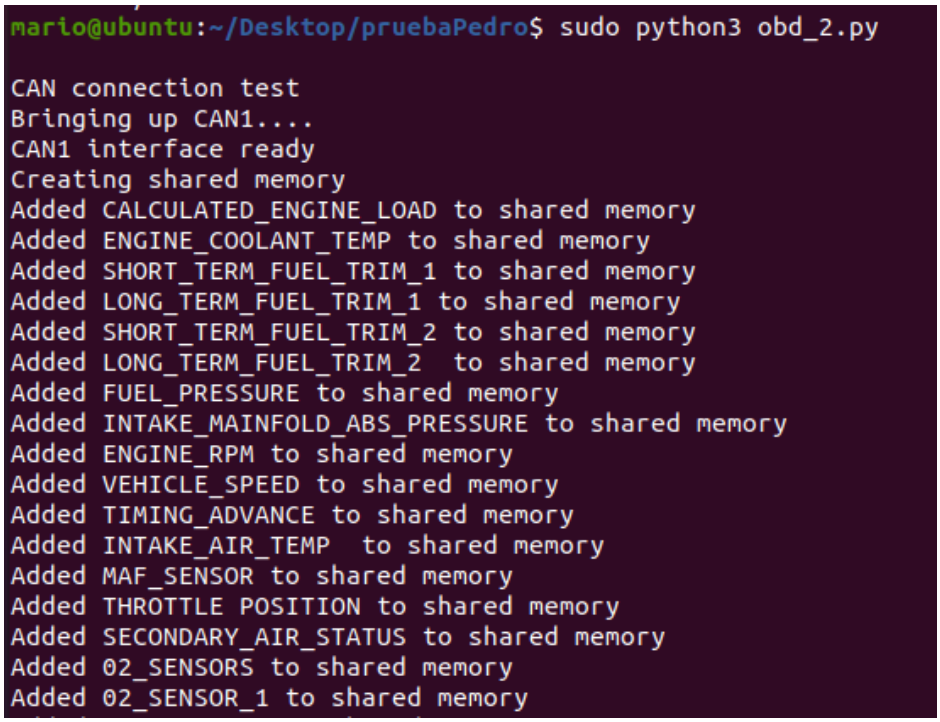


Figure 24: First output from the program.

And then it starts by requesting the available PIDs in the ECU to finally request them in an infinite loop and translate them into human readable data:

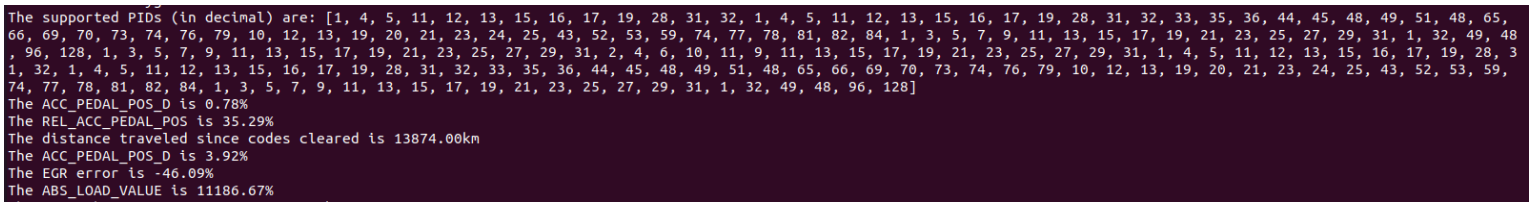


Figure 25: Format of data to be shown to the user.

The same format shown in the terminal in the figure above is used to save the information into the log file. In order to send the information through Redis network the previously shown format must be used.

Is it also interesting to observe how fast is possible the program to detect and show the value changes to the user. For this purpose, a fast gas pedal pressure was performed and recorded to then pass it through the program and check the engine´s rpm. In the next figure the result is shown:

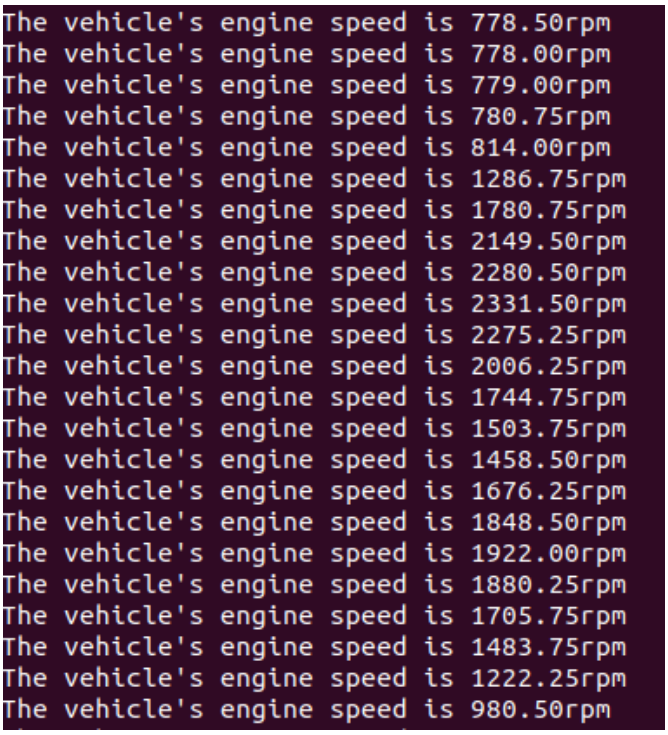


Figure 26: Example of fast gas pedal pressure.

Observing the result, it can be said that it is very successful taking into account that the biggest gap from one recording to another is around 500rpm, which is a small variance in terms of engine speed.

In the next figure a more diverse result in terms of variables is shown:

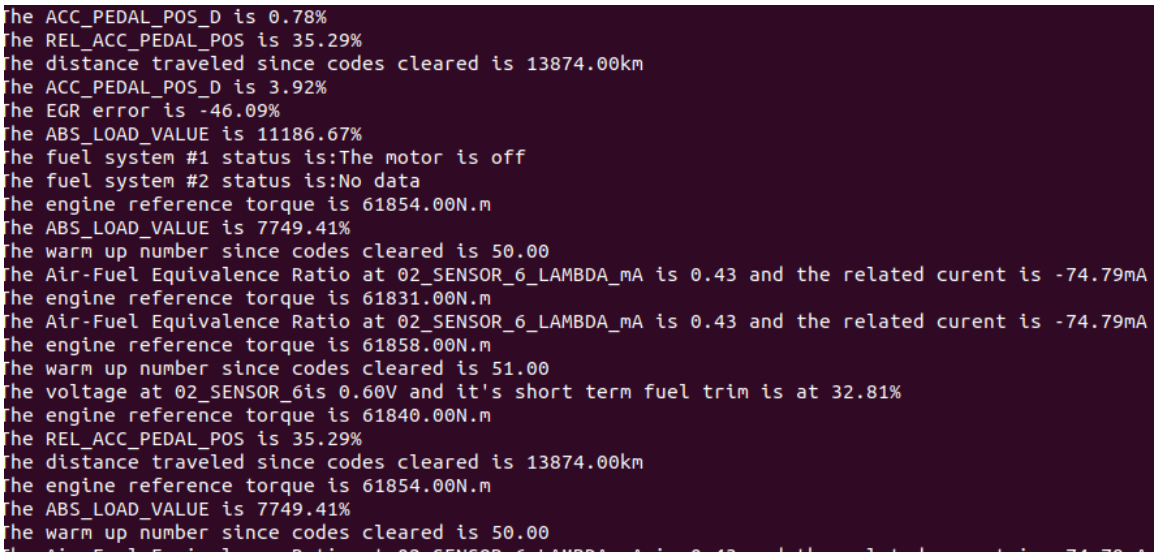


Figure 27: Diverse data visualization.

At this point, it has been shown through the virtual testing that the program is performing as expected and the next testing step can be started, real environment testing.

8.7.2. Real environment testing

In order to perform the real environment testing, the previously explained setup with the Owasys developers board is implemented and connected to a generic car´s OBD connector. As we know, the owa450 will power itself via the OBD connector and will start the communication with the ECU as soon as the program is started. Then it will ask for the available PIDs in the ECU and start asking for those periodically.

As the visual result is the same as in the virtual testing, it is not worth to explain it again. Instead, the next video shows the program running in a real environment:

https://github.com/Orbesito/OwasysTFM/blob/main/Screencast-OBD.mp4

8.7.3. Testing conclusions

As shown through the testing sections, the program performs as designed and expected, so it can be affirmed that the development of the project was successful. This program will be a functional CAN platform for Owasys clients that can be modified as desired, bringing more flexibility and support for its clients.

.

It is worth pointing out that for future developments Owasys has planned to design and implement a real-time user interface to offer to the clients an easy access to all the data retrieved from the car, but as it is not of the scope of this project it has not been done yet and thus it cannot be shown.

9.Planning

This section outlines the planned phases, work packages, and tasks for the successful development of the project. Each phase is structured as a Work Package (WP), consisting of multiple tasks (T). Additionally, the required resources, materials, and the members of the work group will be described. A Gantt diagram will also be provided to visualize the project timeline.

