Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño y desarrollo de un sistema de monitorización de códigos de averías (DTC-OBD) en flotas de vehículos

Autor: Sergio Román González

Tutor: Juan Manuel Vozmediano Torres

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño y desarrollo de un sistema de monitorización de códigos de averías (DTC-OBD) en flotas de vehículos

Autor:
Sergio Román González

Tutor:
Juan Manuel Vozmediano Torres
Profesor Titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado: Diseño y desarrollo de un sistema de monitorización de códigos de averías (DTC-OBD) en flotas de vehículos

Autor: Sergio Román González
Tutor: Juan Manuel Vozmediano Torres

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Me gustaría empezar agradeciendo a todos los docentes con los que he tenido la suerte de cruzarme en este largo camino de crecimiento personal. En especial, a Juan Manuel Vozmediano Torres por su ayuda en el desarrollo de este trabajo y por ser un referente para mí, descubriendo a lo que quiero dedicarme profesionalmente gracias a él.

A lo largo de estos años he conocido a grandes personas que me han acompañado durante esta etapa. A mis compañeros Daniel, Javier, Jesús, Ildefonso, Diego, Guillermo y Abraham, muchas gracias por la ayuda ofrecida y por los buenos momentos que hemos vivido juntos.

No me puedo quedar sin agradecer a mis padres, las personas que me han visto crecer, las que me han apoyado y me han acompañado a la estación en cada viaje que tenía que realizar. Muchas gracias por todo el amor que me habéis dado.

Por último, el agradecimiento más grande que puedo dar, a la persona que me ha acompañado durante estos últimos años y que más me ha ayudado. Por cada abrazo, por cada beso, por cada sonrisa, por cada momento juntos, apoyándome y motivándome, por hacerme feliz, cuando parecía que nada de esto era posible, muchas gracias Cristina Lara. Te quiero. Siempre tú.

Sergio Román González
Sevilla, 2020

Resumen

No cabe duda de la importancia actual de disponer de herramientas que permitan conocer el estado de los vehículos en todo momento. Este sistema, denominado OBD (On Board Diagnostics), es capaz de diagnosticar fallos eléctricos, químicos y mecánicos, y obtener información de los diferentes sensores del vehículo.

El presente documento tiene por objeto el desarrollo de una librería que ofrezca la posibilidad de comunicación bluetooth con el sistema de diagnóstico de vehículos y de un sistema de monitorización de códigos de averías (DTC) en flotas de vehículos.

En primer lugar, se ha realizado una introducción y descripción del sistema OBD para entender los conceptos más importantes, junto con una breve presentación de las herramientas utilizadas en este proyecto. Seguidamente, se detalla la especificación de requisitos del sistema que sirve como base para el diseño y desarrollo software posterior. Por último, se explica el plan de prueba realizado y se finaliza con un apartado de conclusiones obtenidas tras terminar el proyecto.

Abstract

There is no doubt about the current importance of having tools to know the status of vehicles at all times. This system, called OBD (On Board Diagnostics), is capable of diagnosing electrical, chemical and mechanical faults, and obtaining information from the different sensors of the vehicle.

The purpose of this document is to develop a library that offers the possibility of bluetooth communication with the vehicle diagnostic system and of a fault code (DTC) monitoring system in vehicle fleets.

First, an introduction and description of the OBD system has been made to understand the most important concepts, along with a short presentation of the tools used in this project. The following is the system requirements specification that serves as the basis for subsequent software design and development. Finally, the test plan carried out is explained and ends with a section of conclusions obtained after completing the project.

Índice

Resumen III
Abstract V
Índice VII

1 Introducción 1
1.1 Motivación del proyecto 1
1.2 Objetivos 1
1.3 Metodología y plan de trabajo 1
1.4 Arquitectura general y componentes 2

2 Entorno de trabajo y tecnologías 3
2.1 Lenguaje de programación y librerías 3
2.2 Entorno de desarrollo 4
2.3 Control de versiones 4
2.4 Bluetooth 4
2.5 Node.js, MongoDB y Express 4
2.6 OBDSIM y minicom 5
2.7 UMLet 5

3 Descripción del sistema de diagnóstico a bordo OBD 7
3.1 Introducción histórica 7
3.2 Terminología 8
3.2.1 Unidad de Control Electrónico/Motor (ECU) 8
3.2.2 Conector de Enlace de Datos (DLC) 8
3.2.3 Bus CAN 10
3.2.4 ELM327 10
3.2.5 Códigos de Diagnóstico de Problemas (DTC) 11
3.2.6 Identificador de parámetros (PID) 11
3.2.7 Luz indicadora de mal funcionamiento (MIL) 12
3.3 Funcionamiento 12
3.3.1 CAN 12
3.3.2 OBDII 14
3.3.3 Interpretación DTC 15
3.3.4 Comandos AT 16

4 Especificación de requisitos del sistema (ERS) 19
4.1 Introducción 19
4.1.1 Propósito del sistema 19
4.1.2 Alcance del sistema 19
4.1.3 Descripción general del sistema 19
4.1.3.1 Contexto del sistema 19

4.1.3.2 Características de los actores 19
4.2 Descripción de subsistemas a desarrollar 20
4.3 Casos de uso 21
4.3.1 Gestión de la conexión con ELM327 22
4.3.2 Gestión del envío y recepción de mensajes OBD 25
4.3.3 Gestión de lectura de ficheros de configuración 28
4.3.4 Gestión de los datos 28
4.4 Catálogo de requisitos del sistema 30
4.4.1 Requisitos funcionales de información 30
4.4.2 Requisitos funcionales de reglas de negocio 32
4.4.3 Requisitos funcionales de conducta 33
4.4.4 Requisitos no funcionales de fiabilidad 34
4.4.5 Requisitos no funcionales de usabilidad 34
4.4.6 Requisitos no funcionales de eficiencia 35
4.4.7 Requisitos no funcionales de mantenibilidad 36
4.4.8 Requisitos no funcionales de portabilidad 36
4.4.9 Requisitos no funcionales de seguridad 37
4.4.10 Restricciones técnicas 37
4.4.11 Requisitos de integración 38

5 Diseño del sistema 39
5.1 Diagrama de paquetes 39
5.2 Diagrama de componentes 40
5.3 Diagrama de clases 40
5.4 Diagrama de actividad 41
5.5 Diagrama de secuencia 41

6 Implementación 45
6.1 Arquitectura del proyecto 45
6.1.1 Aplicación cliente y librería obd2-bluetooth 45
6.1.2 Servidor Web con BBDD MongoDB 47

7 Plan de pruebas 51
7.1 Pruebas unitarias 51
7.2 Pruebas de integración 52
7.3 Procedimiento de pruebas y resultados 55

8 Conclusiones 61
8.1 Conclusión y línea futura de trabajo 61

Anexo A Manual de instalación 63
A.1 Instalación en Raspberry Pi 63
A.2 Instalación en el servidor 66
A.3 Generación del ejecutable para las pruebas 67

Anexo B Manual de uso de la librería y entorno 69
B.1 Utilización de monDTC en Raspberry Pi 69
B.2 Utilización de obd2-server en el servidor remoto 70

Índice de Figuras 151
Índice de Tablas 153
Índice de Códigos 155
Bibliografía 157

1 Introducción

1.1 Motivación del proyecto

Actualmente el transporte es un sector estratégico básico para el desarrollo global de la sociedad y de su economía. Éste garantiza la movilidad de los ciudadanos, así como la libre circulación de mercancías y constituye una actividad importante en continuo proceso de expansión y modernización.

Esta modernización está marcada por los avances tecnológicos tanto en electrónica como en telecomunicaciones, que permiten mejorar la fiabilidad, adaptabilidad y autodiagnóstico del vehículo, además de disponer de información actualizada de su estado en cualquier localización con aplicaciones desarrolladas para este fin.

Para muchas empresas de transporte el tiempo de reparación de sus vehículos es primordial, dado que son su principal fuente de ingresos. Cuanto antes se conozca qué componente falla, antes se puede solicitar al proveedor un recambio y agilizar así el proceso. De la misma forma, con información actualizada del estado de sus flotas de transporte, junto con un buen mantenimiento integral y sistemático, estas empresas pueden garantizar la disponibilidad de los vehículos, disminuir las averías imprevistas, aumentar la fiabilidad, permitir la optimización de los recursos y, lo más importante, reducir los costes contribuyendo a la eficiencia global de la empresa.

Este proyecto surge del interés de conocer más en profundidad el funcionamiento interno de los vehículos y su comunicación con sistemas externos, así como aprovechar información relevante que pueda ser utilizada para reducir costes y tiempo en la reparación de éstos ante fallos.

1.2 Objetivos

El presente proyecto tiene como objetivo la implementación de una herramienta que permita mediante una conexión Bluetooth conocer el valor de los distintos sensores de un vehículo (turismos, vehículos comerciales ligeros, pesados y autobuses) y más en concreto, los fallos que puedan producirse en éste. Debe de tratarse de una herramienta robusta, eficiente y versátil, que pueda ser utilizada en dispositivos de poco coste y recursos de computación, ya que es necesaria la ejecución de esta herramienta en cada vehículo a monitorizar.

Así mismo, el fin de este proyecto es proporcionar al lector una introducción teórica para conocer el funcionamiento genérico del autodiagnóstico en vehículos y el modo de comunicación con sistemas externos a éste.

Por último, otra finalidad es dar al lector una visión global de las posibles aplicaciones que permiten la herramienta, desarrollando en este caso un sistema con un esquema de arquitectura cliente-servidor .

1.3 Metodología y plan de trabajo

La metodología de trabajo llevada a cabo ha consistido en estudiar la tecnología a implementar, analizar las necesidades del producto a desarrollar, diseñar la arquitectura software para posteriormente realizar el proceso de codificación. En la fase de desarrollo de código, se han realizado una serie de pruebas unitarias de las distintas funciones, y finalmente se han realizado pruebas de integración para validar el funcionamiento global.

1.4 Arquitectura general y componentes

Como introducción del sistema que se necesita desarrollar, se pueden observar en la Figura 1.1 los distintos componentes de la arquitectura y la relación entre ellos.

En primer lugar, se necesita un dispositivo ELM327 con interfaz bluetooth. Este dispositivo es capaz de comunicarse con el vehículo con distintos protocolos y proveer una interfaz de comunicación bluetooth para que otros dispositivos puedan conectarse.

En segundo lugar, se necesita un dispositivo con interfaz bluetooth para conectarse con ELM327. En la Figura 1.1 se muestra de ejemplo una Raspberry Pi, la cuál dispone de modelos con interfaz bluetooth y permite la instalación de programas desarrollados en distintos lenguajes. Se comunica con ELM327 para obtener información del vehículo.

En tercer lugar, se necesita un servidor remoto, el cual puede ser físico o virtual. Recepciona y almacena los datos enviados por las distintas Raspberry Pi de diversos vehículos mediante un servidor web y una base de datos. Además, ofrece una interfaz web para la monitorización de errores.

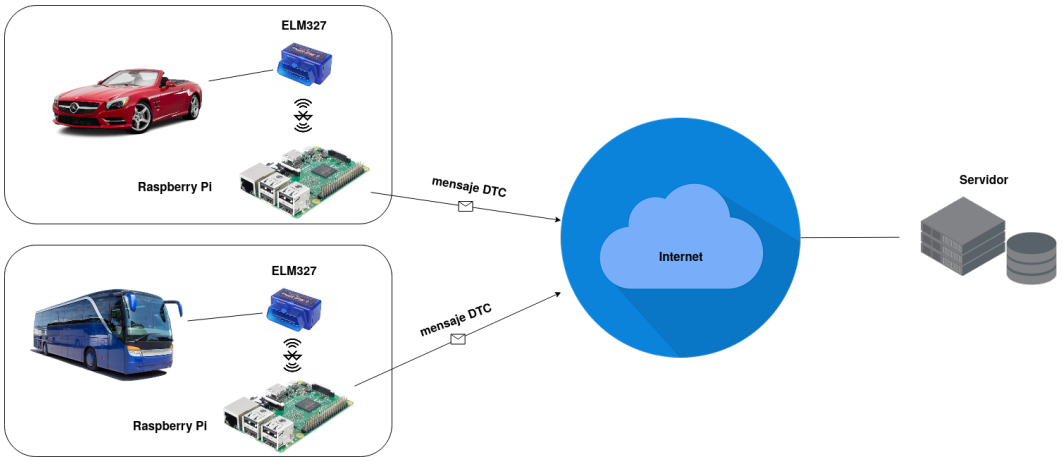


Figura 1.1 Arquitectura general y componentes.

2 Entorno de trabajo y tecnologías

En este apartado se recoge un listado de herramientas y tecnologías utilizadas durante la realización del proyecto y una breve descripción de ellas, junto con los motivos por las que son necesarias..

2.1 Lenguaje de programación y librerías

El lenguaje de programación utilizado en este proyecto es C++. Fue diseñado en 1979 por Bjarne Stroustrup con la intención de extender al lenguaje de programación C mecanismos que permiten la manipulación de objetos. Las principales ventajas de C++ por las que ha sido elegido en este proyecto son:

- **Portabilidad.** Si bien no es un lenguaje automáticamente distribuído en los sistemas Unix, prácticamente todos lo pueden ejecutar, ya sea en una variante comercial o mediante el popular GNU GCC/G++ con lo que la disponibilidad está asegurada.
- **Eficiencia.** Cuando nos referimos a la eficiencia estamos hablando principalmente de la velocidad con la cual un programa logra llevar a cabo diversas tareas. Al tratarse de un lenguaje compilado, se obtiene directamente el "código máquina" que es ejecutado directamente por el microprocesador. Esta simplicidad consigue que el programa tenga una mayor eficiencia en ciertas aplicaciones, a diferencia de lenguajes interpretados.
- **Muy extendido.** Es un lenguaje de largo recorrido que dispone de muchas librerías y mucha documentación que nos sirve para ahorrar tiempo de desarrollo.

En lo que respecta a las librerías externas utilizadas, cabe destacar las siguientes:

- **Framework catch2 [1].** La necesidad de realizar test unitarios y de integración, me llevaron a la búsqueda de un framework que me facilitara esta labor. Catch2 es un framework de testeo multiparadigma para C++ aunque también es compatible con Objective-C. Para su uso, únicamente se necesita incluir un archivo de cabecera en nuestro proyecto.
  Algunas de las características por las que he elegido esta librería es por la sencillez y rapidez de aprendizaje al utilizarla y no tener dependencias externas, únicamente disponer de C++11. Posee la lincencia Boost Software License 1.0 lo que permite reutilizar software dentro de software propietario y el uso comercial de éste.
- **JSON for Modern C++ [2].** A la hora de especificar el formato que deberían de tener los datos y la información obtenida de los vehículos, me decanté por la utilización de JSON (JavaScript Object Notation) por su uso ampliamente extendido en la comunidad de desarrolladores y por la cantidad de herramientas disponibles en distintos lenguajes de programación para integrar el procesamiento de este formato.
  La librería presentada facilita la creación y procesamiento de objetos JSON con una sintaxis intuitiva y una integración trivial , ya que sólo se necesita incluir el fichero de cabecera json.hpp en nuestro proyecto. Cabe destacar, que la utilización de ésta librería no afecta negativamente a nuestro código, dado que mantiene la eficiencia en memoria de los objetos JSON. Requiere disponer de C++11 y posee la licencia MIT License lo que permite reutilizar software dentro de software propietario y el uso comercial de éste.

Las librerías utilizadas incluidas en la mayoría de distribuciones Linux, son las siguientes:

- **BlueZ [3].** Se trata de un conjunto de módulos del kernel, librerías y utilidades que proporciona soporte para las capas y protocolos centrales de Bluetooth. Se puede encontrar en muchas distribuciones Linux y, en general, es compatible con cualquier sistema Linux del mercado.
  Este conjunto de librerías se enlazan en el proceso de compilación, ante la necesidad de comunicación inalámbrica con el vehículo.
- **POSIX Threads[4].** Usualmente referida como pthreads, es una API definida por el estándar POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995). Las implementaciones de la API están disponibles en muchos sitemas operativos UNIX y permiten crear multiples hilos para conseguir un flujo de procesamiento concurrente.
  El motivo de la utilización de esta librería es la necesidad de tener dos procesos ejecutándose de manera concurrente en la comunicación bluetooth: uno para el envío de datos y otro para la recepción de los mismos.

2.2 Entorno de desarrollo

El entorno de desarrollo integrado (IDE) elegido es Sublime Text [5]. Se trata de un editor de texto multiplataforma muy liviano, con edición avanzada, velocidad y es posible obtenerlo de forma gratuita en su versión de evaluación operativa sin fecha límite. Principalmente, me he decantado por esta herramienta por la facilidad de uso y de integración de sus plugins.

Los plugins instalados que han servido de ayuda en este proyecto son los siguientes:

- **SublimeGit [6].** Se trata de un plugin que integra la herramienta de control de versiones Git en Sublime Text. Permite utilizar los comandos de Git sin necesidad de salir del entorno de desarrollo aumentando así la productividad.
- **SublimeGDB [7].** Es un plugin que integra GNU Debugger (GDB) con Sublime Text. Me ha permitido depurar mi código para encontrar errores de manera más sencilla desde el propio Sublime.

2.3 Control de versiones

Para el control de versiones he utilizado la herramienta Git [8]. Git es un sistema de control de versiones distribuido, gratuito y de código abierto, diseñado para manejar proyectos con velocidad y eficiencia.

Fundamentalmente, he elegido Git porque es software libre y open source, me permite comparar o restaurar versiones del proyecto y contar con una copia del código fuente centralizada para volver atrás ante cualquier imprevisto. Para este proyecto se han creado dos repositorios privados en GitHub, uno para el código C++ de la librería y, por otro lado, para el código Javascript (NodeJS) del servidor de recepción de datos.

2.4 Bluetooth

Bluetooth es una especificación para redes inalámbricas de área personal (WPAN) que posibilita la transmisión de datos en distancias cortas entre diferentes dispositivos mediante un enlace por radiofrecuencia en la banda ISM de los 2.4 GHz.

La existencia de dispositivos de bajo coste como el microcontrolador ELM327, que puede disponer de interfaz bluetooth, evita la necesidad de cableado en la conexión con el conector OBDII del vehículo. Por ello, el desarrollo de la librería en este proyecto, tiene como parte clave el uso de la tecnología bluetooth.

2.5 Node.js, MongoDB y Express

Para poder mostrar un caso de uso de aplicación de la librería desarrollada, se decidió crear un servidor web que pudiera mostrar algún tipo de información relacionada con el vehículo. Este servidor web se ha desarrollado con Node.js y uno de sus frameworks más populares, Express. Node.js es un entorno de ejecución para JavaScript orientado a eventos asíncronos cuyas características más importantes son:

- **Gran rendimiento.** Ha sido diseñado para optimizar el rendimiento y la escalabilidad en aplicaciones web en tiempo real.

- **Gestor de paquetes NPM.** NPM (Node Packet Manager) proporciona acceso a un conjunto de componentes reutilizables disponibles públicamente a través de una fácil instalación y de un repositorio en línea, con la versión y la dependencia de gestión.
- **Javascript.** Node.js está basado en Javascript pudiendo desarrollar en el mismo lenguaje tanto en el navegador como en el servidor. Además, la integración con objetos en formato JSON es mayor que en otros lenguajes, cobrando importancia en este proyecto.

Por último, para conservar datos de los vehículos que contacten con el servidor web de forma persistente se ha utilizado una base de datos distribuida MongoDB. Está orientada a documentos, esto quiere decir que en lugar de guardar los datos en registros, guarda los datos en documentos. Estos documentos son almacenados en BSON, que es una representación binaria de JSON. Esto último, junto con la capacidad de realizar consultas utilizando JavaScript, ha decantado la utilización de esta base de datos NoSQL.

2.6 OBDSIM y minicom

La principal dificultad para realizar correctamente el plan de pruebas ha sido poder comprobar la correcta conexión bluetooth con la interfaz OBDII del vehículo y el posterior paso de mensajes. Para este fin, ha servido de gran ayuda OBDSIM [9].

Se trata de un simulador de un dispositivo ELM327 conectado a una o más ECU (Engine Control Unit), con el que se pueden especificar distintos parámetros como códigos de error y valores de alguno de los sensores del vehículo. Por otra parte, para el primer acceso a este simulador, era necesario iniciar un comunicación por el puerto serie, utilizando para ello el programa minicom, disponible en los sistemas UNIX.

2.7 UMLet

UMLet es una herramienta UML de código abierto desarrollada en Java para la creación de diagramas UML en base al Lenguaje Unificado de Modelado. Es una herramienta esencial en este proyecto permitiendo la realización de la fase de análisis y diseño software. Se ha elegido entre otras herramientas UML al tratarse de un software gratuito y sencillo de utilizar.

3 Descripción del sistema de diagnóstico a bordo OBD

En este apartado se procede a explicar el funcionamiento básico del sistema OBD y sus protocolos, tras realizar un breve repaso histórico de los inicios de los sistemas de diagnóstico en los vehículos. De este modo el lector entenderá de forma más clara, las necesidades de diseño del software que en este proyecto se desarrolla.

3.1 Introducción histórica

Para entender el estado actual de los sistemas de diagnóstico en los vehículos, hay que remontarse algunas décadas atrás para poder apreciar cuanto ha avanzado la tecnología. Mientras que el diagnóstico en los vehículos, conocido como OBD, no es un participante activo en las emisiones del vehículo, su desarrollo está relacionado estrechamente al sistema de control de emisiones de gases de los coches y camiones. De hecho, OBD-I fue introducido en 1989 cuyo objetivo principal era la vigilancia por parte del sistema electrónico del motor de los componentes eléctricos influyentes en los gases de escape y la emisión de una señal óptica de advertencia en caso de fallo.

A principios de la década de 1970, los vehículos vendidos en los Estados Unidos se equiparon con la electrónica para el control de varios sistemas y diagnosticar fallos del coche, con el objetivo de reducir al mínimo la contaminación. Esto se produjo al aprobarse la Ley de Aire Limpio y el establecimiento de la Agencia de Protección Medioambiental (EPA) en 1970. Esta electrónica variaba entre los fabricantes y el año del modelo.

En la década de los 80's, a nivel global, se iniciaron una serie de movimientos sociales interesantes de concienciación sobre temas ecológicos y ambientales. Estos movimientos ponían sobre la mesa problemas como la contaminación ambiental, el deterioro de la capa de ozono, así como el calentamiento global y su impacto en el clima. En este contexto, el estado norteamericano de California se vio fuertemente afectado por la contaminación atmosférica debido a los bajos precios del combustible, el elevado nivel de vida y la alta densidad de población. Esto constituyó el arranque para la promulgación de normativas estrictas sobre gases de escape y consumo de vehículos a nivel mundial.

En 1988, la EPA y Junta de Recursos del Aire de California (CARB) ordenó que todos los fabricantes de vehículos incluyesen a partir de 1991 la auto-diagnosis en los vehículos y la Sociedad de Ingenieros Automotrices (SAE) estableció un conector estándar y un conjunto de señales de prueba de diagnóstico. La EPA adaptó la mayoría de estándares y recomendaciones de diagnóstico de la SAE. El sistema de diagnóstico a bordo original (que desde entonces se conoce como OBDI) era relativamente simple y solo monitorizaba el sensor de oxígeno, el sistema EGR (Recirculación de Gases de Escape), el sistema de suministro de combustible y el módulo de control del motor, de modo que resultó evidente que se necesitaría un sistema más sofisticado.

La CARB finalmente desarrolló estándares para el sistema OBD de próxima generación, que se propuso en 1989 y se conoció como OBDII. Los nuevos estándares requerían una incorporación gradual a partir de 1994 y a los fabricantes de automóviles se les dio hasta el año 1996 a partir del cual era un requisito legal para automóviles nuevos en California disponer del sistema OBDII. De forma similar, se incorporaron estándares

con las enmiendas de la Ley Federal de Aire Limpio en 1990, que también requería que todos los vehículos en el resto de estados estuvieran equipados con OBDII para 1996.