9.1. Work-group

In the next table the different members of the work group are shown alongside the charge and role of each one:

Code	Name	Charge	Role
I1	Eduardo Jacob Taquet	Senior Engineer	Management and overview of the project
I2	Maider Huarte Arrayago	Senior Engineer	Management and overview of the project
I3	Unai Orbe Deusto	Junior Engineer	Project developer

Table 4: Work-group

9.2. Phases

This section defines the different phases, work packages, and related tasks in the project planning

9.2.1. WP1 – Project Management and Documentation

This work package spans the entire project duration and focuses on project management and documentation tasks. Monthly meetings will be conducted with the senior engineers to ensure project progress and address any necessary corrections or alternatives. The following tasks are defined:

- T11 - Project Overview: This task involves regular meetings where the main developer provides progress updates to the senior engineers and discusses necessary corrections or alternatives. Duration: 30 hours.
- T12 - Documentation Development: Documentation will be created throughout the project to gather useful information and facilitate the final document preparation. Duration: 80 hours.

The human and material resources required for this work package are as follows:

- I1, I2: 30 hours each.
- I3: 100 hours.
- PC: 100 hours.

9.2.2. WP2 – Project Definition

This phase is crucial as it sets the foundation for the project. The tasks in this work package include defining the project's specifications, objectives, necessary concept studies, alternative analysis, and risk analysis. The following tasks are defined:

- T21 - Project Specifications and Objectives Definition: This task involves defining the project's scope, principal objectives, and necessary resources. Duration: 15 hours.
- T22 - Study of Necessary Concepts: Research and study will be conducted on topics such as CAN and OBD protocols, Python coding and system interconnection. Duration: 80 hours.
- T23 - Alternative Analysis: An analysis will be performed to select the best options for resources, (alternatives) Duration: 40 hours.
- T24 - Risk Analysis: This task involves analyzing and describing potential risks that may arise during the project's development, assigning probabilities and impact levels to each risk. Duration: 20 hours.

The human and material resources required for this work package are as follows:

- I1, I2: 15 hours each.
- I3: 120 hours.
- PC: 130 hours.

9.2.3. WP3 – Implementation Design and Execution

After defining the objectives and conducting necessary research, this work package focuses on designing and executing the test bench for accurate benchmarking and achieving desired results. The following tasks are defined:

- T31 - Test Bench Design: This task involves defining the values to be measured, required resources, and monitoring procedures to obtain the desired results. Duration: 50 hours.
- T32 - Test Bench Correct Functioning Validation: Before conducting any real benchmarks, it is essential to ensure the proper functioning of the test bench. CPU and RAM usage will be monitored to ensure that only the necessary programs are running on the hardware devices. Duration: 20 hours.
- T33 - Test Bench Implementation: The designed test bench will be implemented on the required hardware, and the designed tests will be executed. Duration: 50 hours.

The human and material resources required for this work package are as follows:

- I1, I2: 10 hours each.
- I3: 120 hours.
- PC: 110 hours.

9.2.4 WP4 – Testing and Analysis

This work package focuses on designing a methodology to test the program and analyze the data obtained from the car. The following tasks are defined:

- T41 - Virtual and real environment testing: Defining the best approach for testing the program and fixing the possible problems or malfunctions. Duration: 50 hours.
- T42 - Analysis of the retrieved data: Following the designed test methodology, all acquired data will be stored, sorted, analyzed and compared to the real data. Duration: 20 hours.

The human and material resources required for this work package are as follows:

- I1, I2: 20 hours each.
- I3: 70 hours.
- PC: 70 hours.

10.Executed budget

In this chapter, we outline the financial resources required for the project, including human resources (senior and junior engineers) and hardware resources (PC and Owa4x).

10.1. Internal hours

When determining the budget, various factors need consideration. For human resources, this involves defining the hourly wage and total hours, as detailed below:

- Senior engineer: 65 €/h.
- Junior Engineer: 30 €/h.

Work-Package	Hours			Cost (€)
	Unai Orbe	Maider Huarte	Eduardo Jacob	
WP1	110	30	30	7200
WP2	155	15	15	6600
WP3	120	10	10	6200
WP4	90	20	20	5300
Total	475	75	75	**25300**

Table 5: Total and individual hours and cost.

10.2. Amortization

Hardware usage and its amortization are equally crucial. To account for this, we calculate the hardware's cost, the hours it's used, and its expected lifespan. The table below illustrates the amortization of the utilized hardware-

Hardware	Initial cost (€)	Useful life (months)	Project usage (months)	Amortisation cost (€)
Personal Computer	800	60	9	140
Owa4x	500	84	9	53.5

Table 6: Hardware amortization.

10.3. Expenses

Other secondary resources are also necessary for the correct development of the project. Those are shown in the next table:

Resource	Cost (€)
Office supply	45
Internet access	50
Electricity	65
Total	**160**

Table 7: Secondary expense.

10.4. Total budget

In conclusion, the total budget for the entire project is derived from the sum of internal hours, amortization, and expenses. It's important to note that a 10% contingency margin is included in the budget to account for any unforeseen delays or issues that may arise.

Concept	Cost (€)
Internal hours	25300
Amortisation	193.5
Expenses	160
10% margin	2565.35
Total	**28218.85**

Table 8: Total executed budget.

11.Conclusions

The development of this project has shown that is possible to achieve a functional industrial use program implementing exclusively open-source libraries and modules and self-developed code. This program will allow future Owasys clients to connect and retrieve real-time data from any commercial car on the market that has an OBD interface. These clients will also have the ability to easily modify the program and its variables to fit into the desired performance or requirements. This way the clients will not have to develop a platform to use the CAN interface from any of the Owasys devices, bringing a huge advantage to both clients and company.

In terms of achieving the predefined requirements of the project, the correct CAN communication between device and ECU has been performed, the compatibility with generic commercial cars has been proved with several car brands and a lightweight solution that fits in the 1GB of storage of the owa450 device has been achieved. It can be said that all the initial requirements have been satisfied.

When it comes to personal growth and learning, this project allowed myself to be in some challenging scenarios where I had to find the way to solve them, which made me more confident after solving them. It has showed me the importance of communication between the different team members in order to achieve a correct behaviour of the team. And last but not least, it has brought me closer to a real work environment where I had to perform and deliver.

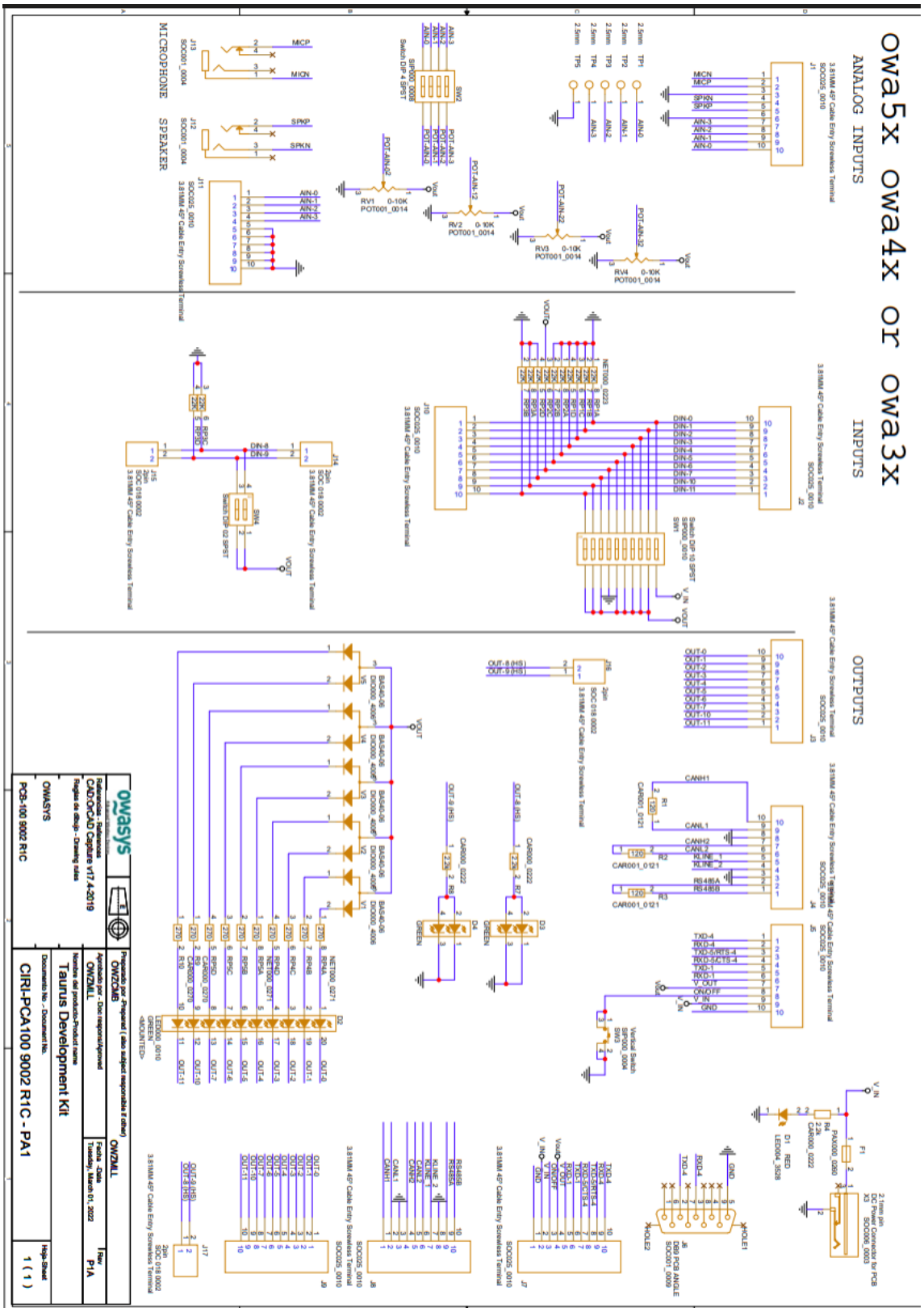
Summarizing, the project has been successfully finished for both the company and myself, offering the clients a plug and play solution for CAN communications with Owasys devices and developing my skills and workflow throughout the project.