La Unión Europea (UE) declaró en 1998, según la Directiva 98/69/CE, que todos los vehículos vendidos en la UE debían de disponer de EOBD, versión europea de OBDII. Más en concreto, se hizo obligatorio en los automóviles de gasolina a partir del año 2001, en los vehículos diesel a partir del año 2003, así mismo vehículos de carga pesados a partir del año 2005.

Cabe destacar, regresando al año 1986, la introducción del protocolo de comunicaciones CAN (Controller Area Network) desarrollado inicialmente por la empresa alemana Robert Bosch GmbH. La idea principal detrás del desarrollo era la reducción drástica de cableado, aminorando así el coste, complejidad y peso en los automóviles. Con este sistema, se pasa de tener un cable dedicado para cada dispositivo electrónico, a tener un solo bus recorriendo el vehículo. La industria automotriz adoptó rápidamente CAN y, en 1993, se convirtió en el estándar internacional conocido como ISO 11898 tras su publicación por parte de la Organización Internacional para la Estandarización (ISO).

Para mediados de la década del 2000, en los automóviles particulares había básicamente 4 diferentes estándares OBD: SAE J1850 (FORD – PWM GM - VPW), ISO9141-2 (Chrysler, Europa y Asia), ISO14230 (KWP2000, Asia) e ISO 15765 (CAN BUS). A partir de 2008 se hizo obligatorio que el único protocolo a utilizar sea el ISO 15765 – 4 [10] / SAE J2480, el cual es una variante del estándar CAN a 500 kbps. Este es el estándar en uso hoy en día.

3.2 Terminología

Es importante conocer en primer lugar los conceptos relacionados con el sistema OBD para posteriormente entender el funcionamiento y la función que desempeña cada uno de ellos en el diagnóstico de vehículos.

3.2.1 Unidad de Control Electrónico/Motor (ECU)

Es un sistema embebido normalmente conectado a una serie de sensores que le proporcionan información y actuadores que ejecutan sus comandos. La ECU puede referirse a un solo módulo o una colección de módulos. Estos son los cerebros del vehículo que supervisan y controlan muchas funciones del automóvil. Pueden ser estándar del fabricante, reprogramables o tener la capacidad de conectarse en cadena para múltiples funciones. Algunos de los tipos de ECU más comunes incluyen:

- **Módulo de control del motor (ECM).** Controla los actuadores del motor, lo que afecta aspectos como el tiempo de encendido, la relación aire - combustible y el ralentí.
- **Módulo de control del vehículo (VCM).** Controla el rendimiento del motor y del vehículo.
- **Módulo de control de la transmisión (TCM).** Controla la transmisión, incluidos elementos como la temperatura del fluido de la transmisión, la posición del acelerador y la velocidad de las ruedas.
- **Módulo de control del tren motriz (PCM).** Controla el tren motriz del vehículo. Generalmente, es una combinación de un ECM y un TCM.
- **Módulo de control de freno electrónico (EBCM).** Controla y lee datos del sistema de frenos anti-bloqueo (ABS).
- **Módulo de control de la carrocería (BCM).** Controla las características de la carrocería del vehículo, como ventanas eléctricas, asientos eléctricos, etc.

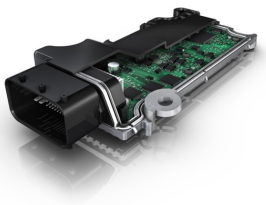


Figura 3.1 Modulo de control de la transmisión (TCM).

3.2.2 Conector de Enlace de Datos (DLC)

También conocido como conector OBDII, es el puerto de conexión de diagnóstico de 16 pines en vehículos. Se utiliza para conectar una herramienta de escaneo con los módulos de control. Generalmente se encuentra

debajo del panel de instrumentos en el lado del conductor o en algún lugar fácilmente accesible desde el asiento del conductor sin el uso de herramientas para acceder a él.

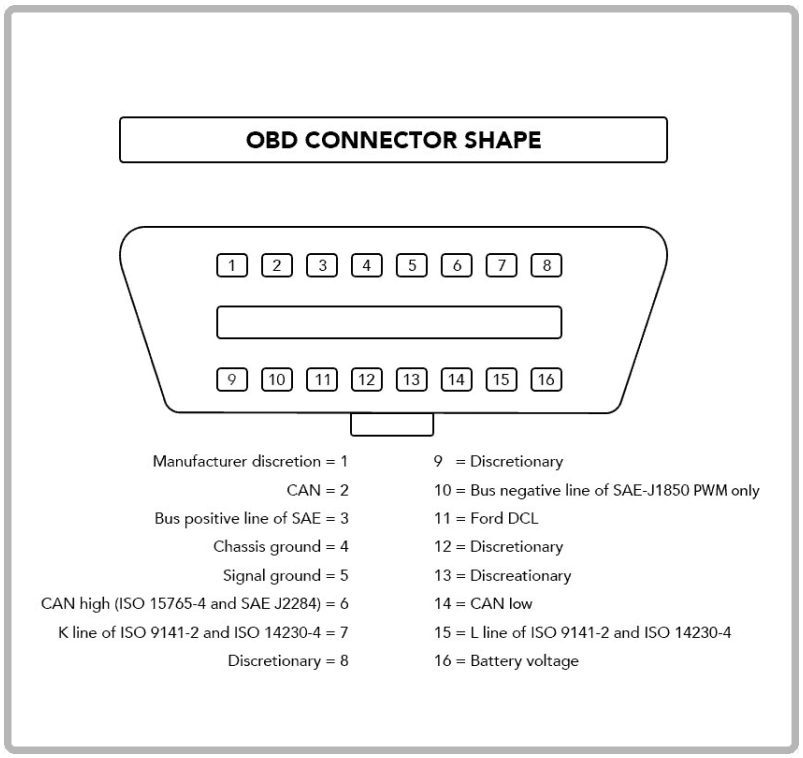


Figura 3.2 Forma del conector OBDII.

La utilización de los pines de este conector depende del protocolo utilizado por el vehículo. En la Figura 3.3 se muestran los pines que deben de utilizarse para el protocolo más usado actualmente en nuevos vehículos ISO 15765-4/SAE J2480 (CAN).

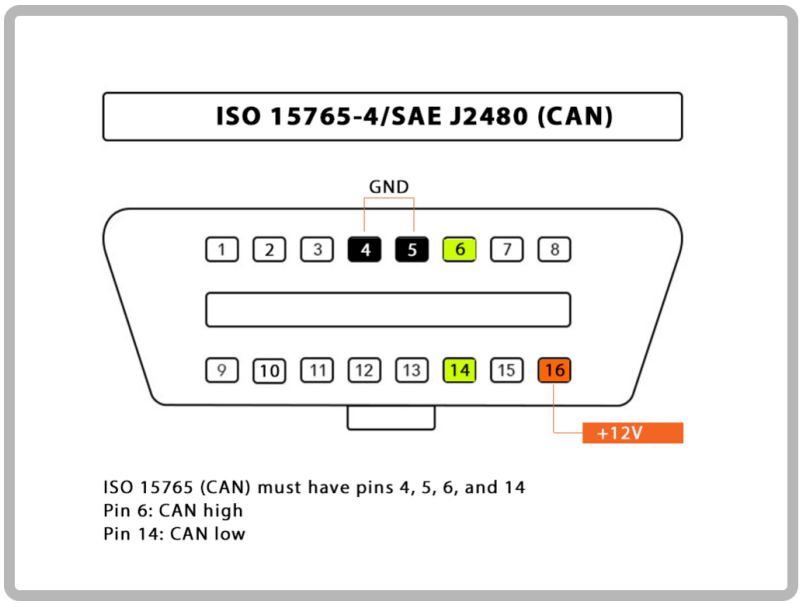


Figura 3.3 DLC para el protocolo ISO 15765-4/SAE J2480 (CAN).

3.2.3 Bus CAN

Un bus de comunicaciones CAN (Controller Area Network) es un bus serie que sirve para la comunicación entre todas las unidades de control del vehículo. Con esta red de comunicaciones toda la información de sensores y actuadores del vehículo puede ser compartida entre todas las unidades utilizando solo dos cables, reduciendo en gran medida los metros de cable necesarios, costes, peso del vehículo y probabilidad de averías.

Está compuesto por dos líneas, CAN High y CAN Low (CAN-H y CAN-L) y las unidades de control que se comunican mediante estas líneas.

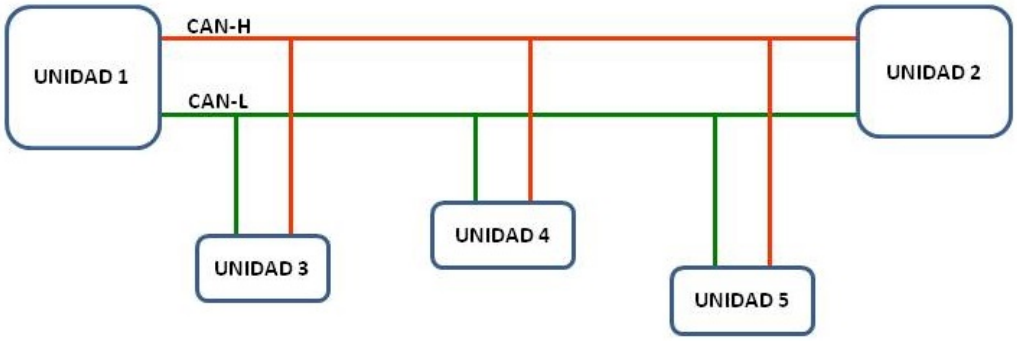


Figura 3.4 Esquema básico del bus CAN entre ECUs.

En un sistema básico, las líneas CAN llegarán directamente al conector DLC, como se puede observar en la Figura 3.5.

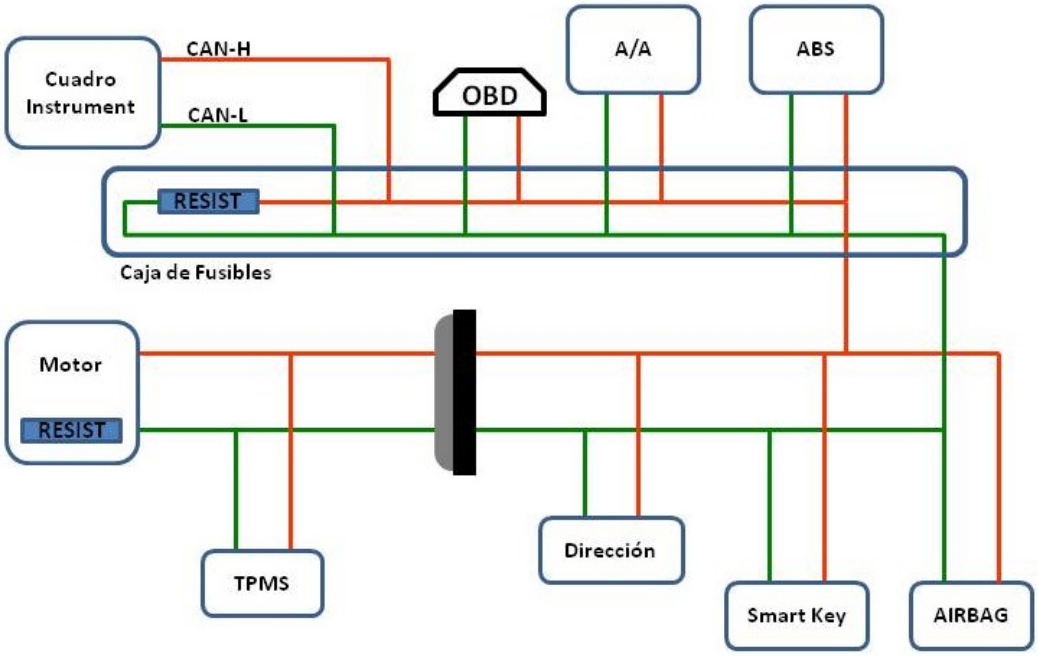


Figura 3.5 Esquema eléctrico genérico de un vehículo con bus CAN.

3.2.4 ELM327

Es un microcontrolador programado, producido por la empresa ELM Electronics, para interpretar diferentes protocolos OBD en vehículos. Los distintos métodos de conexión de este dispositivo con una herramienta de diagnóstico o software informático son por USB, RS-232, Bluetooth o WiFi. Este dispositivo se conecta a la interfaz OBDII o DLC, permitiendo así el envío y recepción de mensajes con las distinas ECUs del vehículo.

Figura 3.6 ELM37 con interfaz Bluetooth.

3.2.5 Códigos de Diagnóstico de Problemas (DTC)

Estos códigos se utilizan para describir dónde está ocurriendo un problema en el vehículo y están definidos por la SAE. Estos códigos pueden ser genéricos o exclusivos del fabricante del vehículo. Estos códigos toman el siguiente formato:

XXXXX

- La primera unidad identifica el tipo de código de error:
  - **P**XXXX. Tren motriz (powertrain). Incluye el motor, la transmisión y todos los accesorios asociados.
  - **B**XXXX. Carrocería (body). Se trata de piezas que se encuentran principalmente en la zona de los pasajeros del vehículo.
  - **C**XXXX. Chasis (chassis). Cubre sistemas mecánicos y funciones como dirección, suspensión y frenado.
  - **U**XXXX. No definido (undefined). Integración de redes y vehículos. Funciones que son administradas y compartidas por los sistemas informáticos a bordo.
- El segundo dígito muestra si el código es exclusivo del fabricante o no:
  - X**0**XXX. Es un código específico del fabricante.
  - X**1**XXX. Es un código estandarizado por la SAE, también conocido como código genérico.
  - X**2**XXX/X**3**XXX. Dependiendo del tipo de error puede ser código específico del fabricante, estandarizado por la SAE o reservado.
- El tercer dígito denota el sistema particular del vehículo que tiene el fallo:
  - XX**0**XX/XX**1**XX/XX**2**XX. Sistema de mediciones de aire y combustible (circuito inyector).
  - XX**3**XX. Sistema de encendido.
  - XX**4**XX. Sistema de controles auxiliares de emisiones.
  - XX**5**XX. Sistemas de control de velocidad, ralentí del vehículo y entradas auxiliares.
  - XX**6**XX. Sistemas informáticos y salidas auxiliares.
  - XX**7**XX/XX**8**XX/XX**9**XX. Sistemas de transmisión (caja de cambios).
  - XX**A**XX. Sistemas de propulsión híbrida.
  - XX**B**XX-XX**F**XX. Reservado según SAE J2012: Diagnostic Trouble Code Definitions [11].
- Los dos últimos dígitos muestran el código del fallo específico: xxx**00**-xxx**99**: se basan en los sistemas definidos en el tercer dígito para concretar el fallo.

3.2.6 Identificador de parámetros (PID)

Son códigos utilizados para solicitar datos a un vehículo a través de OBD-II como pueden ser velocidad del vehículo, RPM del motor, DTC, etc. El estándar OBD-II SAE J1979: E/E Diagnostic Test Modes [12] define diez modos de operación que se pueden observar en la Tabla 3.1

Las respuestas a cada uno de los PIDs se reciben en números binarios o hexadecimales. Este último tipo de comunicación es el que se describe a continuación.

Los bytes de respuesta se representan con las letras A, B, C, etc. A es el byte más significativo. Los bits de cada byte se representan del más significativo al menos con los números del 7 al 0:

Tabla 3.1 Modos de operación PIDs.

Modo (HEX)	Descripción
01	Muestra los parámetros disponibles
02	Muestra los datos almacenados por un evento
03	Muestra los DTC
04	Borra los datos almacenados, incluyendo los DTC
05	Resultados de la prueba de monitorización de sensores de oxígeno (sólo aplica a vehículos sin comunicación CAN)
06	Resultados de la prueba de monitorización de componente del sistema (resultados de la prueba de monitorización de sensores de oxígeno en vehículos con comunicación CAN)
07	Muestra los DTC detectados durante el último ciclo de conducción o el actual
08	Solicitud de control de sistema a bordo, prueba o componente
09	Solicitud de información del vehículo como el Número de Identificación del Vehículo (VIN)
0A	Códigos DTC permanentes (borrados)

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

Figura 3.7 Formato de los bits de respuesta de los PIDs.

Cada PID en concreto espera unos bytes de respuesta predeterminados. Convirtiendo los valores A, B, C, etc, de hexadecimal a decimal y aplicando posteriormente una fórmula específica para cada PID, podemos obtener el dato seleccionado.

3.2.7 Luz indicadora de mal funcionamiento (MIL)

Es un indicador que se utiliza para denotar un mal funcionamiento y se encuentra en la consola de instrumentos de la mayoría de los automóviles. El indicador tiene dos formas de funcionamiento: se ilumina de forma fija para indicar un fallo menor y parpadea repetidamente cuando se detecta un fallo grave para alertar al conductor sobre el peligro de daños severos del motor o del catalizador. Cuando la MIL está encendida, la unidad de control del motor almacena un DTC relacionado con el mal funcionamiento, que se puede recuperar, aunque en muchos modelos esto requiere el uso de una herramienta de escaneo.

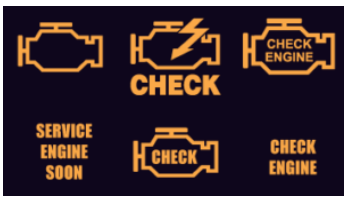


Figura 3.8 Distintas representaciones de una MIL.

3.3 Funcionamiento

En primer lugar, se explicará cómo se comunican internamente las ECUs, para posteriormente entender cómo se lleva a cabo el intercambios de mensajes con la interfaz OBDII.

3.3.1 CAN

Tal y como se indicó en el apartado de terminología dedicado al bus CAN, éste utiliza dos cables dedicados para la comunicación. El controlador CAN está conectado a todos los componentes de la red a través de

estos dos cables. Cada nodo de red tiene un identificador único y sólo responde cuando detecta su propio identificador en las tramas CAN. Además, están conectados en paralelo, por ello los nodos individuales se pueden eliminar de la red sin afectar al resto.

Respecto a la transmisión de datos por el bus en la capa física existen dos estados: dominante y recesivo. En estado recesivo, los dos cables trenzados del bus se encuentran al mismo nivel de tensión , mientras que en estado dominante hay una diferencia de tensión entre CAN_H y CAN_L de al menos 1,5 V y de máximo 3 V. La comunicación se basa en un diferencial de voltaje entre las dos líneas de bus, en lugar de tensión absoluta, proporcionando protección frente a interferencias electromagnéticas. En la especificación CAN un bit dominante equivale al valor lógico 0 y un bit recesivo equivale al valor lógico 1.

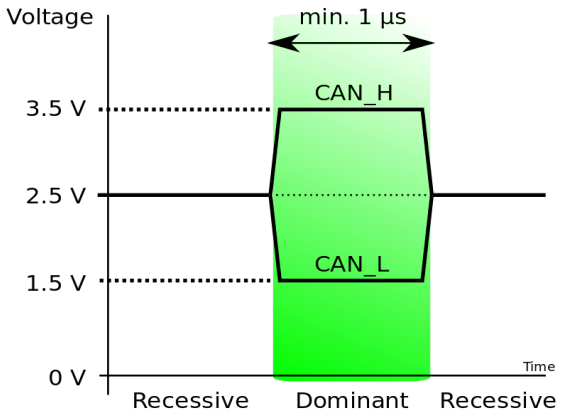


Figura 3.9 Niveles de tensión del bus CAN.

En relación a la capa de enlace se define el método de acceso al medio así como los tipos de tramas para el envío de mensajes. Cuando dos nodos intentan transmitir bits diferentes se denomina colisión y el valor del bit dominante prevalece sobre el valor del bit recesivo. En ese caso el nodo que intentaba transmitir el valor recesivo detecta la colisión y pasa a modo pasivo dejando de transmitir para escuchar lo que transmite el otro nodo. En la Figura 3.10 se muestra una trama CAN estándar con un identificador de 11 bits (CAN 2.0A), que es el tipo utilizado en la mayoría de los automóviles. Los campos de la trama son los siguientes:

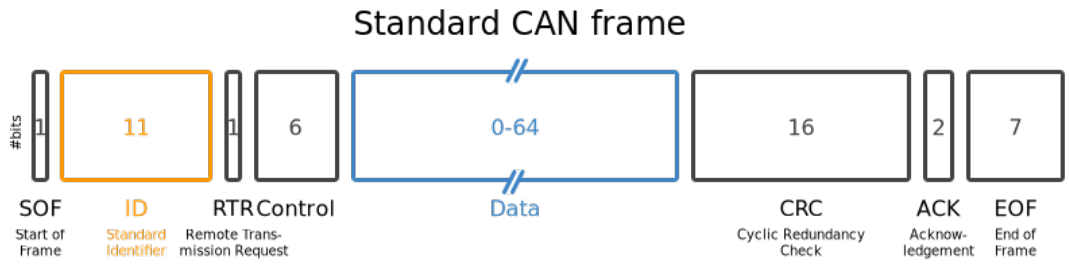


Figura 3.10 Trama CAN estándar.

- **SOF.** El inicio de la trama es un ’0 dominante’para decirle a los otros nodos que un nodo tiene la intención de hablar.
- **ID.** El ID es el identificador de la trama. Los valores más bajos tienen mayor prioridad.
- **RTR.** La Solicitud de Transmisión Remota indica si un nodo envía datos (0 dominante) o solicita datos específicos de otro nodo (1 recesivo).
- **Control.** El control contiene el Bit de Extensión del Identificador (IDE) que es un ’0 dominante’para 11 bits y un bit reservado. También contiene el Código de Longitud de Datos (DLC) de 4 bits que especifica la longitud de los bytes de datos a transmitir (0 a 8 bytes).
- **Data.** Los datos contienen los bytes de datos, también conocidos como carga útil, que incluyen señales CAN que se pueden extraer y decodificar para obtener información.
- **CRC.** La verificación de redundancia cíclica se utiliza para garantizar la integridad de los datos.
- **ACK.** Indica si el nodo ha reconocido y recibido los datos correctamente.

- **EOF.** marca el final de la trama CAN.

En este punto se pueden observar las diferencias y la relación entre OBDII y CAN. El estándar OBDII especifica un conjunto de 5 protocolos de más alto nivel. El protocolo definido en ISO 15765 necesita el protocolo CAN en las capas de enlace y física para funcionar. Además, desde 2008, el bus CAN (ISO 15765) ha sido el protocolo obligatorio para OBD2 en todos los automóviles, lo que básicamente elimina los otros 4 protocolos con el tiempo. En la Figura 3.11 se puede observar la diferencia en relación al modelo OSI. Hay que tener en cuenta que ISO 15765 se refiere a un conjunto de restricciones aplicadas al estándar CAN, este último definido en ISO 11898.

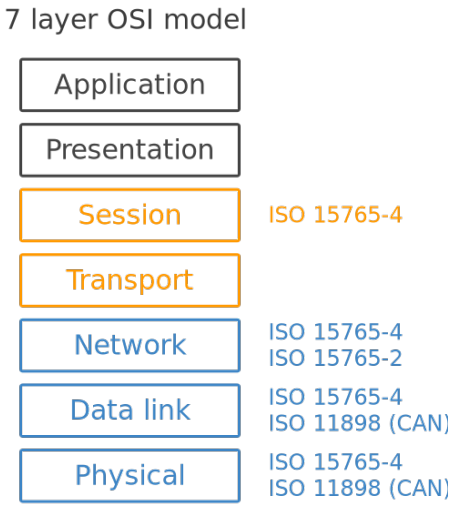


Figura 3.11 ISO 15765-4 (OBDII CAN BUS) e ISO 11898(CAN) en el modelo OSI.

3.3.2 OBDII

Una vez detallada la comunicación interna del vehículo, podemos centrarnos en la comunicación con la interfaz OBD con ayuda del dispositivo ELM327.

Para comenzar a obtener datos OBDII, es útil comprender los conceptos básicos de la estructura del mensaje OBDII sin procesar. Al igual que una trama CAN, un mensaje OBD2 se compone de un identificador y datos. Así mismo, los datos se dividen en Modo, PID y bytes de datos (A, B, C, D) como se muestra en la Figura 3.12.