12.References

[1] Cia and CAN history. Available: https://www.can-cia.org/about-us/the-cia-story/
[2] CanOpen specifications. Available: https://www.can-cia.org/canopen/
[3] Can-fd knowledge. Available: https://www.can-cia.org/can-knowledge/can/can-fd/
[4] OBD-2 interface introduction. Available: https://www.csselectronics.com/pages/obd2-explained-simple-intro
[5] CAN protocol introduction. Available: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial
[6] DBC file introduction. Available: https://www.csselectronics.com/pages/can-dbc-file-database-intro
[7] Generic PIDs description and transformation. Available: https://en.wikipedia.org/wiki/OBD-II_PIDs
[8] C++ features. Available: https://www.geeksforgeeks.org/c-plus-plus/
[9] Python features. Available: https://www.python.org/doc/essays/blurb/
[10] Node.js features. Available: https://nodejs.org/en/about
[11] Owa450 device datasheet. Available: https://www.owasys.com/bundles/owasysweb/docs/products/owa450/DESI-BOK100_9101_latest_owa450_IP40_PLATFORM_DATASHEET.pdf

13.Annexes

13.1. Annex 1: Owasys Developers Board Schematics



13.2. Annex 2: Installed Debian and Python packages

Firstly, the installed Debian packages are shown:

```
root@rootfspython:~# dpkg --list |grep python
ii  dh-python                3.20190308                  all          Debian helper tools for packaging Python libraries and applications
ii  libpython3-dev:armhf     3.7.3-1                     armhf        header files and a static library for Python (default)
ii  libpython3-stdlib:armhf  3.7.3-1                     armhf        interactive high-level object-oriented language (default python3 version)
ii  libpython3.7:armhf       3.7.3-2+deb10u3             armhf        Shared Python runtime library (version 3.7)
ii  libpython3.7-dev:armhf   3.7.3-2+deb10u3             armhf        Header files and a static library for Python (v3.7)
ii  libpython3.7-minimal:armhf 3.7.3-2+deb10u3           armhf        Minimal subset of the Python language (version 3.7)
ii  libpython3.7-stdlib:armhf 3.7.3-2+deb10u3            armhf        Interactive high-level object-oriented language (standard library, version 3.7)
ii  python-pip-whl           18.1-5                      all          Python package installer
ii  python3                  3.7.3-1                     armhf        interactive high-level object-oriented language (default python3 version)
ii  python3-asn1crypto       0.24.0-1                    all          Fast ASN.1 parser and serializer (Python 3)
ii  python3-cffi-backend     1.12.2-1                    armhf        Foreign Function Interface for Python 3 calling C code - runtime
ii  python3-crypto           2.6.1-9+b1                  armhf        cryptographic algorithms and protocols for Python 3
ii  python3-cryptography     2.6.1-3+deb10u2             armhf        Python library exposing cryptographic recipes and primitives (Python 3)
ii  python3-dbus             1.2.8-3                     armhf        simple interprocess messaging system (Python 3 interface)
ii  python3-dev              3.7.3-1                     armhf        header files and a static library for Python (default)
ii  python3-distutils        3.7.3-1                     all          distutils package for Python 3.x
ii  python3-entrypoints      0.3-1                       all          Discover and load entry points from installed packages (Python 3)
ii  python3-gi               3.30.4-1                    armhf        Python 3 bindings for gobject-introspection libraries
ii  python3-keyring          17.1.1-1                    all          store and access your passwords safely - Python 3 version of the package
ii  python3-keyrings.alt     3.1.1-1                     all          alternate backend implementations for python3-keyring
ii  python3-lib2to3          3.7.3-1                     all          Interactive high-level object-oriented language (2to3, version 3.6)
ii  python3-minimal          3.7.3-1                     armhf        minimal subset of the Python language (default python3 version)
ii  python3-pip              18.1-5                      all          Python package installer
ii  python3-pkg-resources    40.8.0-1                    all          Package Discovery and Resource Access using pkg_resources
ii  python3-secretstorage    2.3.1-2                     all          Python module for storing secrets - Python 3.x version
ii  python3-setuptools       40.8.0-1                    all          Python3 Distutils Enhancements
ii  python3-six              1.12.0-1                    all          Python 2 and 3 compatibility library (Python 3 interface)
ii  python3-venv             3.7.3-1                     armhf        pyvenv-3 binary for python3 (default python3 version)
ii  python3-wheel            0.32.3-2                    all          built-package format for Python
ii  python3-xdg              0.25-5                      all          Python 3 library to access freedesktop.org standards
ii  python3.7                3.7.3-2+deb10u3             armhf        Interactive high-level object-oriented language (version 3.7)
ii  python3.7-dev            3.7.3-2+deb10u3             armhf        Header files and a static library for Python (v3.7)
ii  python3.7-minimal        3.7.3-2+deb10u3             armhf        Minimal subset of the Python language (version 3.7)
ii  python3.7-venv           3.7.3-2+deb10u3             armhf        Interactive high-level object-oriented language (pyvenv binary, version 3.7)
```

Next the Python and pip packages and their dependencies:

```
2   -python3
3       -bzip2 file libmagic-mgc libmagic1 libmpdec2 libpython3-stdlib
4           libpython3.7-minimal libpython3.7-stdlib mime-support python3-minimal
5           python3.7 python3.7-minimal xz-utils
6
7   -pip
8       -binutils binutils-arm-linux-gnueabihf binutils-common build-essential cpp
9       cpp-8 dh-python dpkg-dev fakeroot g++ g++-8 gcc gcc-8 gir1.2-glib-2.0
10      libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl
11      libasan5 libbinutils libc-dev-bin libc6-dev libcc1-0 libdpkg-perl
12      libexpat1-dev libfakeroot libfile-fcntllock-perl libgcc-8-dev
13      libgdbm-compat4 libgdbm6 libgirepository-1.0-1 libisl19 libmpc3 libmpfr6
14      libperl5.28 libpython3-dev libpython3.7 libpython3.7-dev libstdc++-8-dev
15      libubsan1 linux-libc-dev make manpages manpages-dev patch perl
16      perl-modules-5.28 python-pip-whl python3-asn1crypto python3-cffi-backend
17      python3-crypto python3-cryptography python3-dbus python3-dev
18      python3-distutils python3-entrypoints python3-gi python3-keyring
19      python3-keyrings.alt python3-lib2to3 python3-pkg-resources
20      python3-secretstorage python3-setuptools python3-six python3-wheel
21      python3-xdg python3.7-dev
22          -pip packages:
23              -python-can
24              -redis
25              -numpy
```

And finally, the version of all pip installed packages:

```
root@A06T86:~# pip3 list
Package       Version
------------- -------
asn1crypto    0.24.0
cryptography  2.6.1
entrypoints   0.3
keyring       17.1.1
keyrings.alt  3.1.1
numpy         1.16.2
pip           18.1
pycrypto      2.6.1
PyGObject     3.30.4
pyserial      3.4
python-can    3.0.0
pyxdg         0.25
redis         3.2.1
SecretStorage 2.3.1
setuptools    40.8.0
six           1.12.0
sysv-ipc      0.6.8
wheel         0.32.3
```