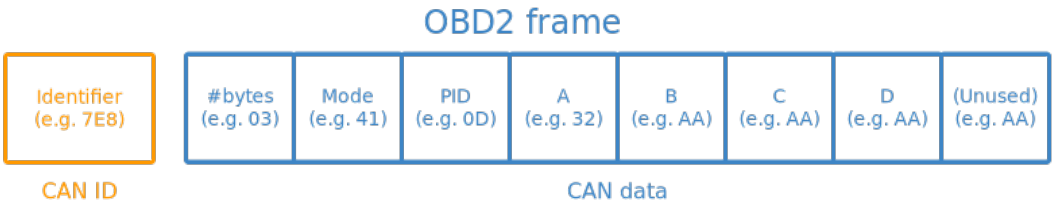


Figura 3.12 Trama OBDII.

El significado de cada campo se explica a continuación:

- **Identificador.** Para mensajes OBD2, el identificador estándar es de 11 bits y se utiliza para distinguir entre "mensajes de solicitud" (ID 7DF) y "mensajes de respuesta" (ID 7E8 a 7EF). Normalmente el ID 7E8 de respuesta será de la ECU principal.
- **Longitud.** Es la longitud en número de bytes de los datos restantes (03 a 06).
- **Modo.** Para solicitudes, será entre 01-0A y son los especificados en la Tabla 3.1. Para las respuestas, el 0 se reemplaza por 4 (es decir, 41, 42,..., 4A).
- **PID.** Para cada modo, existe una lista de PID OBD2 estándar. Cada PID tiene una descripción y algunos tienen un mínimo/máximo específico y una fórmula de conversión.

- **A, B, C, D.** Estos son los bytes de datos en hexadecimal, que deben convertirse a forma decimal antes de que se utilicen en los cálculos de la fórmula del PID. El último byte de datos (después de Dh) no se utiliza.

Un ejemplo de un mensaje de solicitud/respuesta para el PID 'Velocidad del vehículo'con un valor de 50 km/h podría ser el siguiente:

Solicitud: 7DF 02 01 0D 55 55 55 55 55
Respuesta: 7E8 03 41 0D 32 AA AA AA AA

- **Identificador.** Identificador de la solicitud funcional (7DF) y el identificador físico de la ECU que responde (7E8). En este caso, fue la ECU1 la que respondió.
- **Longitud.** En este ejemplo es 02 para la solicitud (ya que los datos son 01 y 0D), mientras que para la respuesta es 03, ya que los datos son 41, 0D y 32. 55 y AA reflejan bytes sin utilizar, por lo que no se tienen en cuenta en la longitud.
- **Modo.** En este caso se ha utilizado el Modo 01 en la solicitud, ya que corresponde con el modo para mostrar los valores de los parámetros del vehículo, en este ejemplo, la velocidad. Para la respuesta, el 0 se remplaza por un 4, con lo se obtiene el valor 41.
- **PID.** El PID 0D corresponde con la velocidad del vehículo.
- **A, B, C, D.** La fórmula para la velocidad es simplemente A, lo que significa que el byte de datos A (que está en hexadecimal) se convierte a decimal para obtener el valor y su unidad es km/h (32 en hexadecimal son 50 km/h en decimal).

Otro ejemplo significativo son los mensajes de respuesta multilínea. Esto quiere decir, que ante una solicitud se obtiene más de un mensaje de respuesta. Un tipo de respuesta de este tipo se obtiene al solicitar el Número de Identificación del Vehículo (VIN) de 17 dígitos con el modo 09 y PID 02, obtendríamos lo siguiente en un vehículo con CAN:

Solicitud: >0902
Respuesta:
014
0: 49 02 01 31 44 34
1: 47 50 30 30 52 35 35
2: 42 31 32 33 34 35 36

Con el formato CAN activado, se agrega un dígito hexadecimal (de 0 a F y luego se repite) con dos puntos (':'), al inicio de cada línea, que ayuda a reensamblar los datos. La primera línea de la respuesta indica que hay '014'bytes en hexadecimal de información en total que convertido a decimal son 20 bytes, que concuerdan con los 6 + 7 + 7 bytes de las 3 líneas. Los dos primeros bytes "49 02" indican que se trata de una respuesta a la solicitud "0902", y el tercer byte "01" indica el número de elementos de datos de la respuesta, en este caso, el vehículo sólo puede tener un VIN. Los 17 dígitos restantes representan los códigos ASCII para los dígitos del VIN:

31	44	34	47	50	30	30	52	35	35	42	31	32	33	34	35	36
1	D	4	G	P	0	0	R	5	5	B	1	2	3	4	5	6

Con el VIN podemos extraer información del vehículo, como la identificación del fabricante, factores técnicos, año en el que se fabricó el coche, etc.

3.3.3 Interpretación DTC

El uso más común e importantante para este proyecto, es la obtención de DTC actuales en el vehículo. Para ello, se necesita realizar una solicitud del Modo 03. En primer lugar, se debe conocer el número de DTC actualmente almacenados, esto se hace con una solicitud del Modo 01 PID 01. Un ejemplo de respuesta podría ser la siguiente:

Solicitud: >0101
Respuesta: 41 01 81 07 65 04

Analizando la respuesta tenemos:

- **"41 01":** Respuesta a la petición "01 01".
- **"81":** Este byte tiene una doble función: el bit más significativo se usa para indicar que la MIL se ha encendido con uno de los códigos almacenados, mientras que los otros 7 bits proporcionan el número real de DTC actuales. Por tanto, para calcular el número de DTC cuando la MIL está encendida, hay que restar 128 (o 80 hexadecimal) a este byte de respuesta. En este caso, habría un único DTC almacenado (81 - 80 = 1) y fue el que provocó la activación de la MIL.
- **"076504":** Proporcionan información sobre los tipos de test soportados. Se puede obtener más información de estos test en el documento SAE J1979 [12].

Tras conocer la existencia de un DTC en el vehículo, el siguiente paso es realizar una solicitud con el Modo 03 sin PID. Una respuesta podría ser:

Solicitud: >03
Respuesta: 43 01 33 00 00 00 00

Examinando los byes de respuesta podemos observar:

- **"43":** Respuesta a la petición "03".
- **"01 33 00 00 00 00":** Estos 6 bytes deben ser leídos en parejas para mostrar los DTC ("0133", "0000" y "0000"). El bit más significativo de cada DTC contiene información adicional que puede interpretarse según la Tabla 3.2. La primera columna indica que si se recibe este bit, hay que reemplazarlo por los caracteres establecidos en la segunda columna. Para este ejemplo, el DTC "0133", el primer dígito (0) sería remplazado por P0, cuyo resultado sería el DTC "P0133" que corresponde con "respuesta lenta del circuito sensor (O2) 1 de oxígeno del banco 1". El resto de bytes "0000" indican que no hay más DTC almacenados.

Tabla 3.2 Interpretación bit más significativo DTC.

0	P0	Powertrain Codes - SAE defined
1	P1	Powertrain Codes - manufacturer defined
2	P2	Powertrain Codes - SAE defined
3	P3	Powertrain Codes - jointly defined
4	C0	Chassis Codes - SAE defined
5	C1	Chassis Codes - manufacturer defined
6	C2	Chassis Codes - manufacturer defined
7	C3	Chassis Codes - reserved for future
8	B0	Body Codes - SAE defined
9	B1	Body Codes - manufacturer defined
A	B2	Body Codes - manufacturer defined
B	B3	Body Codes - reserved for future
C	U0	Network Codes - SAE defined
D	U1	Network Codes - manufacturer defined
E	U2	Network Codes - manufacturer defined
F	U3	Network Codes - reserved for future

Para el protocolo ISO 15765-4 (CAN) la respuesta es muy similar, pero añade un byte extra en segunda posición indicando el número de DTC que contiene la respuesta.

3.3.4 Comandos AT

Los comandos AT o Hayes son instrucciones que conforman un lenguaje de comunicación que inicialmente se desarrollaron para configurar y parametrizar módems. El dispositivo ELM327 esencialmente interpreta estos comandos pudiendo modificar su comportamiento como desactivando el "echo" de las respuestas, eliminar los bytes de cabecera de la respuesta, configurar los valores por defecto, eliminar los espacios entre bytes de la respuesta, utilizar el protocolo automático permitido por el vehículo...

Básicamente, al iniciar la comunicación con el ELM327, éste envía un mensaje de respuesta con su nombre y la versión de software de éste. Además, envía el símbolo ">" que es el prompt para indicar que está a la espera de órdenes:

```
ELM 321 v2.1
>
```

Unos de los comandos más simples para verificar la comunicación es enviar un Reset con el comando "AT Z", que obliga al ELM327 a realizar un reinicio completo, volviendo todas las configuraciones a sus valores predeterminados. Todos los comandos deben terminar con un carácter retorno de carro 0x0D (ên C). Si el comando es un cambio de configuración como "AT S0" (elimina espacios entre bytes en la respuesta), la respuesta será simplemente un "OK" indicando que fue correctamente ejecutado. Algunos de los comandos permiten el paso de números como argumentos que deben estar en hexadecimal. Además, para los comandos del tipo ON/OFF, el segundo carácter es el número 1 (ON) o el número 0 (OFF). En el siguiente ejemplo, se puede ver que el tipo de configuración es OFF:

```
> AT S0
OK
```

En la Tabla 3.3 se pueden observar algunos de los comandos utilizados en este proyecto y la descripción de ellos.

Tabla 3.3 Comandos AT utilizados.

Comando	Descripción
AT D (set all to Defaults)	Se utiliza para configurar las opciones a sus valores predeterminados.
AT DP (Describe the current Protocol)	Se utiliza para conocer qué protocolo OBDII está actualmente configurado.
AT DPN (Describe the Protocol by Number)	Similar a comando DP, pero devuelve el número que representa el protocolo actual.
AT E (Echo off or on)	Se utiliza para controlar si los caracteres enviados se retransmiten de regreso al ELM327 (eco). El valor por defecto es E1 (activado).
AT H (Headers off or on)	Se utiliza para controlar si los bytes de cabecera se muestran o no en las respuestas del vehículo. El valor por defecto es H0 (desactivado).
AT S (printing of Spaces off or on)	Se utiliza para controlar si se insertan o no caracteres de espacio en la respuesta de la ECU. Por defecto, su valor es S1 (activado).
AT SP (Set Protocol to h)	Se utiliza para configurar el ELM327 para que funcione usando el protocolo especificado por h, y también para guardarlo como el nuevo predeterminado.
AT Z (reset all)	Este comando hace que ELM327 realice un reinicio completo (apagado y encendido), volviendo todas las configuraciones a sus valores predeterminados.

La utilización de los comandos AT, nos permitirá una correcta inicialización de la comunicación y la configuración de parámetros que nos permitirá obtener una mayor eficiencia en la comunicación con el vehículo.

4 Especificación de requisitos del sistema (ERS)

La elaboración de la Especificación de Requisitos Software (ERS) se ha estructurado inspirándose en las directrices dadas por el estándar "ISO/IEC/IEEE 29148:2011 Systems and software engineering — Life cycle processes — Requirements engineering" [13].

4.1 Introducción

4.1.1 Propósito del sistema

El propósito general de la librería desarrollada obd2-bluetooth es facilitar a ingenieros y desarrolladores de software una herramienta que abstrae la comunicación OBD con el vehículo, permitiendo implementar fácilmente diferentes aplicaciones sin la necesidad de conocer el funcionamiento de la conexión a más bajo nivel.

4.1.2 Alcance del sistema

El sistema desarrollado no pretende ser una herramienta completa de control de todas las funcionalidades permitidas en la conexión OBD con el vehículo, sino que recoge las funcionalidades más importantes y utilizadas en la actualidad:

- Detección e identificación de DTC del vehículo.
- Obtención de valores de ciertos sensores predefinidos del vehículo.
- Acceso a información de identificación del vehículo como el VIN.
- Revisión del estado del vehículo con una serie de pruebas estandarizadas.

Además, con la ayuda de una serie de librerías y archivos de configuración los usuarios podrán establecer mensajes de notificación a un servidor externo para que en caso de fallo del vehículo pueda obtener información suplementaria como la localización actual del vehículo, si éste dispone de localizador GPS.

Por último, se desarrolla un servidor piloto para conseguir la persistencia de los datos enviados por el vehículo, con una sencilla interfaz gráfica para la visualización de errores.

4.1.3 Descripción general del sistema

4.1.3.1 Contexto del sistema

Se parte de la necesidad de obtener un software que nos permita implementar las funcionalidades OBD como la obtención de los DTC sin afectar al funcionamiento del vehículo y que pueda ejecutarse en dispositivos con pocos recursos. Esto permite desarrollar aplicaciones específicas para empresas con flota de vehículos de transporte con el objetivo de conocer el estado de ellos en todo momento, reducir costes y tiempo en la reparación.

4.1.3.2 Características de los actores

En esta sección se especifican los actores que se han identificado en los casos de uso, es decir, los diferentes tipos de usuarios y otros sistemas con los que interactúa el sistema a desarrollar.

Tabla 4.1 ACT-01: Usuario.

ACT-01	Usuario
Descripción	Representa a los desarrolladores que utilizarán la librería para el desarrollo de aplicaciones.
Anotación	Realizará las llamadas a las distintas funciones definidas en la librería.

Tabla 4.2 ACT-02: ELM327.

ACT-02	ELM327
Descripción	Representa al microcontrolador con interfaz bluetooth que interpreta los distintos protocolos OBD en vehículos.
Anotación	Recibe las peticiones enviadas por el usuario y las interpreta para enviar los mensajes correspondientes a la ECU. Del mismo modo, se encarga de enviar los mensajes de respuesta de la ECU.

Tabla 4.3 ACT-03: Servidor Web (BBDD).

ACT-03	Servidor Web (BBDD)
Descripción	Representa al servidor web junto con la BBDD que recibe la información solicitada al vehículo.
Anotación	Alojado en un servidor remoto y es necesario conocer su IP para el envío de datos.

Tabla 4.4 ACT-04: Archivos de configuración.

ACT-04	Archivos de configuración
Descripción	Representa la información de configuración externa a la librería pero que es necesaria para el correcto funcionamiento del sistema.
Anotación	Se trata de archivos de configuración para la conexión bluetooth, para la conexión con el servidor web y con los PIDS genéricos para la interpretación de los comandos OBD.

4.2 Descripción de subsistemas a desarrollar

Los subsistemas son agrupaciones lógicas de requisitos cuya finalidad es facilitar la comprensión de los mismos. Una buena división en subsistemas facilita el posterior diseño y desarrollo del producto final. En este caso, este sistema puede agruparse en en los siguientes subsistemas:

- **Gestión de la conexión con ELM327:** Subsistema encargado del descubrimiento del dispositivo bluetooth de ELM327 y su posterior conexión para poder inicializar el paso de mensajes OBD con el vehículo. Depende del subsistema de gestión de lectura de ficheros de configuración para obtener el nombre al que conectar del dispositivo bluetooth OBDII.
- **Gestión del envío y recepción de mensajes OBD:** Subsistema más importante del sistema. Gestiona la inicialización de la conexión OBDII con el envío de comandos AT, el envío de mensajes OBD, la recepción de estos y su posterior decodificación. Depende del subsistema de gestión de la conexión con ELM327 y a su vez, del subsistema de gestión de lectura de ficheros de configuración, ya que necesita cargar un fichero con los comandos AT y OBD.
- **Gestión de lectura de ficheros de configuración:** Subsistema que gestiona la lectura de dos ficheros de configuración. Por un lado, existe un fichero con información sobre las conexiones del sistema

(bluetooth y servidor web). Por otro lado, un fichero con información relativa a los comandos AT y OBD de vital importancia para la comunicación con ELM327.

- **Gestión de los datos:** Subsistema cuyo fin es el tratamiento de los datos obtenidos tras la decodificación del mensaje recibido. Gestiona tanto la impresión directa por consola de los datos como el envío de estos al servidor web con una BBDD que proporciona la persistencia de éstos.

4.3 Casos de uso

En esta sección, se describe las interacciones entre el sistema y los usuarios en los que éstos últimos realizan sus tareas de negocio utilizando los servicios ofrecidos por el sistema a desarrollar.

Se muestra inicialmente un diagrama de caso de uso general que engloba de forma resumida el sistema completo. En los próximos apartados se explica más detalladamente cada uno de los casos de uso de los diferentes subsistemas.

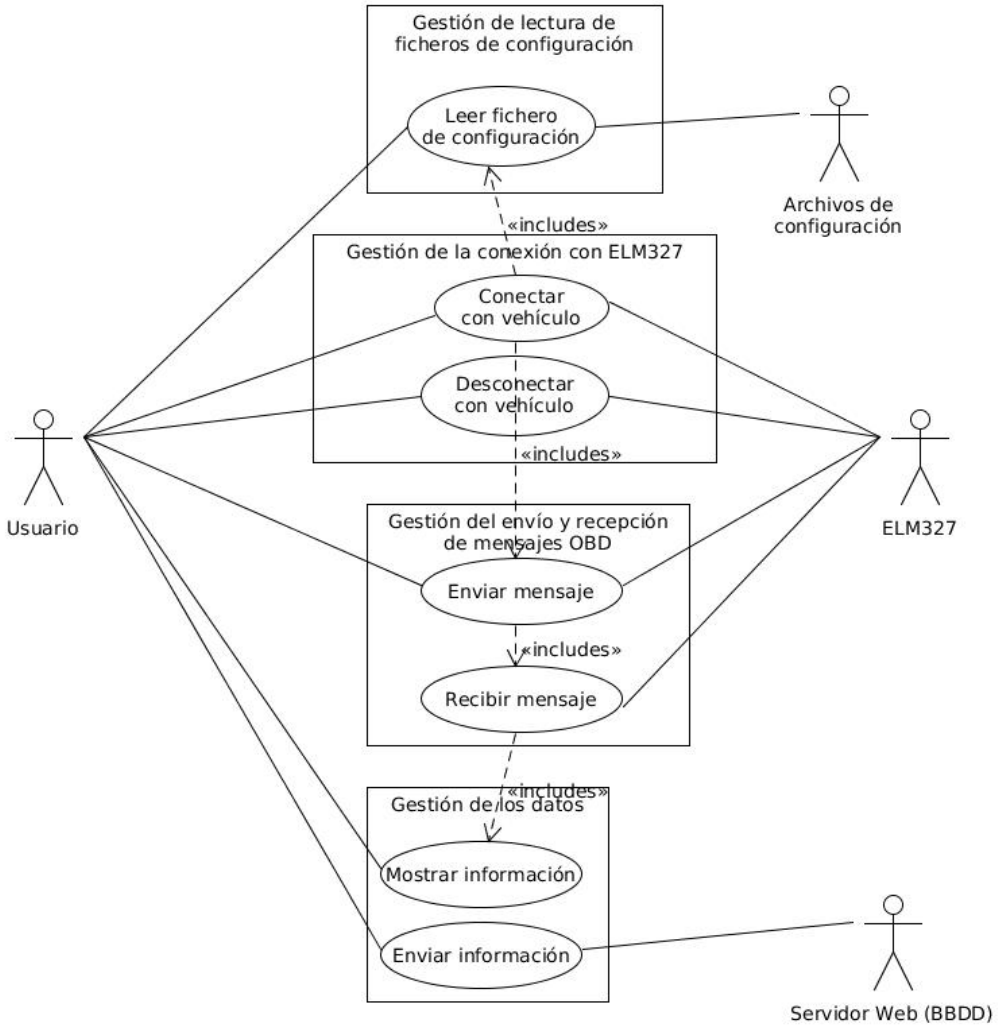


Figura 4.1 Diagrama de casos de uso general del sistema.

En la Figura 4.1 se puede observar las relaciones comentadas entre los subsistemas, cómo la necesidad de que al iniciar el proceso de conexión con el dispositivo ELM327, es necesario leer el fichero de PID's en formato JSON para poder realizar el intercambio de mensajes AT inicial.

4.3.1 Gestión de la conexión con ELM327

Se puede observar en la Figura 4.2 el diagrama de casos de uso correspondiente al subsistema de gestión de la conexión con ELM327. A continuación, se ofrece una descripción de cada uno de los casos de uso para este subsistema.

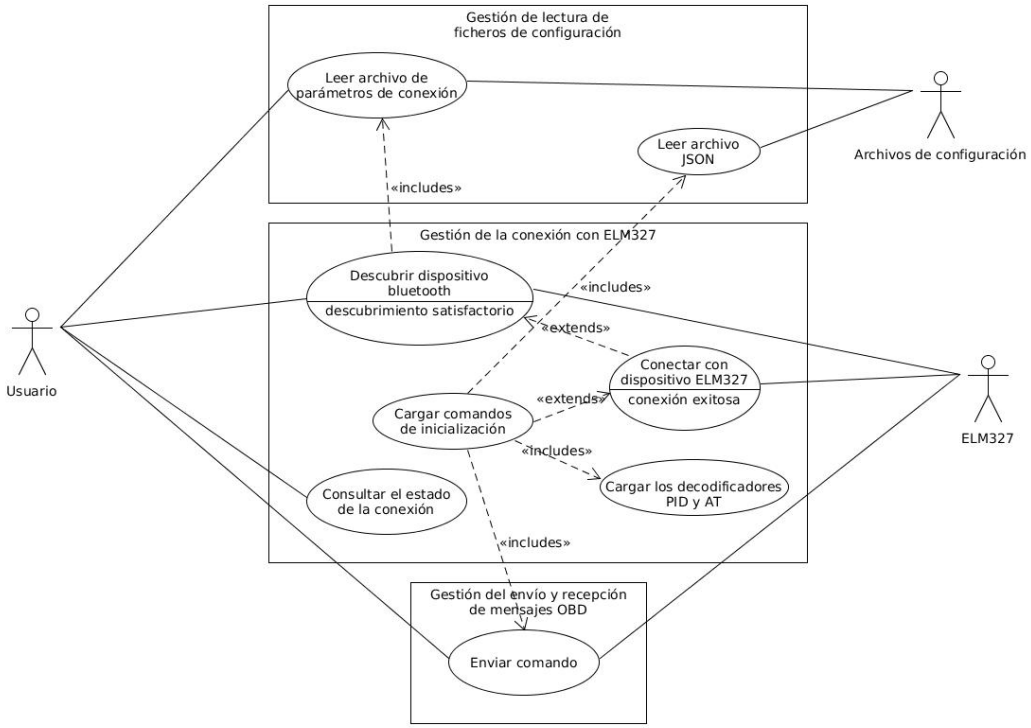


Figura 4.2 Diagrama de casos de uso gestión de la conexión ELM327.

Tabla 4.5 CU-01: Descubrir dispositivo bluetooth.

CU-01	Descubrir dispositivo bluetooth
Dependencias	(ACT-01) Usuario (ACT-02) ELM327 (CU-06) Leer archivo de parámetros de conexión
Descripción	Se inicia el proceso de descubrimiento del dipositivo bluetooth correspondiente a la interfaz bluetooth del ELM327. Este paso es necesario para iniciar la posterior conexión, ya que, en primer lugar, el dispositivo bluetooth tiene que estar disponible y visible. El nombre del dispositivo bluetooth que se utiliza en el descubrimiento se obtiene del archivo de configuración de parámetros de conexión.
Precondición	Existencia de un usuario con la necesidad de comunicarse con el vehículo y con el nombre del dispositivo bluetooth del ELM327, que usualmente es OBDII (CU-06).
Postcondición	Si se encuentra visible el dispositivo bluetooth se procede con (CU-02) la conexión con el dispositivo ELM327. En caso contrario, se finaliza la ejecución con error en el descubrimiento.

Tabla 4.6 CU-02: Conectar con dispositivo ELM327.

CU-02	Conectar con dispositivo ELM327
Dependencias	(ACT-02) ELM327 (CU-01) Descubrir dispositivo bluetooth
Descripción	Se procede a iniciar la conexión bluetooth con el ELM327 necesaria para el posterior envío de comandos.
Precondición	Un correcto descubrimiento bluetooth de OBDII habiendo obtenido su dirección para la conexión en este caso de uso.
Postcondición	En el caso de que se conecte correctamente, se cargan los comandos de inicialización (CU-03) que establecen los parámetros necesarios para un correcto envío de mensajes OBD al vehículo. En caso contrario, se finaliza la ejecución con error en la conexión.
Comentarios	La autenticación mediante PIN del dispositivo bluetooth no ha sido necesaria, ya que el dispositivo conecta con la librería sin exigir este dato, funcionando correctamente.