13.3. Annex 3: Shared Memory Reader and Writer code

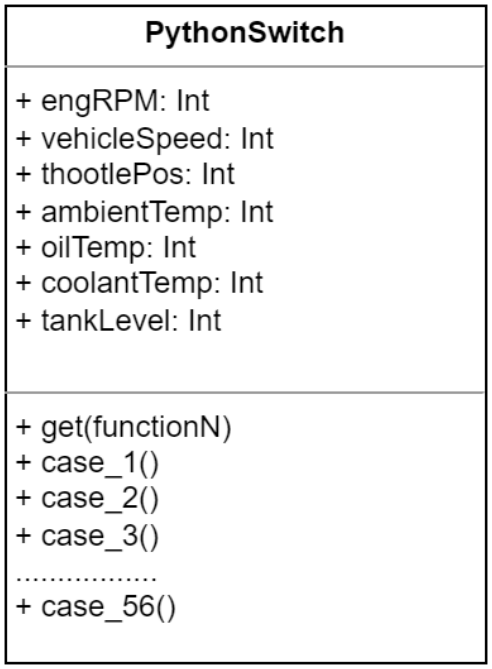
```
import asyncio
import time
import sys
# POSIX shared memory
from ctypes import (Union, Structure, c_uint, c_ulong, c_int,
                    c_long, c_float, c_double, c_longlong, c_char, c_void_p,
c_size_t,
                    sizeof)


from numpy import byte
import sysv_ipc

class VAL(Union):
    '''Pollux Shmem value'''
    _fields_ = [("bVal", c_int),
                ("iVal", c_long),
                ("fVal", c_float),
                ("dVal", c_double),
                ("llVal", c_longlong)]

CHARARR_32 = c_char * 32
class ShVal(Structure):
    '''Pollux Shmem struct'''
    _fields_ = [("type", c_uint), # INT = 0, FLOAT = 1, BOOL = 2, LONG_LONG
= 3, DOUBLE = 4
                ("access", c_uint),
                ("tag", CHARARR_32),
                ("val", VAL),
                ("count", c_ulong),
                ("stamp", c_ulong),
               ]

def type_to_val(sh_val):
    '''Switch on the shared memory type union'''
    val = None
    var_type = sh_val.type
    if var_type == 0:
        val = sh_val.val.iVal
    elif var_type == 1:
        val = sh_val.val.fVal
    elif var_type == 2:
        val = sh_val.val.bVal
    elif var_type == 3:
        val = sh_val.val.llVal
    elif var_type == 4:
        val = sh_val.val.dVal
    else:
        print("Weird variable type")
    #print(f"SHM variable {sh_val.tag} with value {val}, ts
{sh_val.stamp},type {sh_val.type} received")
    return val

def set_val(sh_val,value):
```

```
    '''Switch on the shared memory type union'''
    var_type = sh_val.type
    if var_type == 0:
        sh_val.val.iVal = value
    elif var_type == 1:
        sh_val.val.fVal = value
    elif var_type == 2:
        sh_val.val.bVal = value
    elif var_type == 3:
        sh_val.val.llVal = value
    elif var_type == 4:
        sh_val.val.dVal = value
    else:
        print("Weird variable type")
        return
    # Set new stamp and increment count
    sh_val.stamp = int(time.time())
    sh_val.count = sh_val.count + 1

class ShmReader():
    def __init__(self):
        self.status = {}

    async def print_status(self,var):
        try:
            while True:
                print(var)
                await asyncio.sleep(1)
        except asyncio.CancelledError:
            return "print_status has been cancelled!"

    async def periodic_reader(self, memory_key, interval=1):
        '''Coroutine to read pollux shared-memory structures periodically
(infinite loop)'''
        print(f"Shared-memory reader: key: {memory_key} - interval:
{interval}s")
        try:
            memory = sysv_ipc.SharedMemory(memory_key)
        except sysv_ipc.ExistentialError:
            print(f"The shared memory segment with key {memory_key} does not
exist.")
        else:
            try:
                while True:
                    num_of_structs =
int.from_bytes(memory.read(byte_count=sizeof(c_longlong)),
                                                    byteorder='little')

                    for num in range(0, num_of_structs):
                        memory_value = memory.read(sizeof(ShVal),
sizeof(c_long)+sizeof(ShVal)*num)
                        shared_buffer = bytearray(memory_value)
                        shared = ShVal.from_buffer(shared_buffer)
                        val = type_to_val(shared)
                        self.status.update({str(shared.tag, "ISO-8859-1"):
val})

                #memory.detach()
```