Tabla 4.7 CU-03: Cargar comandos de inicialización.

CU-03	Cargar comandos de inicialización
Dependencias	(CU-02) Conectar con dispositivo ELM327 (CU-04) Cargar los decodificadores PID y AT (CU-07) Leer archivo JSON (CU-08) Enviar comando
Descripción	El sistema se encarga de configurar una serie de parámetros del dispositivo ELM327 mediante el envío de comando AT y de realizar un chequeo de los PIDs disponibles en el vehículo junto a otros comandos OBD.
Precondición	Debe de existir una conexión bluetooth correcta con el dispositivo ELM327. Además, para realizar la inicialización se necesitan conocer los comandos AT y OBD a utilizar que se encuentran en el archivo JSON (CU-07) y cargar los decodificadores (CU-04) para poder traducir las respuestas obtenidas del vehículo.
Postcondición	Tras el envío de todos los comandos de inicio, el usuario puede comenzar a realizar el envío de mensajes específico que desee.

Tabla 4.8 CU-04: Cargar los decodificadores PID y AT.

CU-04	Cargar los decodificadores PID y AT
Descripción	El sistema carga una serie de decodificadores dependientes del tipo de datos que puede obtener de cada tipo de comando PID o AT, necesarios para traducir las respuestas en hexadecimal del vehículo en datos legibles por el humano.
Precondición	Conexión bluetooth correcta con el dispositivo ELM327.
Postcondición	Proceso de inicialización preparado para entender los mensajes de respuesta del vehículo.

Tabla 4.9 CU-05: Consultar el estado de la conexión.

CU-05	Consultar el estado de la conexión
Dependencias	(ACT-01) Usuario
Descripción	Muestra el estado de la conexión bluetooth tras el proceso de inicialización y permite al usuario actuar en el caso de que la conexión sea exitosa.
Precondición	El usuario debe de haber iniciado el proceso de descubrimiento para poder obtener datos sobre la conexión bluetooth.
Postcondición	El usuario conoce si la conexión es correcta o ha fallado.

Tabla 4.10 CU-06: Leer archivo de parámetros de conexión.

CU-06	Leer archivo de parámetros de conexión
Dependencias	(ACT-01) Usuario (ACT-04) Archivos de configuración
Descripción	Se encarga de la lectura de un fichero de configuración de parámetros de las conexiones bluetooth, temporizador, servidor web (IP y puerto) y GPS (en el caso de que exista en el sistema).
Precondición	El usuario debe de haber configurado este archivo correctamente en el escenario concreto de utilización.
Postcondición	El usuario puede iniciar el descubrimiento bluetooth con el nombre del dispositivo que aparece en el fichero de configuración de parámetros.

Tabla 4.11 CU-07: Leer archivo JSON.

CU-07	Leer archivo JSON
Dependencias	(ACT-04) Archivos de configuración
Descripción	Su función es la lectura de toda la información relacionada con los comandos OBD y AT que se encuentran en el archivo de PIDS con formato JSON. Este archivo de datos contiene información de cada comando como los bytes de respuesta, la cadena de texto a enviar, el decodificador que le corresponde, una breve descripción, un nombre por el que el usuario puede ejecutar el comando, sus unidades, el tipo de dato y, valores mínimo y máximo en el caso de que tenga.
Precondición	Conexión bluetooth correcta con el dispositivo ELM327.
Postcondición	Proceso de inicialización preparado para poder enviar los mensajes iniciales configurados.

Tabla 4.12 CU-08: Enviar comando.

CU-08	Enviar comando
Dependencias	(ACT-01) Usuario (ACT-02) ELM327 (CU-13) Recibir respuesta
Descripción	Se encarga del envío de un comando seleccionado o bien por el usuario, o bien en el proceso de inicialización, mediante bluetooth al dispositivo ELM327. Crea un hilo de ejecución para la recepción del mensaje de respuesta.
Precondición	Conexión bluetooth correcta con el dispositivo ELM327, decodificadores de mensajes PID y AT e información de comandos cargada correctamente.
Postcondición	Hilo de ejecución creado el cuál queda pendiente de la respuesta del dispositivo ELM327 al comando enviado.

4.3.2 Gestión del envío y recepción de mensajes OBD

Para el caso de la gestión del envío y recepción de mensajes OBD al vehículo, podemos observar en el diagrama de casos de uso Figura 4.3 las distintas opciones que dispone el usuario para consultar, además del proceso de envío y recepción de los mensajes de forma más detallada.

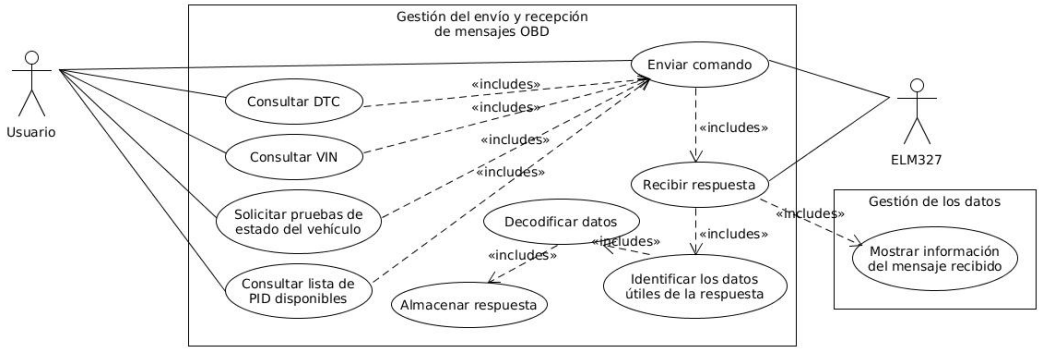


Figura 4.3 Diagrama de casos de uso gestión del envío y recepción de mensajes OBD.

Tabla 4.13 CU-09: Consultar DTC.

CU-09	Consultar DTC
Dependencias	(ACT-01) Usuario (CU-08) Enviar comando
Descripción	Se realiza el procedimiento de consulta por si el vehículo tiene algún DTC activo. En primer lugar, se utiliza el comando STATUS que indica si la MIL está encendida o no. En caso afirmativo, se obtiene el número de DTC activos y se utiliza el comando GET_DTC que permite identificar el DTC.
Precondición	Conexión bluetooth activada y finalizado el proceso de inicialización del dispositivo ELM327. Conocimiento por parte del usuario del uso de los comandos y funciones dedicadas para este fin.
Postcondición	Quedan almacenados en el sistema los DTC activos en el caso de que existan en el vehículo para que posteriormente puedan consultarse.

Tabla 4.14 CU-10: Consultar VIN.

CU-10	Consultar VIN
Dependencias	(ACT-01) Usuario (CU-08) Enviar comando
Descripción	Se realiza el procedimiento de consulta del número de identificación del vehículo (VIN) que permite la identificación del vehículo e información del modelo y fabricante entre otros datos. Se utiliza el comando GET_VIN para el envío del mensaje.
Precondición	Conexión bluetooth activada y finalizado el proceso de inicialización del dispositivo ELM327. Conocimiento por parte del usuario del uso de los comandos y funciones dedicadas para este fin.
Postcondición	Queda almacenado en el sistema el VIN en el caso de que esté disponible en el vehículo para que posteriormente pueda consultarse.

Tabla 4.15 CU-11: Solicitar pruebas de estado del vehículo .

CU-11	Solicitar pruebas de estado del vehículo
Dependencias	(ACT-01) Usuario (CU-08) Enviar comando
Descripción	Se realiza el procedimiento de pruebas para conocer el estado del vehículo. Se utiliza el comando STATUS para el envío del mensaje. Entre los sistemas comprobados según el tipo de motor del vehículo se encuentran los sistemas de combustible, de componentes integrales, de eficiencia del convertidor catalítico, de fugas de aire acondicionado y de sensores de oxígeno entre otros.
Precondición	Conexión bluetooth activada y finalizado el proceso de inicialización del dispositivo ELM327. Conocimiento por parte del usuario del uso de los comandos y funciones dedicadas para este fin.
Postcondición	Quedan almacenados en el sistema los valores de cada una de las pruebas realizadas (incorrecta o correcta) en los distintos sistemas del vehículo para una posterior consulta. También queda almacenada la información de si la MIL se encuentra encendida o apagada.

Tabla 4.16 CU-12: Consultar lista de PID implementados.

CU-12	Consultar lista de PID implementados
Dependencias	(ACT-01) Usuario (CU-08) Enviar comando
Descripción	Se realiza el procedimiento de consulta de PID implementados en el vehículo. Se utiliza los comandos con PID 00, 20, 40, 60, 80, A0 y C0 si existen entre los disponibles para el envío del mensaje. Estos PID específicos tienen la función de indicar si los próximos 32 PID están implementados en el vehículo pudiendo así conocer la lista completa.
Precondición	Conexión bluetooth activada y finalizado el proceso de inicialización del dispositivo ELM327. Conocimiento por parte del usuario del uso de los comandos y funciones dedicadas para este fin.
Postcondición	Quedan almacenados en el sistema los PID implementados en el vehículo pudiendo ser consultados posteriormente.

Tabla 4.17 CU-13: Recibir respuesta.

CU-13	Recibir respuesta
Dependencias	(ACT-02) ELM327 (CU-14) Identificar los datos útiles de la respuesta (CU-17) Mostrar información del mensaje recibido
Descripción	Hilo de ejecución creado al enviar un mensaje OBD que se mantiene a la espera de la recepción de los datos que envíe ELM327. Se encarga de mantener el socket de escucha abierto hasta que se haya recibido todos los mensajes que correspondan con un comando enviado.
Precondición	Un mensaje OBD ha tenido que ser enviado con anterioridad y ha tenido que crear un hilo de ejecución con información sobre el comando enviado.
Postcondición	El sistema obtiene el mensaje en hexadecimal enviado por el vehículo, estando disponbible para identificar los datos útiles de la respuesta.

Tabla 4.18 CU-14: Identificar los datos útiles de la respuesta.

CU-14	Identificar los datos útiles de la respuesta
Dependencias	(CU-15) Decodificar datos
Descripción	Proceso de filtrado de los bytes útiles de la respuesta del mensaje. Dependiendo del tipo de mensaje se obtendrán el número de bytes correspondientes para que pueda ejecutarse correctamente el proceso de decodificación.
Precondición	Recibido el mensaje OBD de respuesta completo.
Postcondición	Se obtienen los bytes de información útiles listos para ser decodificados.

Tabla 4.19 CU-15: Decodificar datos.

CU-15	Decodificar datos
Dependencias	(CU-16) Almacenar respuesta
Descripción	Proceso de decodificación de los bytes útiles de la respuesta. Se utiliza una función específica para cada comando enviado. Se especifica la correspondencia de comandos con su decodificador correspondiente en el archivo en formato JSON cargado en el proceso de inicialización del dispositivo ELM327.
Precondición	Bytes de información útil obtenidos.
Postcondición	Se obtiene el valor que se ha solicitado en la petición que puede ser almacenado.

Tabla 4.20 CU-16: Almacenar respuesta.

CU-16	Almacenar respuesta
Descripción	Se almacena en memoria el último resultado del comando enviado al ELM327 para que pueda ser consultado por el usuario.
Precondición	Información de la respuesta del vehículo ha tenido que ser decodificada.
Postcondición	Se obtiene el valor solicitado en la petición estando accesible por el usuario.

Tabla 4.21 CU-17: Mostrar información del mensaje recibido.

CU-17	Mostrar información del mensaje recibido
Descripción	El sistema muestra por consola o logs el intercambio de mensajes con la información enviada y recibida, pudiéndose ver en tiempo real las respuesta del vehículo.
Precondición	Se tiene que haber enviado y recibido algún mensaje al vehículo.
Postcondición	El usuario obtiene información en tiempo real de la comunicación con el vehículo.

4.3.3 Gestión de lectura de ficheros de configuración

Los casos de uso de este subsistema han sido detallados en los anteriores subsistemas. A continuación, en la Figura 4.4 se puede observar el diagrama de este subsistema.

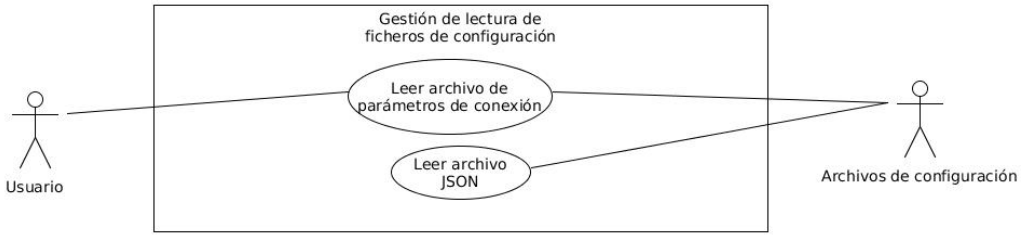


Figura 4.4 Diagrama de casos de uso gestión de lectura de ficheros de configuración.

4.3.4 Gestión de los datos

Por último,todo lo relacionado con obtener información de los datos del vehículo y gestionar su envío al servidor se puede observar en la Figura 4.5.

Tabla 4.22 CU-18: Mostrar DTC.

CU-18	Mostrar DTC
Dependencias	(ACT-01) Usuario
Descripción	Muestra al usuario información sobre el DTC detectado en el caso de que exista.
Precondición	Se debe de haber consultado los DTC antes.
Postcondición	El usuario puede observar los DTC del vehículo.

Tabla 4.23 CU-19: Mostrar VIN.

CU-19	Mostrar VIN
Dependencias	(ACT-01) Usuario
Descripción	Muestra al usuario información sobre el número de identificación del vehículo (VIN) obtenido.
Precondición	Se debe de haber consultado el VIN antes.
Postcondición	El usuario puede observar el VIN del vehículo.

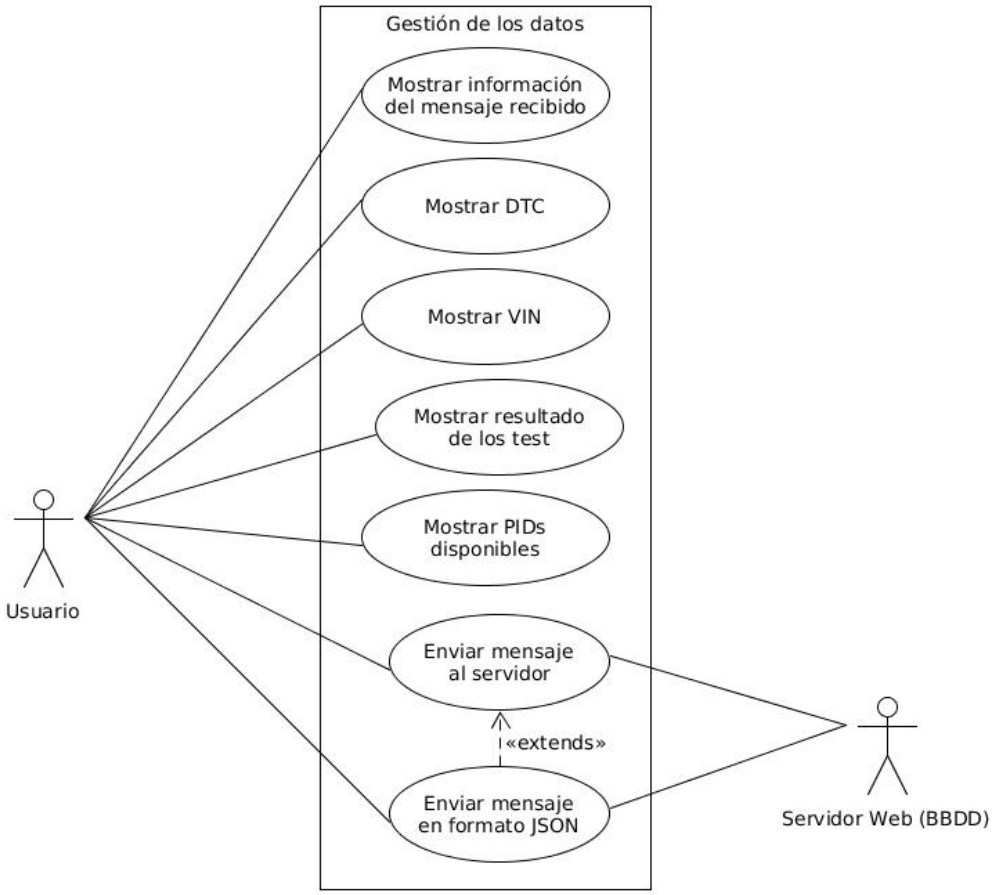


Figura 4.5 Diagrama de casos de uso gestión de los datos.

Tabla 4.24 CU-20: Mostrar resultado de los test.

CU-20	Mostrar resultado de los test
Dependencias	(ACT-01) Usuario
Descripción	Muestra al usuario información sobre las pruebas realizadas y el resultados de estas pruebas junto con el estado de la MIL.
Precondición	Se debe de haber consultado el estado del vehículo con anterioridad.
Postcondición	El usuario puede observar las pruebas realizadas al vehículo y saber si la MIL está encendida o apagada.

Tabla 4.25 CU-21: Mostrar PIDs disponibles y valores almacenados.

CU-21	Mostrar PIDs disponibles y valores almacenados
Dependencias	(ACT-01) Usuario
Descripción	Muestra al usuario información sobre los PIDs implementados en el vehículo para conocer de antemano cuáles utilizar. Además, el usuario puede ver los valores obtenido por un comando concreto enviado.
Precondición	Se deben de haber consultados los PIDs implementados con anterioridad. En el caso de un PID en concreto, se debe de haber ejecutado su comando.
Postcondición	El usuario puede observar los PIDs disponibles y el valor de uno en concreto.

Tabla 4.26 CU-22: Enviar mensaje al servidor.

CU-22	Enviar mensaje al servidor
Dependencias	(ACT-01) Usuario (ACT-03) Servidor Web (BBDD)
Descripción	Se encarga del envío de un mensaje en un formato genérico establecido con los valores de la MAC desde dónde se envía la información al servidor, la geolocalización si la hubiera y el DTC obtenido.
Precondición	El sistema debe conocer los DTC en caso de que haya, la MAC del equipo donde se ejecuta el software y la localización en el caso de que estuviera disponible antes de enviar la información al servidor.
Postcondición	El servidor web dispone de la información comentada tras el envío de ésta desde el vehículo.

Tabla 4.27 CU-23: Enviar mensaje en formato JSON.

CU-23	Enviar mensaje en formato JSON
Dependencias	(ACT-01) Usuario (ACT-03) Servidor Web (BBDD) (CU-22) Enviar mensaje al servidor
Descripción	Es una particularización del caso de uso enviar mensaje al servidor (CU-22) en la que se envía la información en formato JSON añadiendo además un campo con la fecha en la que se ha enviado el mensaje.
Precondición	El sistema debe conocer los DTC en caso de que haya, la MAC del equipo donde se ejecuta el software y la localización en el caso de que estuviera disponible antes de enviar la información al servidor.
Postcondición	El servidor web dispone de la información comentada tras el envío de ésta desde el vehículo.

4.4 Catálogo de requisitos del sistema

4.4.1 Requisitos funcionales de información

A continuación se describe qué información debe almacenar el sistema para poder ofrecer los servicios necesarios.

Tabla 4.28 RFI-01: Información del dispositivo bluetooth ELM327.

RFI-01	Información del dispositivo bluetooth ELM327
Descripción	El sistema debe almacenar información sobre el dispositivo bluetooth ELM327 al que conectarse.
Datos específicos	• Nombre del dispositivo (Cadena de caracteres). • PIN (Cadena de caracteres).
Importancia	Alta

Tabla 4.29 RFI-02: Información de comandos AT y OBD.

RFI-02	Información de comandos AT y OBD
Descripción	El sistema debe almacenar información de los distintos comandos OBD y AT que el usuario puede utilizar (PID).
Datos específicos	• Nombre del comando (Cadena de caracteres). • Descripción del comando (Cadena de caracteres). • Comando (Cadena de caracteres). • Bytes de respuesta (Entero). • Decodificador de la respuesta (Cadena de caracteres). • Valor máximo de la respuesta (Flotante). • Valor mínimo de la respuesta(Flotante). • Unidad de la respuesta (Cadena de caracteres). • Tipo de dato (Cadena de caracteres). • Valor de la respuesta (Cualquiera).
Importancia	Alta

Tabla 4.30 RFI-03: Información del servidor web.

RFI-03	Información del servidor web
Descripción	El sistema debe almacenar información de la conexión con el servidor web al que enviar los datos, las interfaces de red (MAC) y GPS del dispositivo donde se ejecuta el software, y el periodo de muestreo de la obtención de DTC en el vehículo.
Datos específicos	• Dirección IP (Cadena de caracteres). • Puerto de conexión (Cadena de caracteres). • Nombre de la interfaz de red con conexión a Internet (Cadena de caracteres). • Puerto de escucha del servicio gpsd GPS (Cadena de caracteres). • Periodo de muestreo en segundos (Entero).
Importancia	Alta

Tabla 4.31 RFI-04: Información del mensaje de envío al servidor.

RFI-04	Información del mensaje de envío al servidor
Dependencias	(RFI-02) Información de comandos AT y OBD (RFI-03) Información del servidor web
Descripción	El sistema debe establecer un formato para el mensaje de envío al servidor web, almacenando los datos de respuesta necesarios con anterioridad.
Datos específicos	• Fecha y hora (Cadena de caracteres). • (RFI-03) Coordenadas (Vector de Enteros). • (RFI-03) Identificación del vehículo (MAC) (Cadena de caracteres). • (RFI-02) Valor del dato a enviar al servidor (Cadena de caracteres).
Importancia	Media

Tabla 4.32 RFI-05: Información relativa al vehículo.

RFI-05	Información relativa al vehículo
Dependencias	(RFI-01) Información del dispositivo bluetooth ELM327 (RFI-02) Información de comandos AT y OBD
Descripción	El sistema debe almacenar información relativa al estado del vehículo como puede ser el estado de la conexión OBD, lista de PIDs disponibles, resultados de las pruebas realizadas, número de DTC activos, valor de la última respuesta a un comando o el número de identificación del vehículo.
Datos específicos	• Lista de PIDs implementados en el vehículo (Vector de cadena de caracteres). • Lista de DTC activos (Vector de cadena de caracteres). • (RFI-02) Lista de comandos que pueden enviarse al vehículo (Contenedor asociativo de cadena de caracteres para comandos). • Número de Identificación del Vehículo (VIN) (Cadena de caracteres). • Protocolo OBD utilizado (Cadena de caracteres). • Número de protocolo OBD utilizado (Cadena de caracteres). • Resultados de las pruebas (Contenedor asociativo de cadena de caracteres). • Decodificadores de respuestas (Contenedor asociativo de cadena de caracteres para funciones). • (RF-01) Dirección física conexión bluetooth (Cadena de caracteres). • Estado de la conexión bluetooth (Booleano).
Importancia	Alta

4.4.2 Requisitos funcionales de reglas de negocio

En esta sección se describen restricciones, reglas o políticas del negocio que deben ser respetadas por el sistema.

Tabla 4.33 RFN-01: Intervalo de muestreo.

RFN-01	Intervalo de muestreo
Dependencias	(RFI-03) Información del servidor web
Descripción	Los usuarios no podrán modificar el intervalo en el que se realizan consultas de información al vehículo en tiempo de ejecución. Éste valor está preestablecido mediante el fichero de configuración y se carga al inicio de la ejecución.
Importancia	Media
Comentarios	Deberá tener un valor que no sobrecargue el dispositivo, no recomendable menos de 30 segundos, y dependerá de los datos que se necesiten obtener.

Tabla 4.34 RFN-02: Conexiones simultáneas al ELM327.

RFN-02	Conexiones simultáneas al ELM327
Descripción	Sólo puede existir una única conexión con el dispositivo bluetooth de ELM327 para el envío de mensajes AT y OBD.
Importancia	Alta
Comentarios	Pueden existir problemas en la comunicación en caso de más de una conexión.

Tabla 4.35 RFN-03: Reconexión ante caída del servicio.

RFN-03	Reconexión ante caída del servicio
Descripción	Si el servicio pierde la conexión bluetooth o deja de funcionar por cualquier circunstancia, el servicio debe de reintentar desde el inicio la conexión para recuperar el normal funcionamiento.
Importancia	Alta
Comentarios	Puede darse el caso de pérdida de conexión con el dispositivo bluetooth por cualquier circunstancia y el sistema debe ser capaz de recuperarse automáticamente.

Tabla 4.36 RFN-04: Incrementar tiempo entre envíos tras detectar DTC.

RFN-04	Incrementar tiempo de envío tras detectar DTC
Dependencias	(RFI-03) Información del servidor web
Descripción	Tras detectar un DTC puede que éste se recupere con el transcurso de los ciclos o sea necesaria una reparación. En este último caso, para no sobrecargar de envío de mensajes al servidor se debe de aumentar el tiempo entre envío de mensajes proporcionalmente al tiempo transcurrido tras la primera detección del DTC.
Importancia	Media
Comentarios	Este parámetro debe ser configurable a través de archivo de configuración.

Tabla 4.37 RFN-05: No debe interferir en el funcionamiento del vehículo.

RFN-05	No debe interferir en el funcionamiento del vehículo
Descripción	El sistema debe ser únicamente un sistema de consulta y nunca debe interferir en el correcto funcionamiento del vehículo.
Importancia	Alta

4.4.3 Requisitos funcionales de conducta

Estos requisitos describen los servicios que debe ofrecer el sistema para que los usuarios u otros sistemas puedan realizar sus tareas de negocio que no se hayan especificado mediante los casos de uso.

Tabla 4.38 RFC-01: Mostrar evolución del intercambio de mensajes OBD.

RFC-01	Mostrar evolución del intercambio de mensajes OBD
Descripción	El sistema puede dejar registrado el intercambio de mensajes con el vehículo para poder obtener distintos tipos de informe o estadísticas si el usuario lo habilita.
Importancia	Baja
Comentarios	Hay que tener en cuenta que al habilitar el registro de todos los mensajes puede repercutir en un aumento del tamaño de almacenamiento en los logs del sistema, debiendo configurar correctamente algún tipo de herramienta de rotado de logs.

4.4.4 Requisitos no funcionales de fiabilidad

En este apartado se define aspectos relacionados con la capacidad del software desarrollado para mantener su nivel de prestación bajo condiciones establecidas y durante un período de tiempo establecido.

Tabla 4.39 RNFF-01: Tiempo de recuperación del sistema menor de 1 minuto.

RNFF-01	Tiempo de recuperación del sistema menor de 1 minuto
Dependencias	(RFN-03) Reconexión ante caída del servicio
Descripción	El sistema deberá de recuperarse en caso de fallo en menos de 1 minuto mediante un reinicio automático del servicio si se encuentra inoperativo.
Importancia	Alta

Tabla 4.40 RNFF-02: Reinicio del sistema no implica pérdida de datos.

RNFF-02	Reinicio del sistema no implica pérdida de datos
Descripción	Dado que el sistema envía los datos que deben almacenarse persistentemente en el servidor web, éste puede reiniciarse recuperando los valores que tiene el vehículo actualmente, mientras que los valores más antiguos estarán almacenados en la BBDD del servidor web.
Importancia	Alta

Tabla 4.41 RNFF-03: Fallos en la respuesta del vehículo no implican corte en el servicio.

RNFF-03	Fallos en la respuesta del vehículo no implican corte en el servicio
Descripción	Una incorrecta respuesta por parte del dispositivo ELM327 no impidirá que el sistema pueda seguir enviando y recibiendo información del resto de mensajes.
Importancia	Alta
Comentarios	Únicamente en este caso, no estará disponible el dato solicitado cuya respuesta por parte del ELM327 no esté disponible en el vehículo o no sea válida.

4.4.5 Requisitos no funcionales de usabilidad

Los requisitos no funcionales de usabilidad definen aspectos relacionados con las dificultades que pueden encontrar los usuarios al enfrentarse al uso del sistema.

Tabla 4.42 RNFU-01: Utilización con un único fichero de cabecera.

RNFU-01	Utilización con un único fichero de cabecera
Descripción	Los usuarios pueden utilizar el sistema de forma fácil y sencilla, sin necesidad de dependencias, añadiendo únicamente un fichero de cabecera a su proyecto.
Importancia	Media

Tabla 4.43 RNFU-02: Acceso rápido a datos del vehículo sin necesidad de comprensión del funcionamiento.

RNFU-02	Acceso rápido a datos del vehículo sin necesidad de comprensión del funcionamiento
Descripción	Los usuarios mediante la instanciación de un objeto de conexión al vehículo pueden obtener datos directamente sin necesidad de enviar ningún mensaje y sin conocer el funcionamiento en profundidad de cómo se establece la conexión OBD ni el formato de sus mensajes.
Importancia	Media

Tabla 4.44 RNFU-03: Utilización de comandos mediante su nombre descriptivo.

RNFU-03	Utilización de comandos mediante su nombre descriptivo
Descripción	Para solicitar algún dato del vehículo el sistema debe permitir la utilización de nombres que representen la información en lugar del comando en bytes para facilitar al usuario la comprensión y aumentar la atractividad de utilizar la herramienta.
Importancia	Media

4.4.6 Requisitos no funcionales de eficiencia

En este apartado se pueden observar aspectos que indican la proporción entre el nivel de cumplimiento del software y la cantidad de recursos necesitados bajo condiciones establecidas.

Tabla 4.45 RNFE-01: Tiempo de respuesta a una solicitud debe ser inferior a 1 segundo.

RNFE-01	Tiempo de respuesta a una solicitud debe ser menor de 1 segundo
Descripción	Cuando el usuario realice alguna solicitud para obtener algún dato del vehículo, el tiempo en que se obtienen los datos de respuesta y se procesan debe ser inferior a 1 segundo.
Importancia	Media
Comentarios	No incluye el tiempo que transcurre desde que la ECU envía el primer bit de información hasta el último, sino el tiempo una vez recibida toda la información de la ECU.

Tabla 4.46 RNFE-02: Disponibilidad de mantenimiento del servicio.

RNFE-02	Disponibilidad de mantenimiento del servicio
Descripción	El servicio debe estar operativo todos los días de la semana y todas las horas del día, en función de la utilización del vehículo. Para el mantenimiento del servicio, se tendrá disponible las horas en las que un vehículo no se utilice.
Importancia	Media

Tabla 4.47 RNFE-03: Independencia con el servidor web.

RNFE-03	Independencia con el servidor web
Descripción	El sistema es independiente del estado del servicio en el servidor web, con lo que continúa enviando la información a éste aunque esté inoperativo.
Importancia	Baja

4.4.7 Requisitos no funcionales de mantenibilidad

Los requisitos no funcionales de mantenibilidad definen aspectos relacionados con la facilidad de ampliar la funcionalidad, modificar o corregir errores en un sistema.

Tabla 4.48 RNFM-01: Amplia modularidad del sistema.

RNFM-01	Amplia modularidad del sistema
Descripción	El sistema debe ser desarrollado con una modularidad que permita la inclusión de nuevos PID y nuevos modos sin que implique cambios radicales en la aplicación del usuario.
Importancia	Alta

Tabla 4.49 RNFM-02: Cambios en el sistema a través de ficheros de configuración.

RNFM-02	Cambios en el sistema a través de ficheros de configuración
Descripción	El cambio de configuración de aspectos relevantes solo implica cambios en ficheros de configuración y no necesita una recompilación del código original.
Importancia	Media
Comentarios	Esto facilita la utilización de pruebas en distintos entornos, con sólo aplicar un reinicio del servicio.

4.4.8 Requisitos no funcionales de portabilidad

En esta sección, se muestran aspectos relacionados con la capacidad de un sistema software para ser transferido desde una plataforma a otra.

Tabla 4.50 RNFP-01: Autoconfiguración del servicio a través de script.

RNFP-01	Autoconfiguración del servicio a través de script
Descripción	El usuario debe de disponer de un script de autonconfiguración del sistema para que esté disponible como servicio en el sistema operativo.
Importancia	Media

Tabla 4.51 RNFP-02: Distribuciones derivadas de Debian.

RNFP-02	Distribuciones derivadas de Debian
Descripción	El sistema desarrollado debe ser compatible con cualquier distribución derivada de Debian con compilador G++ (C++17).
Importancia	Alta

4.4.9 Requisitos no funcionales de seguridad

A continuación se detallan aspectos relativos a la políticas de privacidad del sistema.

Tabla 4.52 RNFS-01: Acceso físico al dispositivo donde se ejecute el software debe estar restringido.

RNFS-01	Acceso físico al dispositivo dónde se ejecute el software debe estar restringido
Descripción	El dispositivo donde se ejecute el software debe estar restringido y oculto, para evitar que alguien pueda acceder directamente desde el vehículo.
Importancia	Alta

Tabla 4.53 RNFS-02: Envío de mensajes al servidor web.

RNFS-02	Envío de mensajes al servidor web sin encriptar
Descripción	Debido al tipo de arquitectura, se envía un mensaje al servidor web con el protocolo UDP sin encriptar. Por tanto, hay que aplicar algún tipo de filtrado por MAC del dispositivo que envía el mensaje en el servidor, para sólo aceptar mensajes permitidos.
Importancia	Alta

4.4.10 Restricciones técnicas

En este apartado se detallan las limitaciones de diversa índole que se imponen al proyecto y que impactan tanto en el desarrollo como en el producto final.

Tabla 4.54 RT-01: Debe ser un sistema distribuido.

RT-01	Debe ser un sistema distribuido
Descripción	El sistema debe utilizar la arquitectura cliente-servidor para el envío de mensajes. El cliente se ejecutará en el vehículo y el servidor recibirá los mensajes de los distintos vehículos.
Importancia	Media

Tabla 4.55 RT-02: Características del cliente.

RT-02	Características del cliente
Descripción	Sistema linux con distribución derivada de Debian. Uso de compilador G++ con C++17 y librerías internas bluetooth y pthread. Disponer de interfaz bluetooth para la comunicación con ELM327 y conexión a Internet para el envío de datos al servidor.
Importancia	Media
Comentarios	Si se necesita conocer la ubicación, además se necesita un GPS conectado al dispositivo cliente.

Tabla 4.56 RT-03: Características del servidor.

RT-03	Características del servidor
Descripción	Servidor remoto con Node.js y MongoDB.
Importancia	Baja

Tabla 4.57 RT-04: Lenguaje de programación C++.

RT-04	Lenguaje de programación C++
Descripción	La librería debe ser programada en C++.
Importancia	Alta

4.4.11 Requisitos de integración

A continuación se identifican los componentes software, como librerías externas, cuya funcionalidad es relevante para el sistema a desarrollar en este proyecto.

Tabla 4.58 RI-01: Integración con librerías externas.

RI-01	Integración con librerías externas
Descripción	Se utilizan librerías externas para facilitar la utilización de objetos JSON en C++ y la utilización de un framework para realizar pruebas de software.
Importancia	Alta

5 Diseño del sistema

Con la ayuda de la especificación de requisitos del sistema se procede con el diseño de la librería. Para ello, en este capítulo se describe con ayuda de diagramas UML los diferentes subsistemas comentados en el capítulo anterior.

5.1 Diagrama de paquetes

Este diagrama representa de forma estática los paquetes del sistema que está siendo modelado, entendidos como un conjunto de elementos lógicos. Estos paquetes deben tener alguna función diferenciada del resto de elementos. Para este proyecto, se ha recogido en el diagrama los cuatro subsistemas detallados en la ERS.

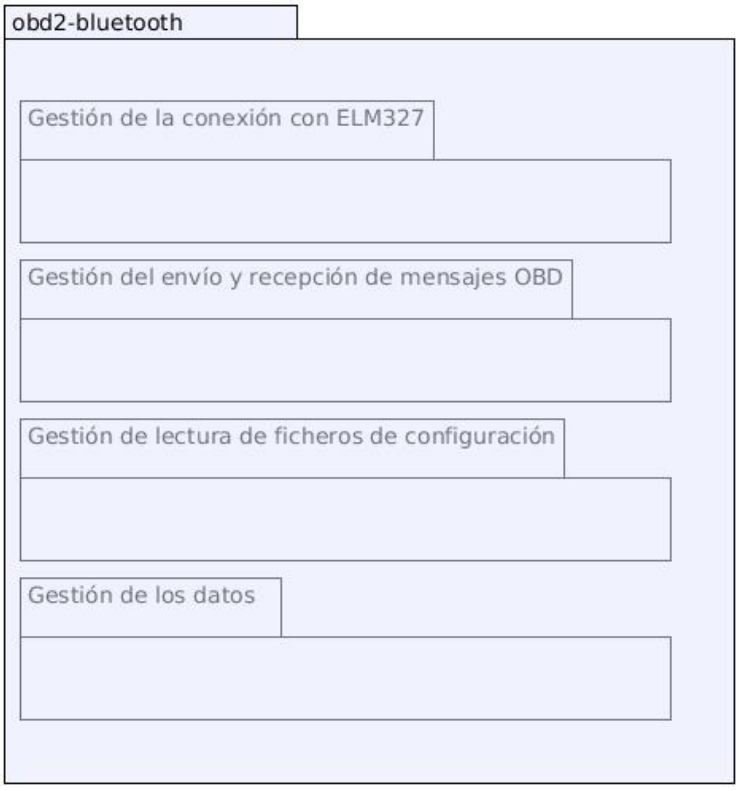


Figura 5.1 Diagrama de paquetes del sistema.

5.2 Diagrama de componentes

El diagrama de componente mostrado en la Figura 5.2 se utiliza para modelar los componentes que implementan las funcionalidades del sistema con una vista de alto nivel, representando la forma en la que estos se organizan y sus dependencias.

Se puede observar la existencia de un componente central main que relaciona y utiliza el resto de componentes del sistema. Este componente gestor del sistema sirve de puente para poder utilizar los módulos de gestión descritos con anterioridad y para acceder a las interfaces expuestas por el dispositivo ELM327 y el servidor web.

Cabe destacar la importancia de los artefactos representados como archivos de configuración de la conexión (configuration.cfg) y de los PIDs (PIDS.json) que se encuentran el entorno de ejecución del sistema operativo dónde se ejecutará el software.

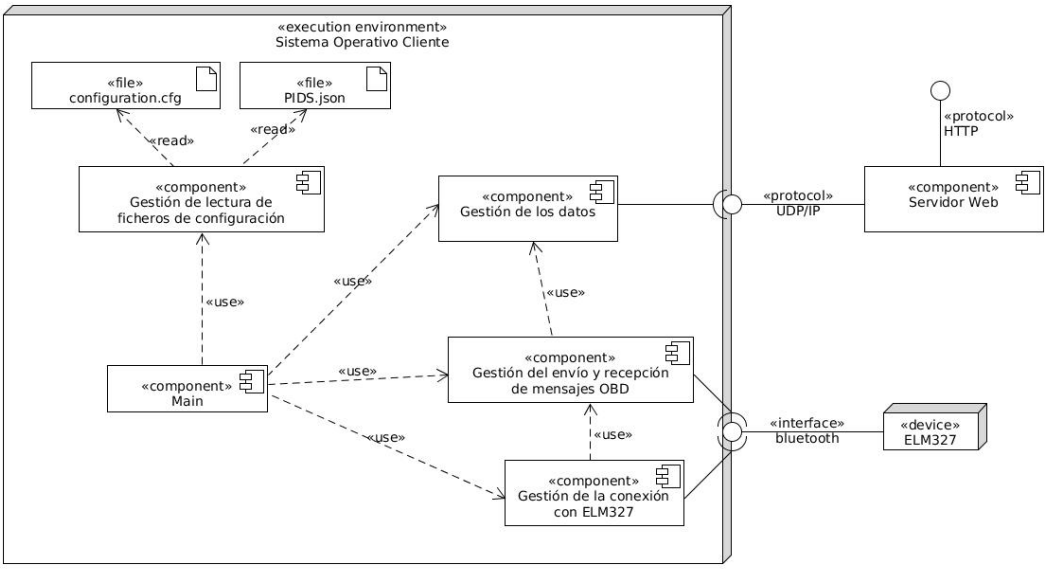


Figura 5.2 Diagrama de componentes del sistema.

5.3 Diagrama de clases

El diagrama de clase representado en la Figura 5.3 define las clases que se utilizarán cuando se proceda con la fase de desarrollo, la manera en que se relacionan las mismas y muestra el modelo lógico de los datos del sistema.

En este caso, podemos observar que existen cuatro clases con el estereotipo de entidad (Obd, Commands, AlarmFile y GpsClient), y una con el estereotipo de control (Main). Además, se representan sus atributos y métodos, junto con la relación existente entre ellas. Sólo es necesario el uso de estas clases debido a que el desarrollo en C++ permite la importación de funciones sin necesidad de la creación de clases para ello.

Las clases representadas son las siguientes:

- **Obd:** Clase que representa la conexión bluetooth y OBD con el vehículo y almacena información de sus respuestas. Es la clase principal para el funcionamiento del proyecto.
- **Commands:** Clase que representa el conjunto de comandos AT y OBD genéricos implementados en los vehículos y sus características. Necesaria para el intercambio de mensajes.
- **AlarmFile:** Clase que representa la conexión con el servidor web para el envío de información de forma remota. Almacena información relativa al servidor como IP y puerto de conexión.
- **GpsClient:** Clase que representa la conexión con el servicio gpsd para la obtención de coordenadas GPS. Almacena información relativa al servicio gpsd como el puerto UDP de conexión.
- **Main:** Clase controladora que instancia el resto de clases para permitir el funcionamiento global del sistema.

La relación entre las clases es la siguiente:

- Relación de composición entre Obd y Commands, en la que no tienen sentido la existencia de los comandos que hay que utilizar en la comunicación OBD, si no hay conexión OBD creada. Una instancia de OBD puede contener múltiples comandos, sin embargo, un comando solo puede pertenecer a una conexión OBD.
- Relación de dependencia de Main de instanciación de las clases Obd, AlarmFile y GpsClient. Únicamente se necesita instanciar un objeto de cada clase.

5.4 Diagrama de actividad

En esta sección, se puede observar en la Figura 5.4, el diagrama de actividad del sistema que muestra una secuencia de acciones, es decir, el flujo de trabajo del sistema que va desde un punto inicial hasta un punto final.

De este diagrama cabe destacar, que las actividades que guían el flujo de trabajo que se ha descrito, vienen definidas con el único objetivo de monitorizar los DTC del vehículo y enviarlos al servidor web para su notificación. Por este motivo, la finalización en la ejecución sólo se produce en el caso de que no exista una correcta conexión con el vehículo. En caso contrario, existe un envío constante de mensajes al vehículo cada cierto tiempo configurado.

5.5 Diagrama de secuencia

El diagrama de secuencia representa el intercambio de mensajes entre los distintos objetos del sistema para cumplir con una funcionalidad. En este caso, en la Figura 5.5 en el diagrama de actividad, se representa el intercambio de mensajes necesario para la monitorización de DTC y la visualización de ellos a través de un servidor web que recibe estos mensajes.

Figura 5.3 Diagrama de clases del sistema.

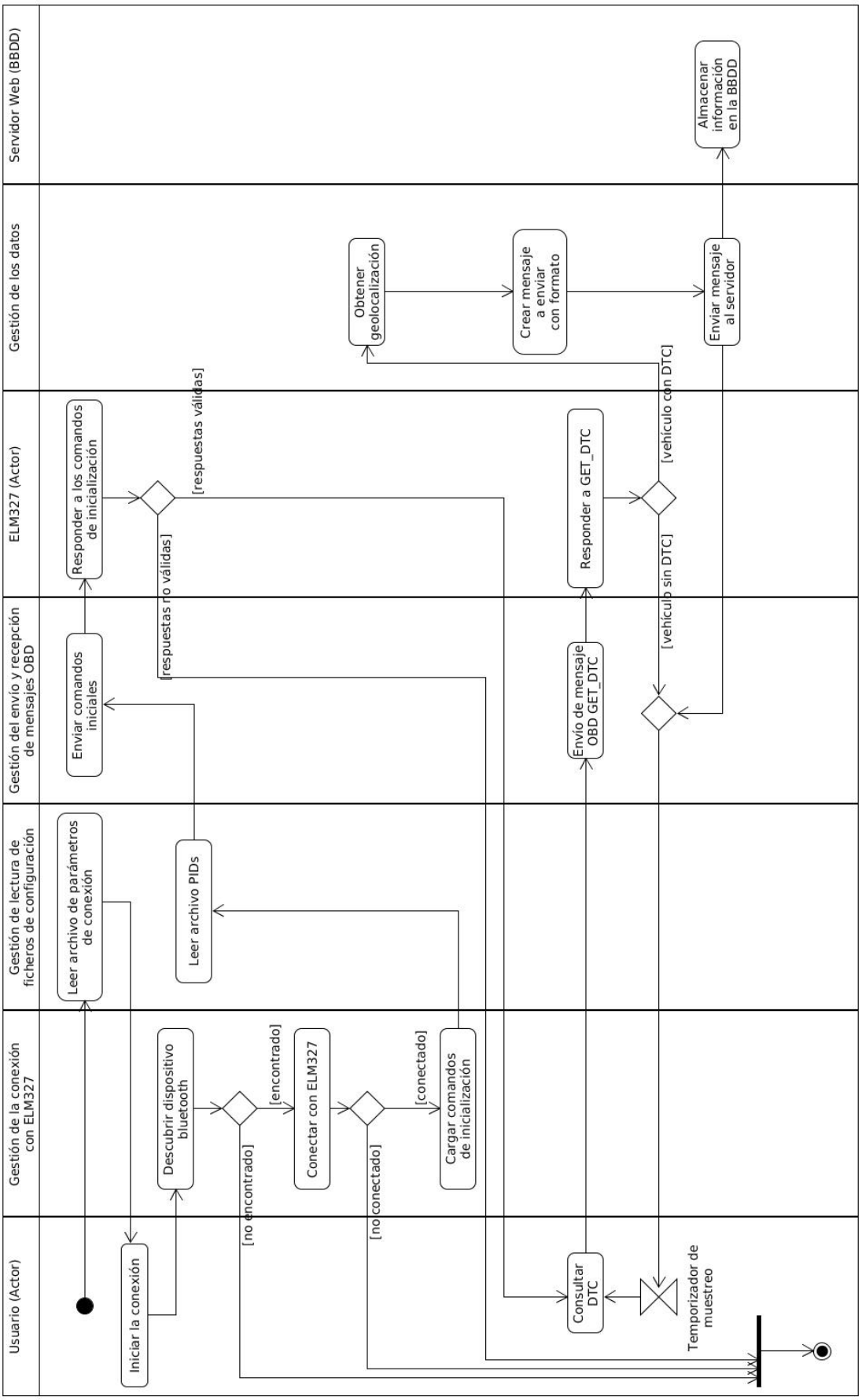


Figura 5.4 Diagrama de actividad del sistema.