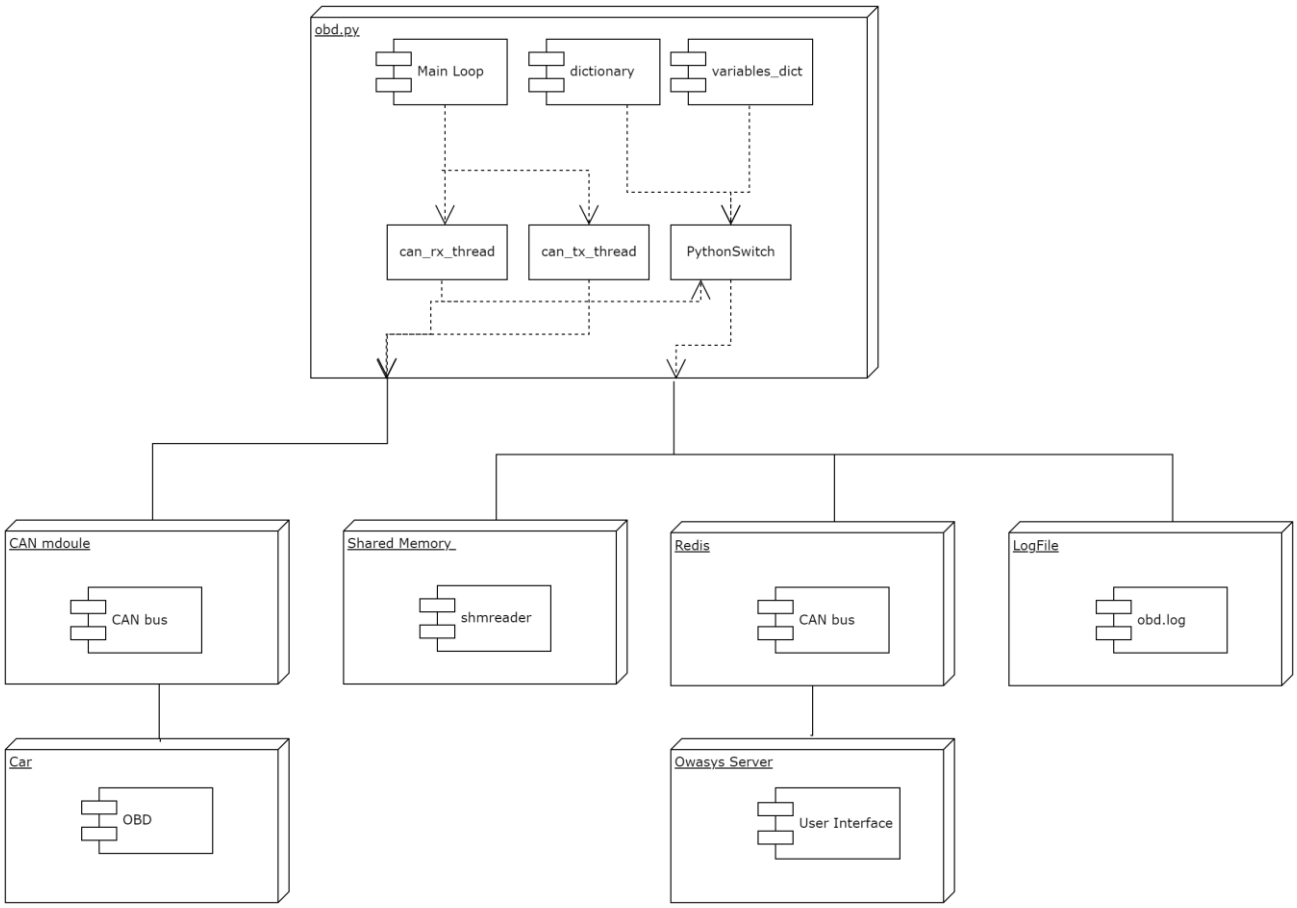
```
                    await asyncio.sleep(interval)

            except asyncio.CancelledError:
                return "Shared-memory reader has been cancelled!"

    def update_shm_key(self, memory_key, tag, value):
        try:
            memory = sysv_ipc.SharedMemory(memory_key)
        except sysv_ipc.ExistentialError:
            print(f"The shared memory segment with key {memory_key} does not
exist")
        else:
            num_of_structs =
int.from_bytes(memory.read(byte_count=sizeof(c_longlong)),
                                                    byteorder='little')
            for num in range(0, num_of_structs):
                memory_value = memory.read(sizeof(ShVal),
sizeof(c_long)+sizeof(ShVal)*num)
                shared_buffer = bytearray(memory_value)
                shared = ShVal.from_buffer(shared_buffer)
                if str(shared.tag, "ISO-8859-1") == tag:
                    set_val(shared,value)
                    memory.write(shared,sizeof(c_long)+sizeof(ShVal)*num)
                    memory.detach()
                    return
                else:
                    continue

            memory.detach()
            print(f"There in no variable with tag {tag} defined in shared
memory id {memory_key}")


    def create_shm(self, memory_key,data):
        memory = sysv_ipc.SharedMemory(memory_key, flags=sysv_ipc.IPC_CREAT,
mode=644, size=(len(data)*sizeof(ShVal)+sizeof(c_longlong)))
        nofdata =  c_ulong(len(data))
        memory.write(nofdata,0)
        i = 0
        for item in data:
            shm_val = ShVal()
            shm_val.tag = str.encode(item)
            shm_val.type = data[item]
            memory.write(shm_val,sizeof(c_ulong)+sizeof(ShVal) * i)
            i = i + 1
            print(f"Added {item} to shared memory")
        print(f"Shared memory created with {len(data)} elements")
```

13.4. Annex 4: UML Diagrams

13.4.1. Class diagram



13.4.2. Component Diagram



13.4.3. Activity Diagram