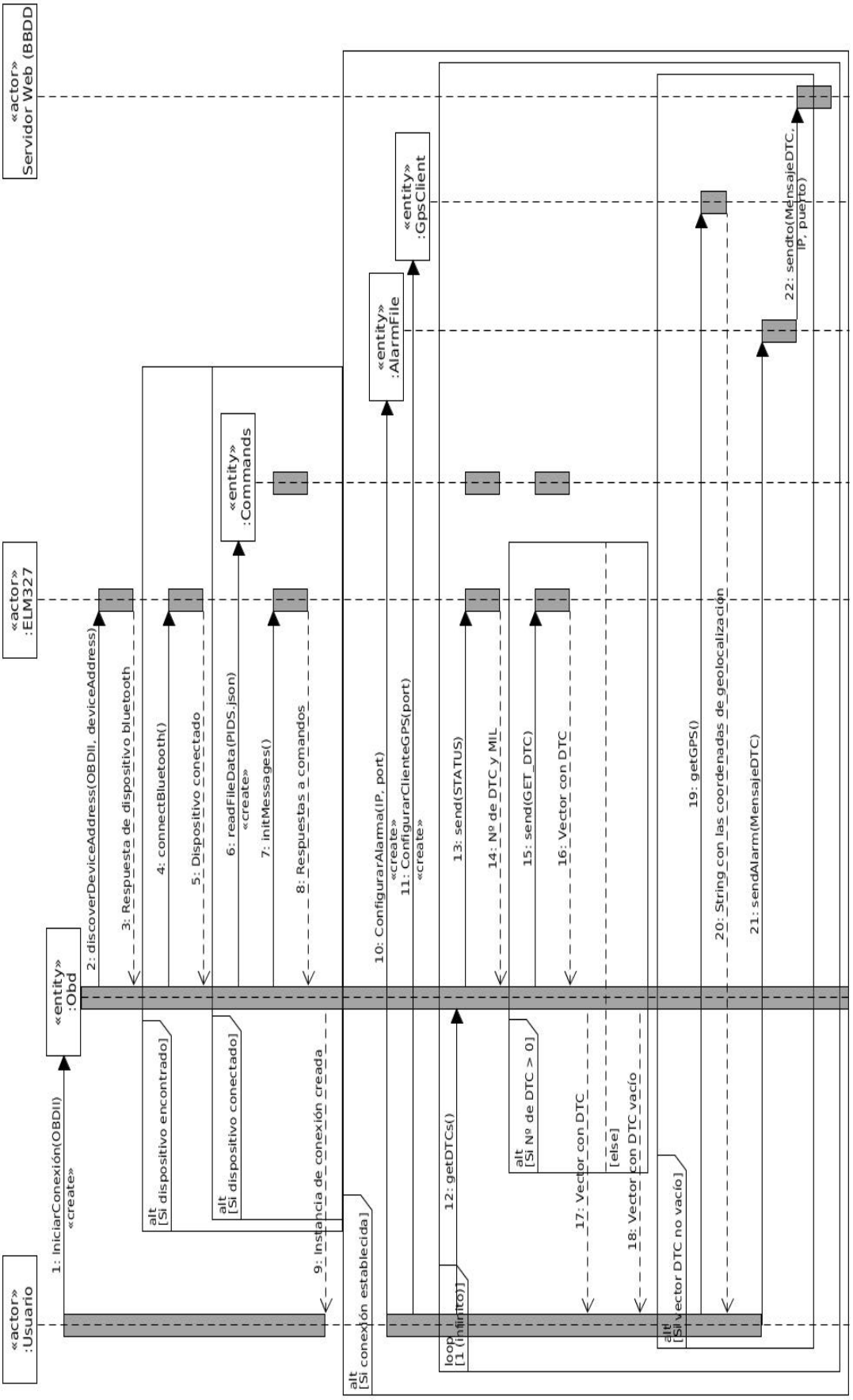


Figura 5.5 Diagrama de secuencia del sistema.

6 Implementación

Prosiguiendo el ciclo de vida del desarrollo software, en este capítulo se detallará la implementación del sistema, programando la librería de conexión OBD, junto con una aplicación y un servidor web que permita la monitorización de fallos en los vehículos, obteniendo en el servidor web el DTC en el momento de producirse.

6.1 Arquitectura del proyecto

El núcleo principal de este proyecto es el desarrollo de la librería obd2-bluetooth para facilitar el desarrollo de aplicaciones que permitan la comunicación con los vehículos y poder utilizar datos relevantes de ellos. Sin embargo, para comprender mejor qué utilidad puede ofrecer esta librería, se ha desarrollado una aplicación completa que obtiene los DTC del vehículo y los envía y almacena en un servidor web. Esto facilita la monitorización de los vehículos a los responsables de éstos a través de una interfaz web sencilla.

Por consiguiente, se ha dividido esta sección en las dos partes del sistema completo a desarrollar: aplicación cliente y librería obd2-bluetooth, y servidor web con BBDD MongoDB.

6.1.1 Aplicación cliente y librería obd2-bluetooth

Con el objetivo de entender de qué ficheros está compuesta la herramienta desarrollada y la finalidad de cada uno de ellos, se muestra en la Figura 6.1 la distribución del código de este subsistema del proyecto. A continuación se explica brevemente el origen de cada fichero, y la función que desempeña en el sistema global:

- **Carpeta conf:** Es la carpeta de configuración del sistema en la que existen los siguientes ficheros:
  - **configuration.cfg:** Archivo de configuración en la que aparecen datos de la conexión bluetooth, la conexión con el servidor remoto (IP y puerto) y el periodo de consulta de DTC en el vehículo.
  - **installService.sh y monDTC.service:** Script de instalación del software como servicio en el sistema operativo y fichero con la información del servicio.
  - **pairBluetooth.sh:** Script de vinculación bluetooth de forma manual. No es necesario para el funcionamiento del sistema, pero se proporciona para casos en los que se desee vincular el dispotivo bluetooth de forma manual antes de ejecutar la aplicación.
- **Carpeta data:** Carpeta de datos del sistema. Contiene el fichero PIDS.json en el que aparecen los comandos AT y PIDS OBD genéricos. El sistema los necesita para conocer el tipo de mensaje a enviar en cada caso. Además, se proporciona el fichero DTC-description que muestra la descripción de cada DTC.
- **Carpeta doc:** Contiene la documentación del proyecto generada con doxygen.
- **Carpeta lib:** Carpeta en la que se genera la librería con extensión .a
- **Carpeta src:** Carpeta con el código fuente de la aplicación. A su vez ésta, contiene la carpeta external, con los ficheros de cabecera json.hpp [1] y catch.hpp [2] que corresponden con las librerías externas comentadas anteriormente. La descripción de los ficheros es la siguiente:
  - **Obd.hpp:** Fichero que contiene la declaración de la clase Obd en este proyecto. Es la clase principal en torno a la que gira todo el proyecto, ya que, para la conexión con el vehículo sólo

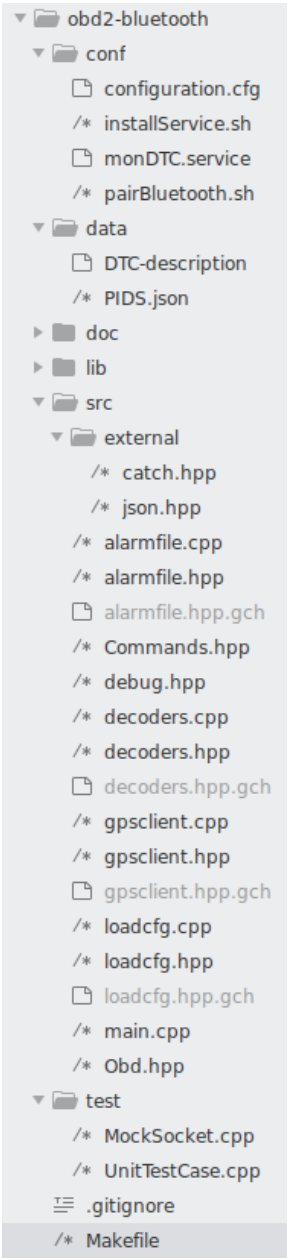


Figura 6.1 Distribución de código del proyecto obd2-bluetooth.

se necesita realizar una instanciación de esta clase. Es la encargada de la conexión bluetooth y del posterior envío de mensajes AT y OBD. Cualquier interacción que se necesite realizar con el vehículo, necesita de la instancia OBD para el envío y recepción de mensajes.

- **Commands.hpp:** Fichero que contiene la declaración de la clase Commands en este proyecto. La clase Obd se apoya en ésta para la lista de comandos que puede ejecutar y funciones relacionadas con ellos.
- **decoders.cpp/decoders.hpp:** Ficheros de declaración y definición de funciones de decodificación para los distintos tipos de mensajes OBD implementados en el vehículo.
- **main.cpp:** Fichero principal de la aplicación que se ha desarrollado con la finalidad de monitorizar los DTC de un vehículo y notificar al servidor remoto en caso de activación de alguno. Este fichero representa un ejemplo de lo que el usuario (desarrollador) podría realizar con la librería para una aplicación determinada.
- **debug.hpp:** Fichero que define funciones que facilitan la depuración de errores, sin la necesidad de que la aplicación final disponga de todos esos mensajes.

  - **alarmfile.cpp/alarmfile.hpp:** Ficheros que definen la clase AlarmFile para el envío de mensajes al servidor remoto.
  - **loadcfg.cpp/loadcfg.hpp:** Ficheros que definen las funciones encargadas de la lectura de un fichero de configuración con la estructura clave-valor.
  - **gpsclient.cpp/gpsclientgps.hpp:** Ficheros que definen las funciones que pueden obtener la geolocalización mediante el envío de un datagrama a un servicio externo que responde con las coordenadas.

- **Carpeta test:** Carpeta que contiene los archivos relacionados con el plan de pruebas:
  - **MockSocket.hpp:** Fichero desarrollado para simular la conexión bluetooth para las pruebas de integración.
  - **UnitTestCase.cpp:** Fichero principal de ejecución de pruebas unitarias y de integración. Utiliza el fichero MockSocket.hpp y el simulador obdsim para las pruebas desarrolladas.

6.1.2 Servidor Web con BBDD MongoDB

El motivo de elección de un servidor web para la recepción de datos es la sencillez de visualización de los datos obtenidos del vehículo. En este caso, además, se ha optado por utilizar Node.js y MongoDB para la interfaz web y el almacenamiento de datos respectivamente. En la Figura 6.2 se puede observar la distribución del código del servidor remoto:

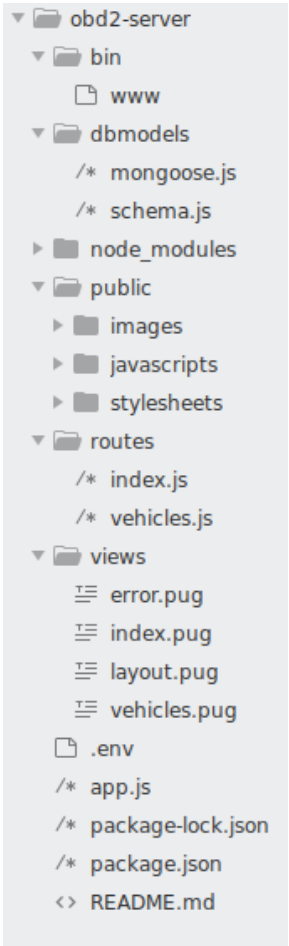


Figura 6.2 Distribución de código del proyecto obd2-server.

- **Carpeta bin:** Carpeta que contiene el fichero www ejecutable del servidor. Este archivo se encarga de realizar una llamada de ejecución a los ficheros app.js y mongoose.js, y de iniciar el servidor UDP de recepción de mensajes del vehículo y el servidor HTTP de visualización.

- **Carpeta dbmodels:** Contiene los ficheros mongoose.js (conexión con la BBDD MongoDB cuyo servicio debe estar activo) y schema.js (define el esquema de inicio del tipo de mensaje a recibir del vehículo).
- **Carpeta node_modules:** Carpeta con los módulos, almacenados por node, que necesita la aplicación para funcionar (express, mongoose, pug...).
- **Carpeta public:** Carpeta que contiene los ficheros públicos del servidor web como imágenes y hojas de estilo (bootstrap).
- **Carpeta routes:** Carpeta que contiene los ficheros de ruta para cada enlace en la interfaz web. En este caso, tenemos las rutas para la interfaz principal (index) y para la interfaz de DTC de vehículos (vehicles). Estos ficheros se encargan de cargar el fichero pug correspondiente.
- **Carpeta views:** Carpeta que contiene los ficheros pug con los que se puede escribir código HTML con una sintaxis mucho más sencilla. El fichero layout.pug nos sirve de base de reutilización para la página principal (index.pug), la de DTC de vehículos (vehicles.pug) y la página de error (error.pug).
- **.env:** Fichero del entorno de pruebas para configurar la dirección y puerto de la BBDD.
- **app.js:** Fichero que realiza la inicialización de los ficheros de rutas y de módulos encargados del procesamiendo de datos web.
- **package.json y package-lock.json:** Ficheros de node que se utilizan para el control de dependencias de módulos y versionado del proyecto.

Se puede observar en la Figura 6.3 la interfaz web pricipal y en la Figura 6.4 la tabla con los DTC capturados en una de las pruebas realizadas.

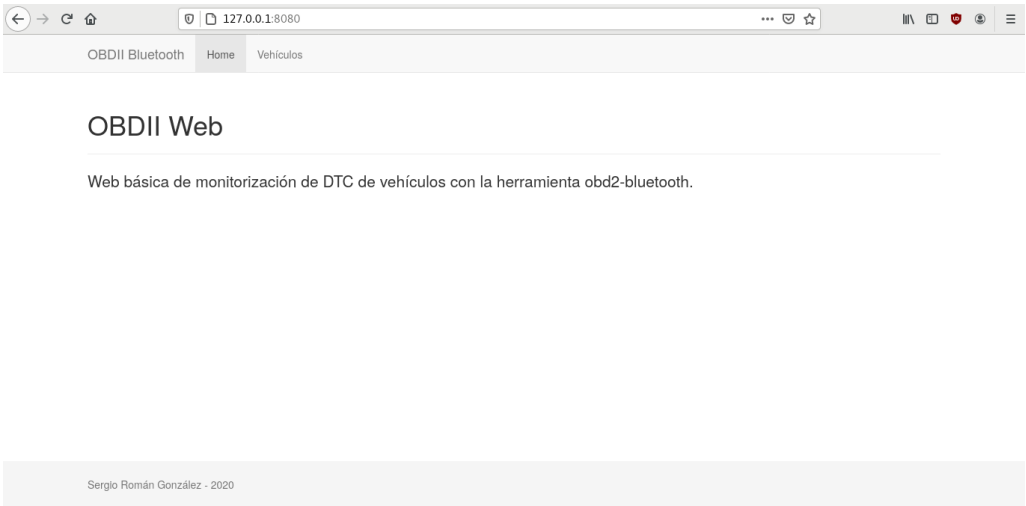


Figura 6.3 Interfaz web de inicio del servidor web remoto.

El formato de envío del mensaje del vehículo hacia el servidor web remoto es JSON facilitando así la introducción de la información en la BBDD MongoDB. Se puede observar el formato de un mensaje en formato JSON enviado y recepcionado en el servidor en el siguiente fragmento. Se muestran tres mensajes con distintos DTC.

Código 6.1 Formato del mensaje JSON enviado al servido remoto.

```
{"code":"P0454","coordinates":["171804","021116","3854.791","N","07702.465","W
    ","171804","171804","171804"],"date":"Tue Sep 8 20:07:39 2020\n","
    description":"https://codigosdtc.com/P0454","vehicle":"9c:ad:97:a3:a9:cf"}
{"code":"P0254","coordinates":["171804","021116","3854.791","N","07702.465","W
    ","171804","171804","171804"],"date":"Tue Sep 8 20:07:39 2020\n","
    description":"https://codigosdtc.com/P0254","vehicle":"9c:ad:97:a3:a9:cf"}
```

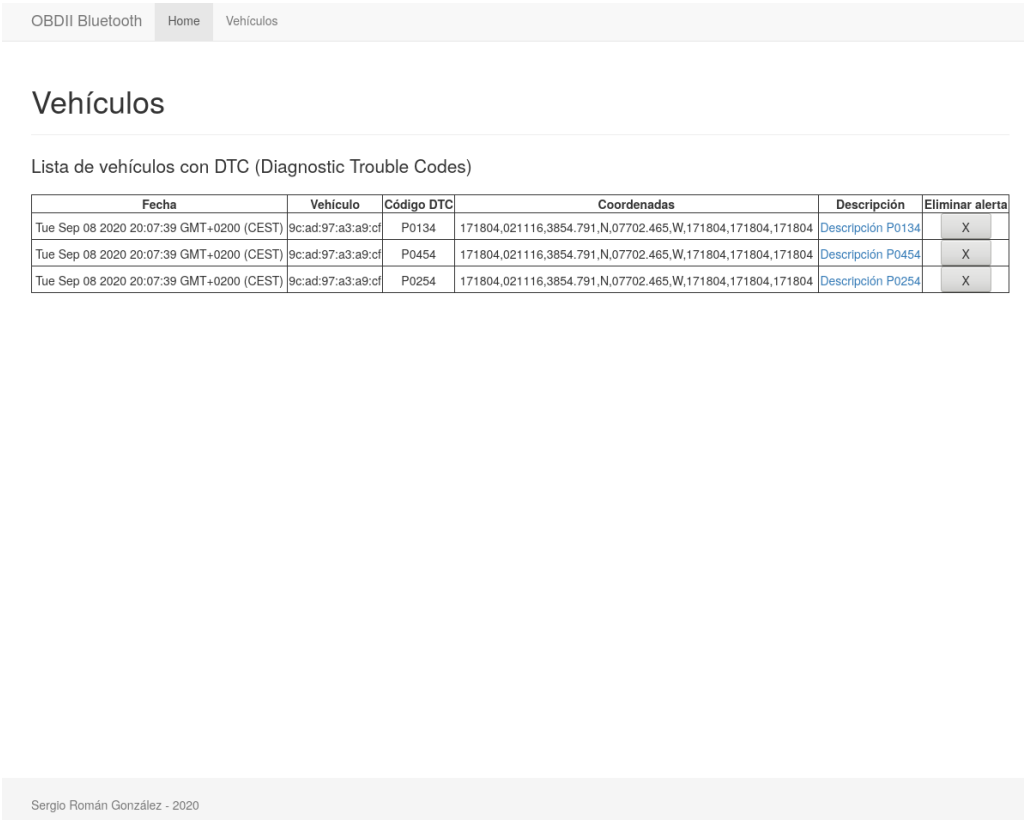


Figura 6.4 Interfaz web de DTC de vehículos del servidor web remoto.


```
{"code":"P0134","coordinates":["171804","021116","3854.791","N","07702.465","W
    ","171804","171804","171804"],"date":"Tue Sep 8 20:07:39 2020\n","
    description":"https://codigosdtc.com/P0134","vehicle":"9c:ad:97:a3:a9:cf"}
```


La descripción es un enlace a una web con la información detallada del DTC, de los síntomas, causas y posibles soluciones y el vehículo es la MAC de la interfaz de red del dispositivo asociado al vehículo.

También se puede observar en la Figura 6.5 mediante la herramienta Compass de conexión directa con la BBDD MongoDB los datos en formato JSON almacenados tras la recepción de los DTC del vehículo.

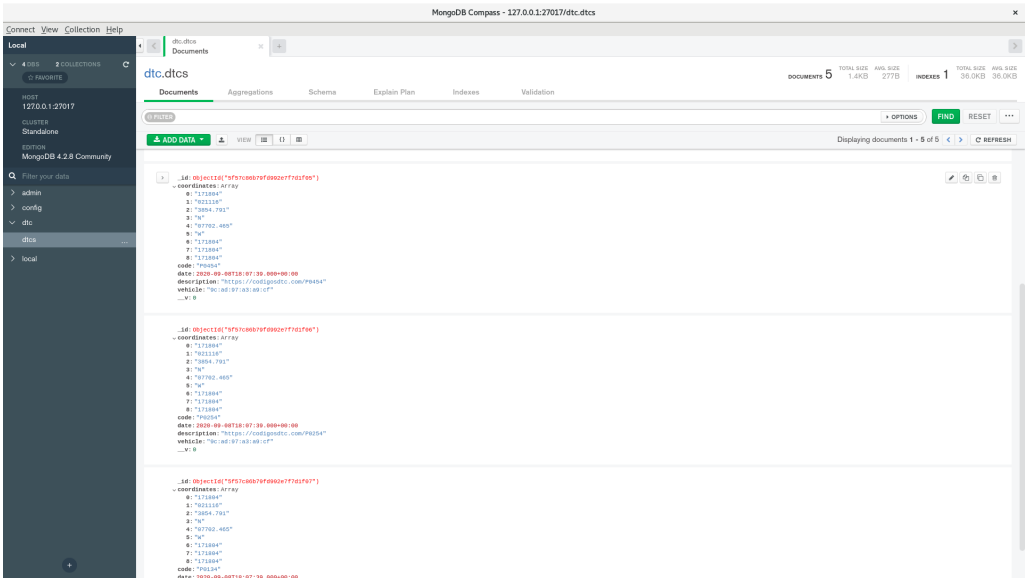


Figura 6.5 DTC recepcionados en la BBDD MongoDB en Compass.

7 Plan de pruebas

Con el objetivo de verificar y validar los requisitos del sistema, y comprobar el correcto funcionamiento de éste, es necesario la realización de una correcta estrategia de pruebas. Así pues, en este capítulo se describe el conjunto de pruebas necesarias a realizar, además de las herramientas utilizadas que han permitido la simulación de un dispositivo ELM327 de prueba.

En primer lugar, para la realización de pruebas de C++ se ha utilizado el framework catch2 [2] descrito anteriormente, ya que, ha facilitado la realización de éstas. Además, para la simulación del dispositivo ELM327 se han utilizado la herramientas obdsim [9] y minicom. Al mismo tiempo, se han desarrollado objetos simulados para poder realizar las pruebas sin la necesidad de conexión bluetooth. Por último, se han realizado pruebas en un vehículo para comprobar que el sistema funciona realmente. En la Figura 7.1 se muestra la interfaz gráfica disponible de esta aplicación.

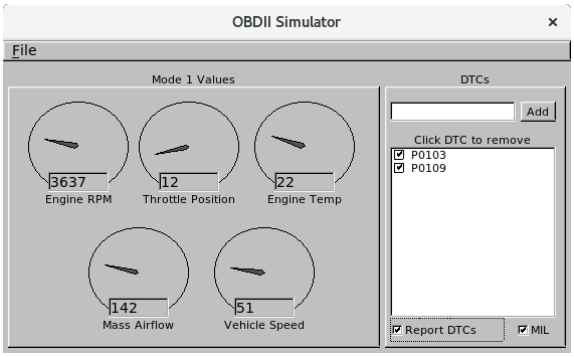


Figura 7.1 Interfaz gráfica del simulador OBDSIM.

7.1 Pruebas unitarias

En relación a las pruebas unitarias se muestran las siguientes tablas descriptivas.

Tabla 7.1 PU-01: Decodificadores.

PU-01	Decodificadores
Descripción	Pruebas que se encargan de comprobar que los valores devueltos por cada uno de los decodificadores son correctos. Se debe probar cada decodificador con su valor mínimo, máximo y medio y validar si el resultado es el esperado.
Casos de uso	(CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos
Prerequisitos	Disponer de los bytes útiles de la respuesta OBD del vehículo.
Resultado esperado	Se espera obtener el valor mínimo, máximo y medio de cada decodificador específico con sus distintos tipos de datos.

Tabla 7.2 PU-02: Conversor DTC.

PU-02	Conversor DTC
Descripción	El objetivo de esta prueba es comprobar que la conversión del primer byte de respuesta del DTC se realiza correctamente siguiendo las normas establecidas en la Tabla 3.2.
Casos de uso	(CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos (CU-18) Mostrar DTC
Prerequisitos	Disponer de los bytes útiles de la respuesta OBD del vehículo y como mínimo disponer de un DTC activo.
Resultado esperado	Se espera obtener el valor correspondiente a la conversión de la Tabla 3.2.

Tabla 7.3 PU-03: Conversor VIN.

PU-03	Conversor VIN
Descripción	Esta prueba tiene como fin la validación de la conversión de los bytes de respuesta útiles de VIN a caracteres.
Casos de uso	(CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos (CU-19) Mostrar VIN
Prerequisitos	Disponer de los bytes útiles de la respuesta OBD del vehículo.
Resultado esperado	Se espera obtener el valor correspondiente a la conversión de byte a 17 caracteres.

7.2 Pruebas de integración

A continuación se describen las distintas pruebas de integración a realizar y su correspondencia con los casos de uso del sistema. Cabe destacar que las pruebas del subsistema de gestión de la conexión con ELM327 vienen implícitas en todas las siguientes pruebas, ya que se producen posteriormente a la conexión bluetooth y OBD.

Tabla 7.4 PI-01: Mostrar DTC.

PI-01	Mostrar DTC
Descripción	Prueba que realiza la conexión con el vehículo, consulta los DTC activos y los muestra por consola.
Casos de uso	(CU-01) Descubrir dispositivo bluetooth (CU-02) Conectar con dispositivo ELM327 (CU-03) Cargar comandos de inicialización (CU-04) Cargar los decodificadores PID y AT (CU-07) Leer archivo JSON (CU-08) Enviar comando (CU-09) Consultar DTC (CU-13) Recibir respuesta (CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos (CU-16) Almacenar respuesta (CU-18) Mostrar DTC
Prerequisitos	Disponer de un simulador para la respuesta de los mensajes OBD (obd-sim) o bien disponer de un vehículo físico para las pruebas.
Resultado esperado	En el caso de que exista DTC obtener su código.

Tabla 7.5 PI-02: Mostrar VIN.

PI-02	Mostrar VIN
Descripción	Prueba que realiza la conexión con el vehículo, consulta el VIN y lo muestra por consola.
Casos de uso	(CU-01) Descubrir dispositivo bluetooth (CU-02) Conectar con dispositivo ELM327 (CU-03) Cargar comandos de inicialización (CU-04) Cargar los decodificadores PID y AT (CU-07) Leer archivo JSON (CU-08) Enviar comando (CU-10) Consultar VIN (CU-13) Recibir respuesta (CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos (CU-16) Almacenar respuesta (CU-19) Mostrar VIN
Prerequisitos	Disponer de un simulador para la respuesta de los mensajes OBD (obd-sim) o bien disponer de un vehículo físico para las pruebas.
Resultado esperado	Obtener el Número de Identificación del Vehículo.

Tabla 7.6 PI-03: Mostrar resultados de los test.

PI-03	Mostrar resultados de los test
Descripción	Prueba que realiza la conexión con el vehículo, solicita pruebas de estado del vehículo y muestra sus resultados por consola.
Casos de uso	(CU-01) Descubrir dispositivo bluetooth (CU-02) Conectar con dispositivo ELM327 (CU-03) Cargar comandos de inicialización (CU-04) Cargar los decodificadores PID y AT (CU-07) Leer archivo JSON (CU-08) Enviar comando (CU-11) Solicitar pruebas de estado del vehículo (CU-13) Recibir respuesta (CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos (CU-16) Almacenar respuesta (CU-20) Mostrar resultado de los test
Prerequisitos	Disponer de un simulador para la respuesta de los mensajes OBD (obd-sim) o bien disponer de un vehículo físico para las pruebas.
Resultado esperado	Obtener los resultados de cada una de las pruebas del vehículo, si han sido superadas o no.

Tabla 7.7 PI-04: Mostrar PIDs disponibles.

PI-04	Mostrar PIDs disponibles
Descripción	Prueba que realiza la conexión con el vehículo, consulta los PIDs implementados en el vehículo y los muestra por consola.
Casos de uso	(CU-01) Descubrir dispositivo bluetooth (CU-02) Conectar con dispositivo ELM327 (CU-03) Cargar comandos de inicialización (CU-04) Cargar los decodificadores PID y AT (CU-07) Leer archivo JSON (CU-08) Enviar comando (CU-12) Consultar lista de PID implementados (CU-13) Recibir respuesta (CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos (CU-16) Almacenar respuesta (CU-21) Mostrar PIDs disponibles y valores almacenados
Prerequisitos	Disponer de un simulador para la respuesta de los mensajes OBD (obd-sim) o bien disponer de un vehículo físico para las pruebas.
Resultado esperado	Obtener una lista con los PIDs implementados en el vehículo y que pueden ser consultados al enviar el comando directamente.

Tabla 7.8 PI-05: Obtener valor específico del vehículo.

PI-04	Obtener valor específico del vehículo
Descripción	Prueba que realiza la conexión con el vehículo, consulta la velocidad actual del vehículo y la muestra por consola.
Casos de uso	(CU-01) Descubrir dispositivo bluetooth (CU-02) Conectar con dispositivo ELM327 (CU-03) Cargar comandos de inicialización (CU-04) Cargar los decodificadores PID y AT (CU-07) Leer archivo JSON (CU-08) Enviar comando (CU-13) Recibir respuesta (CU-14) Identificar los datos útiles de la respuesta (CU-15) Decodificar datos (CU-16) Almacenar respuesta
Prerequisitos	Disponer de un simulador para la respuesta de los mensajes OBD (obd-sim) o bien disponer de un vehículo físico para las pruebas.
Resultado esperado	El valor que corresponda con el PID solicitado como puede ser las RPM del motor, temperatura o la velocidad del vehículo.

7.3 Procedimiento de pruebas y resultados

Para llevar a cabo la batería de pruebas configuradas, es necesario el ejecutable "test" que se genera con el Makefile del proyecto, ejecutando "make test" en el dispositivo (Raspberry Pi) donde se vaya a ejecutar la herramienta. Para más información detallada de como se genera este ejecutable lea detenidamente el Anexo A.

Código 7.1 Ejemplo de ejecución de pruebas automatizadas y sus resultados.

```
obd2-bluetooth$ ./test/test
[LOG][test/UnitTestCase.cpp][____C_A_T_C_H____T_E_S_T____0][Line 105]
    Comenzando test OBD class DTC
[LOG][test/UnitTestCase.cpp][____C_A_T_C_H____T_E_S_T____0][Line 107] Iniciamos
     el simulador OBDSIM
[LOG][test/UnitTestCase.cpp][initOBDSIM][Line 51] Ejecutamos OBDSIM.
...
[LOG][test/../src/Obd.hpp][polling][Line 399] Enviar DTC: P0103
[LOG][test/../src/Obd.hpp][polling][Line 399] Enviar DTC: U0401
---------------------------------------------------------------
{
    "description": "Get DTCs",
    "name": "GET_DTC",
    "units": "No units",
    "value": [
        "P0103",
        "U0401"
    ]
}
...
[LOG][test/UnitTestCase.cpp][____C_A_T_C_H____T_E_S_T____2][Line 128]
    Comenzando test OBD class data SPEED
```

```
[LOG][test/UnitTestCase.cpp][____C_A_T_C_H____T_E_S_T____2][Line 130] Iniciamos
     el simulador OBDSIM
[LOG][test/UnitTestCase.cpp][initOBDSIM][Line 51] Ejecutamos OBDSIM.
...
[LOG][test/../src/Obd.hpp][polling][Line 428] Get Vehicle Identification Number
     = No disponible.
...
[LOG][test/../src/Obd.hpp][initMessages][Line 499] Nº de comandos disponibles =
     95
[LOG][test/UnitTestCase.cpp][____C_A_T_C_H____T_E_S_T____2][Line 138]
    Finalizado el proceso de inicio de conexión OBD
0143 - ABSOLUTE_LOAD - Absolute load value - Min=0 Max=25700
0149 - ACCELERATOR_POS_D - Accelerator pedal position D - Min=0 Max=100
014A - ACCELERATOR_POS_E - Accelerator pedal position E - Min=0 Max=100
014B - ACCELERATOR_POS_F - Accelerator pedal position F - Min=0 Max=100
0112 - AIR_STATUS - Secondary Air Status - Min=0 Max=0
0146 - AMBIANT_AIR_TEMP - Ambient air temperature - Min=-40 Max=215
011E - AUX_INPUT_STATUS - Auxiliary input status (power take off) - Min=0 Max=0
0133 - BAROMETRIC_PRESSURE - Barometric Pressure - Min=0 Max=255
013C - CATALYST_TEMP_B1S1 - Catalyst Temperature: Bank 1 - Sensor 1 - Min=-40
    Max=6513.5
013E - CATALYST_TEMP_B1S2 - Catalyst Temperature: Bank 1 - Sensor 2 - Min=-40
    Max=6513.5
013D - CATALYST_TEMP_B2S1 - Catalyst Temperature: Bank 2 - Sensor 1 - Min=-40
    Max=6513.5
013F - CATALYST_TEMP_B2S2 - Catalyst Temperature: Bank 2 - Sensor 2 - Min=-40
    Max=6513.5
012C - COMMANDED_EGR - Commanded EGR - Min=0 Max=100
0144 - COMMANDED_EQUIV_RATIO - Commanded equivalence ratio - Min=0 Max=2
0142 - CONTROL_MODULE_VOLTAGE - Control module voltage - Min=0 Max=65.535
0131 - DISTANCE_SINCE_DTC_CLEAR - Distance traveled since codes cleared - Min=0
     Max=65535
0121 - DISTANCE_W_MIL - Distance Traveled with MIL on - Min=0 Max=65535
012D - EGR_ERROR - EGR Error - Min=-100 Max=99.2
0105 - ENGINE_COOLANT - Engine Coolant Temperature - Min=-40 Max=215
0104 - ENGINE_LOAD - Calculated Engine Load - Min=0 Max=100
012E - EVAPORATIVE_PURGE - Commanded Evaporative Purge - Min=0 Max=100
0132 - EVAP_VAPOR_PRESSURE - Evaporative system vapor pressure - Min=-8192 Max
    =8191.75
0102 - FREEZE_DTC - DTC that triggered the freeze frame - Min=0 Max=0
012F - FUEL_LEVEL - Fuel Level Input - Min=0 Max=100
010A - FUEL_PRESSURE - Fuel Pressure - Min=0 Max=765
0123 - FUEL_RAIL_PRESSURE_DIRECT - Fuel Rail Pressure (direct inject) - Min=0
    Max=655350
0122 - FUEL_RAIL_PRESSURE_VAC - Fuel Rail Pressure (relative to vacuum) - Min=0
     Max=5177.27
0103 - FUEL_STATUS - Fuel System Status - Min=0 Max=0
010B - INTAKE_PRESSURE - Intake Manifold Pressure - Min=0 Max=255
010F - INTAKE_TEMP - Intake Air Temp - Min=-40 Max=215
0107 - LONG_FUEL_TRIM_1 - Long Term Fuel Trim - Bank 1 - Min=-100 Max=99.2
0109 - LONG_FUEL_TRIM_2 - Long Term Fuel Trim - Bank 2 - Min=-100 Max=99.2
0110 - MAF - Air Flow Rate (MAF) - Min=0 Max=655.35
014F - MAX_VALUES - Various Max values - Min=0 Max=255
0114 - O2_B1S1 - O2: Bank 1 - Sensor 1 Voltage - Min=0 Max=1.275
0115 - O2_B1S2 - O2: Bank 1 - Sensor 2 Voltage - Min=0 Max=1.275
0116 - O2_B1S3 - O2: Bank 1 - Sensor 3 Voltage - Min=0 Max=1.275
0117 - O2_B1S4 - O2: Bank 1 - Sensor 4 Voltage - Min=0 Max=1.275
```

```
0118 - O2_B2S1 - O2: Bank 2 - Sensor 1 Voltage - Min=0 Max=1.275
0119 - O2_B2S2 - O2: Bank 2 - Sensor 2 Voltage - Min=0 Max=1.275
011A - O2_B2S3 - O2: Bank 2 - Sensor 3 Voltage - Min=0 Max=1.275
011B - O2_B2S4 - O2: Bank 2 - Sensor 4 Voltage - Min=0 Max=1.275
0134 - O2_S1_WR_CURRENT - O2 Sensor 1 WR Lambda Current - Min=0 Max=2
0124 - O2_S1_WR_VOLTAGE - O2 Sensor 1 WR Lambda Voltage - Min=0 Max=2
0135 - O2_S2_WR_CURRENT - O2 Sensor 2 WR Lambda Current - Min=0 Max=2
0125 - O2_S2_WR_VOLTAGE - O2 Sensor 2 WR Lambda Voltage - Min=0 Max=2
0136 - O2_S3_WR_CURRENT - O2 Sensor 3 WR Lambda Current - Min=0 Max=2
0126 - O2_S3_WR_VOLTAGE - O2 Sensor 3 WR Lambda Voltage - Min=0 Max=2
0137 - O2_S4_WR_CURRENT - O2 Sensor 4 WR Lambda Current - Min=0 Max=2
0127 - O2_S4_WR_VOLTAGE - O2 Sensor 4 WR Lambda Voltage - Min=0 Max=2
0138 - O2_S5_WR_CURRENT - O2 Sensor 5 WR Lambda Current - Min=0 Max=2
0128 - O2_S5_WR_VOLTAGE - O2 Sensor 5 WR Lambda Voltage - Min=0 Max=2
0139 - O2_S6_WR_CURRENT - O2 Sensor 6 WR Lambda Current - Min=0 Max=2
0129 - O2_S6_WR_VOLTAGE - O2 Sensor 6 WR Lambda Voltage - Min=0 Max=2
013A - O2_S7_WR_CURRENT - O2 Sensor 7 WR Lambda Current - Min=0 Max=2
012A - O2_S7_WR_VOLTAGE - O2 Sensor 7 WR Lambda Voltage - Min=0 Max=2
013B - O2_S8_WR_CURRENT - O2 Sensor 8 WR Lambda Current - Min=0 Max=2
012B - O2_S8_WR_VOLTAGE - O2 Sensor 8 WR Lambda Voltage - Min=0 Max=2
0113 - O2_SENSORS - O2 Sensors Present - Min=0 Max=0
011D - O2_SENSORS_ALT - O2 Sensors Present (alternate) - Min=0 Max=0
011C - OBD_COMPLIANCE - OBD Standards Compliance - Min=0 Max=0
0120 - PIDS_B - Supported PIDs [21-40] - Min=0 Max=0
0140 - PIDS_C - Supported PIDs [41-60] - Min=0 Max=0
0145 - RELATIVE_THROTTLE_POS - Relative throttle position - Min=0 Max=100
010C - RPM - Engine RPM - Min=0 Max=16383.8
011F - RUN_TIME - Engine Run Time - Min=0 Max=65535
014D - RUN_TIME_MIL - Time run with MIL on - Min=0 Max=65535
0106 - SHORT_FUEL_TRIM_1 - Short Term Fuel Trim - Bank 1 - Min=-100 Max=99.2
0108 - SHORT_FUEL_TRIM_2 - Short Term Fuel Trim - Bank 2 - Min=-100 Max=99.2
010D - SPEED - Vehicle Speed - Min=0 Max=255
0101 - STATUS - Status since DTCs cleared - Min=0 Max=0
0141 - STATUS_DRIVE_CYCLE - Monitor status this drive cycle - Min=0 Max=0
014C - THROTTLE_ACTUATOR - Commanded throttle actuator - Min=0 Max=100
0111 - THROTTLE_POS - Throttle Position - Min=0 Max=100
0147 - THROTTLE_POS_B - Absolute throttle position B - Min=0 Max=100
0148 - THROTTLE_POS_C - Absolute throttle position C - Min=0 Max=100
014E - TIME_SINCE_DTC_CLEARED - Time since trouble codes cleared - Min=0 Max
    =65535
010E - TIMING_ADVANCE - Timing Advance - Min=-64 Max=63.5
0130 - WARMUPS_SINCE_DTC_CLEAR - Number of warm-ups since codes cleared - Min=0
     Max=255
...
{
    "description": "Vehicle Speed",
    "name": "SPEED",
    "units": "km/h",
    "value": "42.000000"
}
{
    "description": "Vehicle Speed",
    "name": "SPEED",
    "units": "km/h",
    "value": "53.000000"
}
{
```

```
    "description": "Vehicle Speed",
    "name": "SPEED",
    "units": "km/h",
    "value": "63.000000"
}
...
{
    "description": "Vehicle Speed",
    "name": "SPEED",
    "units": "km/h",
    "value": "165.000000"
}
{
    "description": "Vehicle Speed",
    "name": "SPEED",
    "units": "km/h",
    "value": "175.000000"
}
{
    "description": "Vehicle Speed",
    "name": "SPEED",
    "units": "km/h",
    "value": "185.000000"
}
===============================================================================
All tests passed (276 assertions in 26 test cases)
```

Para entender un poco mejor los resultados obtenidos se explica brevemente las pruebas realizadas. En primer lugar, para la comprobación de los DTC ha sido necesario utilizar el tipo de simulador obdsim Error el cuál ofrece una serie de DTC prefijados en el vehículo.

En segundo lugar, se ha utilizado el tipo de simulador obsim Cycler, que proporciona un ciclo continuo de distintos valores para cada sensor disponible. Este tipo de simulador no ofrece el VIN del vehículo pero si un mayor número de comandos disponibles que el tipo gui_fltk de la Figura 7.1. Por este motivo, se realiza la prueba de mostrar los comandos disponibles y ejecutar en un bucle la monitorización de la velocidad del vehículo, obteniendo un dato cada segundo durante quince segundos, pudiendo observarse su evolución. Por último, se observa el número de test pasados exitosamente correspondientes a las pruebas comentadas anteriormente, junto con las pruebas unitarias de decodificadores y funciones de conversión OBD.

Por último, se muestran los resultados obtenidos en un vehículo real de la prueba de obtención del número de identificación del vehículo, ya que, este dato no se pudo obtener con el simulador. La marca y modelo del vehículo en el que se ha realizado la prueba es Opel Corsa.

Código 7.2 Ejemplo de ejecución de prueba VIN en vehículo.

```
[LOG][test/../src/Obd.hpp][send][Line 273] Mensaje a enviar: 0902
[LOG][test/../src/Obd.hpp][send][Line 274] Enviando mensaje...
[LOG][test/../src/Obd.hpp][polling][Line 300] Polling function
[LOG][test/../src/Obd.hpp][polling][Line 340] Mensaje recibido:


014
0:490201573056
1:3058455036384A
2:34313430303530

>
[LOG][test/../src/Obd.hpp][polling][Line 349] Información: 01573056
```

```
1:3058455036384A
2:34313430303530
GET_VIN - Get Vehicle Identification Number - Min=0 Max=0
-> W0V0XEP68J4140050
-----------------------------------------------------------
```

Para concluir, podemos analizar el VIN para obtener información de él.

- **Tres primeros dígitos - W0V.** Es el Identificador Mundial del Fabricante (WMI). El primer dígito identifica el país de la empresa fabricante, en este caso, W corresponde con Alemania. El segundo dígito identifica a la empresa fabricante, en este caso, 0 corresponde con Opel. El tercer caracter (V) junto con la información de los dos primeros puede identificar el tipo de vehículo (camión, SUV o coche).
- **Del cuarto a octavo dígito - 0XEP6.** Es la Sección del Descriptor del Vehículo (VDS). Identifica el modelo, el tipo de carrocería, el motor y la transmisión entre otras cosas.
- **Del noveno a último dígito - 8J4140050.** Es la Sección de Identificación del Vehículo (VIS). Se obtiene información de identificación del vehículo.

8 Conclusiones

8.1 Conclusión y línea futura de trabajo

Este proyecto tenía como fin el desarrollo de una herramienta que permita la comunicación con el vehículo para obtener información relevante de éste. Durante el desarrollo de este trabajo se ha llevado a cabo un estudio del funcionamiento básico de los distintos componentes del vehículo, la comunicación entre ellos y la comunicación con el dispositivo ELM327 a través de la interfaz habilitada para este fin. Sin este estudio previo habría sido complicado poder entender el alcance y las distintas dificultades que podría encontrarme en el camino.

Por otra parte, se ha realizado un análisis más profundo del formato e intercambio de mensajes necesarios para la comunicación con el vehículo, ya que, era parte esencial para obtener los datos como los códigos de fallo. Además, se ha realizado un estudio del funcionamiento bluetooth y las librerías necesarias en C/C++ de forma general para cualquier dispositivo, para así poder aplicarlo al caso en concreto del dispositivo ELM327.

Disponiendo de toda esta información, se ha llevado a cabo un estudio del sistema en base a conocimientos de ingeniería de software. La especificación de requisitos ha permitido mostrar aspectos que a simple vista no podía haber tenido en cuenta y durante el diseño de la arquitectura se ha conseguido estructurar de forma más clara el código desarrollado. Finalmente, se ha comprobado que el producto final del desarrollo software funcionaba correctamente, con una propuesta de sistema completo que podría utilizarse de forma funcional.

Sin embargo, al tratarse de un sistema muy extenso y complejo, no se ha podido abarcar todas las posibilidades en este proyecto, por lo que podrían destacarse algunos aspectos de mejora y líneas futuras de trabajo para mejorarlo.

En primer lugar, podrían implementarse más PID genéricos o incluso implementar algún PID propio de algún fabricante específico. Para el objetivo de este proyecto no era necesario implementar todos los PID, por lo que se han limitado a los 80 primeros. En segundo lugar, se podrían desarrollar más modos de operación definididos en la Tabla 3.1, ya que para este proyecto se han definido los modos de operación 01 (parámetros), 03 (DTC) y 09 (VIN). Se podría añadir la funcionalidad a la librería de poder eliminar datos almacenados como los DTC, para que en el caso de resolución de la avería, eliminar el error de la ECU correspondiente. Por último, otra opción de mejora podría ser el desarrollo de funcionalidades en la interfaz web, permitiendo la visualización mediante gráficas de los valores del vehículos cómo la velocidad o la visualización de la geolocalización del vehículo mediante un mapa.

Para terminar, como conclusión personal y profesional, cabe destacar que este trabajo me ha proporcionado nuevos conocimientos del mundo automovilístico permitiéndome utilizar conceptos aprendidos durante el periodo universitario para desarrollar un producto final de mucha utilidad en la actualidad y que puede servir muchas personas en sus proyectos personales.

Anexo A
Manual de instalación

Con el propósito de dotar de documentación para una correcta instalación de la librería y del entorno desarrollado, en este anexo, se detallan los pasos a seguir para conseguir tal fin.

Para empezar, es importante conocer qué herramientas y procesos son necesarios en cada componente del sistema para llevar a cabo el proceso de instalación. En la Figura A.1 se puede observar cómo es necesario tener un proceso monDTC y gpsd ejecutándose en cada Raspberry Pi y, por tanto, en cada vehículo, para el envío de los mensajes al servidor remoto. Por otro lado, sólo necesitamos un servidor remoto con un servidor web Node.js y una única base de datos MongoDB.

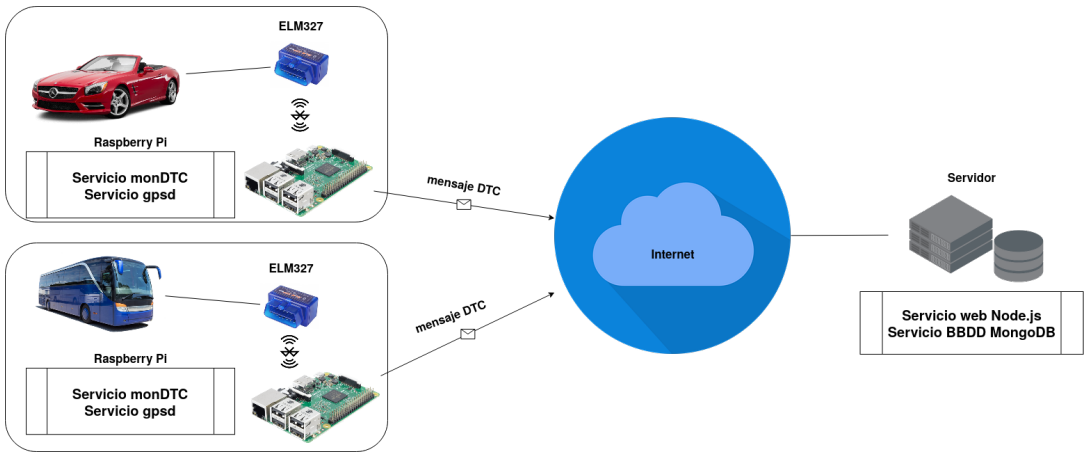


Figura A.1 Arquitectura general y componentes para la instalación.

A.1 Instalación en Raspberry Pi

En primer lugar, se debe obtener el código fuente del repositorio de GitHub de la librería y aplicación monDTC para instalarla como servicio. Para ello se ejecuta las siguientes líneas (se necesita la herramienta git):

Código A.1 Descarga del código fuente obd2-bluetooth.

```
sudo apt-get install git
git clone https://github.com/sergiorg96/obd2-bluetooth
```

Tras esto, se procede con la configuración del archivo de parámetros configuration.cfg con el editor de texto que se desee, en mi caso, nano:

Código A.2 Configuración del fichero configuration.cfg.

```
cd obd2-bluetooth
obd2-bluetooth$ nano conf/configuration.cfg
```

Se comprueba que los valores especificados sean los correctos para el caso concreto de la instalación. En el Código A.3 se detalla en los propios comentarios la función de cada parámetro en el fichero configuration.cfg.

Código A.3 Descripción de variables de configuration.cfg.

```
###
# BT-DISP-NAME
# Nombre del dispositivo Bluetooth ELM327 que se necesita encontrar en el
# descubrimiento bluetooth. Este parámetro depende del fabricante del
# ELM327, ya que, está programado en el dispositivo, por lo que es
# necesario conocerlo antes de configurar este archivo.
#
# Generalmente este nombre es OBDII
###

BT-DISP-NAME=OBDII

###
# PIN
# Contraseña por defecto del dispositivo bluetooth de ELM327 para el
# emparejamiento bluetooth. Depende del fabricante del ELM327.
#
# Generalmente es 1234
###

PIN=1234

###
# IP
# Dirección IP del servidor remoto dónde se envían los mensajes
# con el DTC detectado.
###

IP=127.0.0.1

###
# PORT
# Puerto UDP de escucha del servidor remoto dónde se envían los mensajes
# con el DTC detectado.
##

PORT=33333

###
# PERIOD
# Tiempo de espera entre solicitudes DTC al vehículo.
# Si se desea reducir el número de solicitudes al vehículo,
# es necesario aumentar este valor.
###

PERIOD=30
```

```
###
# PORT-GPS
# Puerto UDP de escucha del servicio gpsd para la obtención
# de la geolocalización del vehículo.
###

PORT-GPS=5555

###
# ALARM-FILE-NAME
# Nombre del fichero de alarma de la clase AlarmFile que puede utilizarse para
# almacenar información del último mensaje enviado.
###
ALARM-FILE-NAME=alarmOBD

###
# INTERFACE-NAME
# Nombre de la interfaz de red del dispositivo donde se ejecuta monDTC con
# conexión a Internet, utilizada para la identificación del vehículo.
###

INTERFACE-NAME=wlp2s0
```

El siguiente paso es compilar la librería generando el fichero libobd2-bluetooth.a en la carpeta lib que genera el propio Makefile y a su vez, compilar el programa que utiliza la librería. Se necesita la librería bluetooth Bluez instalada para la correcta compilación. Para ello se ejecutan los siguientes comandos:

Código A.4 Compilación con make.

```
obd2-bluetooth$ sudo apt-get install libbluetooth-dev
obd2-bluetooth$ make
g++ -std=c++17 -Wall -O -c src/main.cpp
g++ -std=c++17 -Wall -O -c -x c++ src/Obd.hpp src/Commands.hpp src/debug.hpp
g++ -std=c++17 -Wall -O -c src/decoders.cpp src/decoders.hpp
g++ -std=c++17 -Wall -O -c src/alarmfile.cpp src/alarmfile.hpp
g++ -std=c++17 -Wall -O -c src/loadcfg.cpp src/loadcfg.hpp
g++ -std=c++17 -Wall -O -c src/gpsclient.cpp src/gpsclient.hpp
if [ ! -d "lib" ]; then mkdir lib; fi
ar rcs lib/libobd2-bluetooth.a Obd.o decoders.o alarmfile.o loadcfg.o gpsclient
    .o
g++ -std=c++17 -Wall -o monDTC main.o -Llib -lobd2-bluetooth -lbluetooth -
    lpthread
```

Después de esta ejecución obtendremos, el ejecutable "monDTC" y la librería libobd2-bluetooth.a en la carpeta lib. Están disponbiles también las siguientes opciones en el Makefile para su utilización:

- **make clean:** Elimina los ficheros .o y .gch generados en la compilación de la librería para obtener libobd2-bluetooth.a .
- **make install:** Se utiliza para la instalación de la aplicación como servicio en la Raspberry Pi. Se explica a continuación.
- **make monDTC-debug:** Obtiene el ejecutable monDTC-debug que tiene la misma funcionalidad que monDTC pero muestra mensajes de depuración de la aplicación. Útil para el desarrollo de nuevas funcionalidades.
- **make test:** Genera el ejecutable test en la carpeta de test para la ejecución de la batería de pruebas unitarias y de integración.

- **make libobd2-bluetooth.a:** Obtiene únicamente la librería libobd2-bluetooth con extensión .a
- **make all:** Ejecución de las opciones monDTC, monDTC-debug, test y clean.

Para configurar la aplicación como un servicio en el sistema disponemos del fichero de configuración del servicio monDTC.service en la carpeta conf el cual se muestra a continuación:

Código A.5 Descripción del fichero de configuración del servicio.

```
# ***monDTC.service***
[Unit]
Description=Servicio de obtención de DTC de un vehículo y envío a servidor
    remoto
After=multi-user.target

[Service]
Type=simple
ExecStart=/RUTA-AL-DIRECTORIO-DE-DESCARGA-GITHUB/obd2-bluetooth/monDTC
User=USUARIO-DEL-SISTEMA
RestartSec=10
WorkingDirectory=/RUTA-AL-DIRECTORIO-DE-DESCARGA-GITHUB/obd2-bluetooth
Restart=always
StandardOutput=syslog
StandardError=syslog

[Install]
WantedBy=multi-user.target
```

En este fichero es necesario modificar los siguientes parámetros:

- **WorkingDirectory y ExecStart:** Hay que añadir la ruta al directorio dónde se encuentra la carpeta descargada de GitHub con el ejecutable monDTC.
- **User:** Hay que configurar el usuario del sistema que ejecutará el servicio, por ejemplo, root.

Una vez se hayan configurado estos parámetros podemos ejecutar "sudo make install" para la instalación del servicio. En el proceso de instalación se ejecuta el script de la carpeta conf "installService.sh" que copia el fichero de configuración "monDTC.service" en la ruta /lib/systemd/system del sistema operativo.

Código A.6 Instalación de la aplicación como servicio en Linux.

```
obd2-bluetooth$ sudo make install
./conf/installService.sh
```

Finalizaría así la instalación del servicio monDTC en la Raspberry Pi.

A.2 Instalación en el servidor

Al igual que con obd2-bluetooth, se debe obtener el código fuente del repositorio de GitHub del servidor remoto. Para ello se ejecuta las siguientes líneas (se necesita la herramienta git):

Código A.7 Descarga del código fuente de obd2-server.

```
sudo apt-get install git
git clone https://github.com/sergiorg96/obd2-server
```

Para la ejecución del servidor web, es necesario disponer de Node.js, NPM y todas las dependencias del proyecto. Se puede instalar de la siguiente forma:

Código A.8 Instalación de Node.js.

```
sudo apt-get install nodejs npm
cd obd2-server
obd2-server$ npm install
```

Por último, es necesario instalar MongoDB para almacenar la información enviada por las Raspberry Pi de los vehículos. Para ello seguimos las instrucciones de la web oficial de MongoDB [14] añadiendo el repositorio de la distribución correspondiente y lo instalamos. En este ejemplo, para Debian 10 "Buster":

Código A.9 Instalación de MongoDB.

```
wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add
     -

echo "deb http://repo.mongodb.org/apt/debian buster/mongodb-org/4.4 main" |
    sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list

sudo apt-get update

sudo apt-get install mongodb-org
```

Tras esto, es posible ejecutar el servidor tras iniciar el servicio de MongoDB, tal y cómo se muestra en el Anexo B.

A.3 Generación del ejecutable para las pruebas

Las pruebas se deben realizar en el dispositivo dónde se vaya a ejecutar la herramienta monDTC (Raspberry Pi). Para realizar la batería de pruebas desarrolladas es necesario tener instaladas las herramientas minicom y obdsim:

Código A.10 Instalación obdsim y minicom.

```
sudo apt-get install obdgpslogger minicom
```

Para ejecutar las pruebas se necesita el ejecutable que se genera de la siguiente forma con make:

Código A.11 Generar ejecutable de las pruebas.

```
obd2-bluetooth$ make test
```

Finalmente, ejecutamos las pruebas de la siguiente forma:

Código A.12 Ejecución de las pruebas con el ejecutable test.

```
obd2-bluetooth$ ./test/test
```

Anexo B
Manual de uso de la librería y entorno

Este manual de uso sirve de referencia para entender cómo usar el entorno desarrollado para este proyecto e incluye documentación de la librería obd2-bluetooth obtenida con la herramienta Doxygen [15]

B.1 Utilización de monDTC en Raspberry Pi

Una vez configurado el servicio monDTC como se describe en el Anexo A, podemos habilitarlo para que inicie al arranque del sistema operativo, arrancarlo, pararlo y comprobar su estado:

Código B.1 Operaciones disponibles del servicio con systemd.

```
obd2-bluetooth$ systemctl status monDTC
    monDTC.service - Servicio de obtención de DTC de un vehículo y envío a
    servidor remoto
   Loaded: loaded (/lib/systemd/system/monDTC.service; disabled; vendor preset:
        enabled)
   Active: inactive (dead)

obd2-bluetooth$ sudo systemctl enable monDTC
Created symlink /etc/systemd/system/multi-user.target.wants/monDTC.service  /
    lib/systemd/system/monDTC.service.

obd2-bluetooth$ sudo systemctl start monDTC
obd2-bluetooth$ systemctl status monDTC
    monDTC.service - Servicio de obtención de DTC de un vehículo y envío a
    servidor remoto
   Loaded: loaded (/lib/systemd/system/monDTC.service; enabled; vendor preset:
       enabled)
   Active: active (running) since Mon 2020-09-07 00:02:53 CEST; 1s ago
 Main PID: 8204 (main)
    Tasks: 1 (limit: 4569)
   CGroup: /system.slice/monDTC.service
           8204    /home/sergiorg/GitHubProjects//monDTC/monDTC

sep 07 00:02:53 sergiorg systemd[1]: Started Servicio de obtención de DTC de un
     vehículo y envío a servidor remoto

obd2-bluetooth$ sudo systemctl stop monDTC
```

B.2 Utilización de obd2-server en el servidor remoto

Una vez configurado e instalado el servicio monDTC, se procede con el servidor remoto. Para ello, es necesario tener arrancado el servicio de MongoDB.

Código B.2 Comprobación y arranque del servicio mongod.

```
obd2-server$ sudo systemctl start mongod

obd2-server$ systemctl status mongod
    mongod.service - MongoDB Database Server
   Loaded: loaded (/lib/systemd/system/mongod.service; disabled; vendor preset:
        enabled)
   Active: active (running) since Mon 2020-09-07 00:14:36 CEST; 16s ago
     Docs: https://docs.mongodb.org/manual
 Main PID: 8483 (mongod)
   CGroup: /system.slice/mongod.service
           8483    /usr/bin/mongod --config /etc/mongod.conf

sep 07 00:14:36 sergiorg systemd[1]: Started MongoDB Database Server.
```

Ejecutamos el siguiente comando para arrancar tanto el servidor web, como el servidor UDP para la recepción de datos con el vehículo.

Código B.3 Iniciar el servidor remoto.

```
obd2-server$ DEBUG=obd2-server:* npm run start

> obd2-server@0.0.0 start /home/sergiorg/Descargas/pruebaGit/obd2-server
> node ./bin/www

Servidor UDP escuchando en 127.0.0.1:33333
  obd2-server:server Listening on port 8080 +0ms
Mongoose connection open
```

Podemos comprobar desde un navegador, accediendo a la dirección IP del servidor en el puerto 8080 que muestra la interfaz web para la visualización de DTC.

obd2-bluetooth

1.0

Generado por Doxygen 1.8.13

Índice general

1 obd2-bluetooth **1**

2 Índice de estructura de datos **3**

2.1 Estructura de datos 3

3 Indice de archivos **5**

3.1 Lista de archivos 5

4 Documentación de las estructuras de datos **7**

4.1 Referencia de la Clase AlarmFile 7

4.1.1 Descripción detallada 7

4.1.2 Documentación del constructor y destructor 7

4.1.2.1 AlarmFile() 7

4.1.3 Documentación de las funciones miembro 8

4.1.3.1 sendAlarm() 8

4.2 Referencia de la Clase Commands 8

4.2.1 Descripción detallada 10

4.2.2 Documentación del constructor y destructor 10

4.2.2.1 Commands() 10

4.2.3 Documentación de las funciones miembro 10

4.2.3.1 getBytesResponse() 10

4.2.3.2 getCMD() 11

4.2.3.3 getCMDResponse() 11

4.2.3.4 getDecoder() 11

4.2.3.5 getDescription() 12

4.2.3.6 getJson() . . . 12
4.2.3.7 getMAX() . . . 12
4.2.3.8 getMIN() . . . 13
4.2.3.9 getName() . . . 13
4.2.3.10 getResValue() . . . 13
4.2.3.11 getTypeData() . . . 13
4.2.3.12 getUnits() . . . 14
4.2.3.13 setBytesResponse() . . . 14
4.2.3.14 setCMD() . . . 14
4.2.3.15 setDecoder() . . . 14
4.2.3.16 setDescription() . . . 15
4.2.3.17 setMAX() . . . 15
4.2.3.18 setMIN() . . . 15
4.2.3.19 setName() . . . 16
4.2.3.20 setResValue() . . . 16
4.2.3.21 setTypeData() . . . 16
4.2.3.22 setUnits() . . . 17
4.3 Referencia de la Clase GpsClient . . . 17
4.3.1 Descripción detallada . . . 17
4.3.2 Documentación del constructor y destructor . . . 17
4.3.2.1 GpsClient() . . . 17
4.3.3 Documentación de las funciones miembro . . . 18
4.3.3.1 getGPS() . . . 18
4.4 Referencia de la Clase Obd . . . 18
4.4.1 Descripción detallada . . . 19
4.4.2 Documentación del constructor y destructor . . . 19
4.4.2.1 Obd() . . . 19
4.4.3 Documentación de las funciones miembro . . . 20
4.4.3.1 connectBluetooth() . . . 20
4.4.3.2 disconnectBluetooth() . . . 20

4.4.3.3 discoverDeviceAddress() . . . 20
4.4.3.4 existPID() . . . 21
4.4.3.5 getDTCs() . . . 21
4.4.3.6 getVIN() . . . 22
4.4.3.7 initDecoderFunctions() . . . 22
4.4.3.8 initMessages() . . . 22
4.4.3.9 isValid() . . . 22
4.4.3.10 polling() . . . 22
4.4.3.11 printPIDs() . . . 23
4.4.3.12 printStatus() . . . 23
4.4.3.13 readFileData() . . . 23
4.4.3.14 send() . . . 23
4.4.4 Documentación de los campos . . . 24
4.4.4.1 map_commands . . . 24
4.5 Referencia de la Estructura OxigenoResponse . . . 24
4.5.1 Descripción detallada . . . 24
4.5.2 Documentación de los campos . . . 24
4.5.2.1 A . . . 25
4.5.2.2 B . . . 25
4.6 Referencia de la Estructura RelacionesResponse . . . 25
4.6.1 Descripción detallada . . . 25
4.6.2 Documentación de los campos . . . 25
4.6.2.1 A . . . 26
4.6.2.2 B . . . 26
4.6.2.3 C . . . 26
4.6.2.4 D . . . 26

5 Documentación de archivos **27**

5.1 Referencia del Archivo alarmfile.cpp . . . 27

5.1.1 Descripción detallada . . . 27

5.2 Referencia del Archivo alarmfile.hpp . . . 28

5.2.1 Descripción detallada . . . 28

5.3 Referencia del Archivo Commands.hpp . . . 29

5.3.1 Descripción detallada . . . 29

5.4 Referencia del Archivo debug.hpp . . . 30

5.4.1 Descripción detallada . . . 31

5.5 Referencia del Archivo decoders.cpp . . . 31

5.5.1 Descripción detallada . . . 32

5.5.2 Documentación de las funciones . . . 33

5.5.2.1 convertDTCs() . . . 33

5.5.2.2 decodeAjusteCombustibleEGR() . . . 33

5.5.2.3 decodeAvanceTiempo() . . . 33

5.5.2.4 decodeCargaPosicionEGR() . . . 34

5.5.2.5 decodeDescribeProtocol() . . . 34

5.5.2.6 decodeDTCs() . . . 35

5.5.2.7 decodeHexToDec() . . . 35

5.5.2.8 decodePIDS() . . . 35

5.5.2.9 decodePresionCombColector() . . . 36

5.5.2.10 decodePresionCombustible() . . . 36

5.5.2.11 decodePresionMedidorCombustible() . . . 36

5.5.2.12 decodePresionVapor() . . . 37

5.5.2.13 decodeRelacionCombAire() . . . 37

5.5.2.14 decodeRelacionCombAireActual() . . . 38

5.5.2.15 decodeRelacionCombAireBasica() . . . 38

5.5.2.16 decodeRelaciones() . . . 38

5.5.2.17 decodeRPM() . . . 39

5.5.2.18 decodeSensorOxigeno() . . . 39

5.5.2.19 decodeStatus() . . . 39
5.5.2.20 decodeTempCatalizador() . . . 40
5.5.2.21 decodeTempGeneral() . . . 40
5.5.2.22 decodeVelocidadMAF() . . . 41
5.5.2.23 decodeVIN() . . . 41
5.5.2.24 decodeVoltajeControl() . . . 41
5.6 Referencia del Archivo decoders.hpp . . . 42
5.6.1 Descripción detallada . . . 44
5.6.2 Documentación de los 'defines' . . . 44
5.6.2.1 PID_BITS . . . 44
5.6.2.2 STATUS_BITS . . . 44
5.6.3 Documentación de las funciones . . . 45
5.6.3.1 convertDTCs() . . . 45
5.6.3.2 decodeAjusteCombustibleEGR() . . . 45
5.6.3.3 decodeAvanceTiempo() . . . 45
5.6.3.4 decodeCargaPosicionEGR() . . . 46
5.6.3.5 decodeDescribeProtocol() . . . 46
5.6.3.6 decodeDTCs() . . . 47
5.6.3.7 decodeHexToDec() . . . 47
5.6.3.8 decodePIDS() . . . 47
5.6.3.9 decodePresionCombColector() . . . 48
5.6.3.10 decodePresionCombustible() . . . 48
5.6.3.11 decodePresionMedidorCombustible() . . . 48
5.6.3.12 decodePresionVapor() . . . 49
5.6.3.13 decodeRelacionCombAire() . . . 49
5.6.3.14 decodeRelacionCombAireActual() . . . 50
5.6.3.15 decodeRelacionCombAireBasica() . . . 50
5.6.3.16 decodeRelaciones() . . . 50
5.6.3.17 decodeRPM() . . . 51
5.6.3.18 decodeSensorOxigeno() . . . 51

5.6.3.19 decodeStatus() . . . 51
5.6.3.20 decodeTempCatalizador() . . . 52
5.6.3.21 decodeTempGeneral() . . . 52
5.6.3.22 decodeVelocidadMAF() . . . 53
5.6.3.23 decodeVIN() . . . 53
5.6.3.24 decodeVoltajeControl() . . . 53
5.7 Referencia del Archivo gpsclient.cpp . . . 54
5.7.1 Descripción detallada . . . 54
5.7.2 Documentación de los 'defines' . . . 54
5.7.2.1 NOGPSDATA . . . 55
5.8 Referencia del Archivo gpsclient.hpp . . . 55
5.8.1 Descripción detallada . . . 56
5.9 Referencia del Archivo loadcfg.cpp . . . 56
5.9.1 Descripción detallada . . . 57
5.9.2 Documentación de las funciones . . . 57
5.9.2.1 getmac() . . . 57
5.9.2.2 loadCfg() . . . 57
5.9.2.3 shit() . . . 58
5.10 Referencia del Archivo loadcfg.hpp . . . 58
5.10.1 Descripción detallada . . . 59
5.10.2 Documentación de las funciones . . . 59
5.10.2.1 getmac() . . . 59
5.10.2.2 loadCfg() . . . 60
5.10.2.3 shit() . . . 60
5.11 Referencia del Archivo MockSocket.cpp . . . 60
5.11.1 Descripción detallada . . . 61
5.11.2 Documentación de las funciones . . . 61
5.11.2.1 findDevPTS() . . . 62
5.12 Referencia del Archivo Obd.hpp . . . 62
5.12.1 Descripción detallada . . . 63
5.12.2 Documentación de los 'defines' . . . 63
5.12.2.1 MAX_EP_EVTS . . . 64
5.13 Referencia del Archivo UnitTestCase.cpp . . . 64
5.13.1 Descripción detallada . . . 65
5.13.2 Documentación de los 'defines' . . . 66
5.13.2.1 BUFSIZE . . . 66
5.13.2.2 CATCH_CONFIG_MAIN . . . 66
5.13.2.3 WAIT_OBDSIM . . . 66
5.13.3 Documentación de las funciones . . . 66
5.13.3.1 getMinicomCMD() . . . 66
5.13.3.2 initOBDSIM() . . . 67

Índice **69**

Capítulo 1

obd2-bluetooth

Aplicación y librería para la comunicación OBD con vehículos desarrollada en C++.

Capítulo 2

Índice de estructura de datos

2.1. Estructura de datos

Lista de estructuras con una breve descripción:

AlarmFile
: Clase que representa la conexión con el servidor remoto para el envío de un mensaje (alarma) 7

Commands
: Clase que representa los comandos AT y PIDS que se necesitan en el intercambio de mensajes con el dispositivo ELM327 8

GpsClient
: Clase que representa la conexión con el servicio gpsd para obtener las coordenadas GPS . . 17

Obd
: Clase que representa el acceso a la conexión con el dispositivo ELM327 18

OxigenoResponse
: Estructura de datos para las respuesta de dos valores en PIDS relacionados con gases de escape 24

RelacionesResponse
: Estructura de datos para las respuesta de cuatro valores en PIDS relacionados con gases de escape 25

Capítulo 3

Indice de archivos

3.1. Lista de archivos

Lista de todos los archivos documentados y con descripciones breves:

alarmfile.cpp
Archivo que contiene la definición de la clase para la configuración y envío de un mensaje a un servidor remoto . . . 27
alarmfile.hpp
Archivo que contiene la declaración de la clase para la configuración y envío de un mensaje a un servidor remoto . . . 28
Commands.hpp
Archivo que contiene la clase con la definición de la estructura de los comandos AT y OBD . . 29
debug.hpp
Archivo que contiene las funciones de debug en la salida estándar y de error del sistema . . . 30
decoders.cpp
Archivo que contiene la definición de las funciones de decodificación de las respuestas del dispositivo ELM327 . . . 31
decoders.hpp
Archivo que contiene la declaración de las funciones de decodificación de las respuestas del dispositivo ELM327 . . . 42
gpsclient.cpp
Archivo que contiene la definición de la clase para la conexión con el servicio gpsd de obtención de coordenadas GPS . . . 54
gpsclient.hpp
Archivo que contiene la declaración de la clase para la conexión con el servicio gpsd de obtención de coordenadas GPS . . . 55
loadcfg.cpp
Archivo que contiene la definición de las funciones para la lectura de un fichero de configuración del tipo clave=valor . . . 56
loadcfg.hpp
Archivo que contiene la declaración de las funciones para la lectura de un fichero de configuración del tipo clave=valor . . . 58
main.cpp . . . **??**
MockSocket.cpp
Archivo que contiene las funciones mock bluetooth para poder realizar las pruebas de integración . . . 60
Obd.hpp
Archivo que contiene la clase con la implementación de la conexión y envío de mensajes OBD con el dispositivo ELM327 . . . 62
UnitTestCase.cpp
Archivo que contiene el conjunto de pruebas unitarias y de integración del sistema . . . 64

Capítulo 4

Documentación de las estructuras de datos

4.1. Referencia de la Clase AlarmFile

Clase que representa la conexión con el servidor remoto para el envío de un mensaje (alarma).

`#include <alarmfile.hpp>`

Métodos públicos

- AlarmFile (std::string AlarmHost, std::string AlarmPort, std::string AlarmFilename, std::string LastAlarm↩ Filename)

  Constructor de la clase AlarmFile.
- bool sendAlarm (std::string msg)

  Método para enviar el mensaje/alarma al servidor remoto.

4.1.1. Descripción detallada

Clase que representa la conexión con el servidor remoto para el envío de un mensaje (alarma).

Clase utilizada para el envío de datos del vehículo al servidor remoto.

Definición en la línea 21 del archivo alarmfile.hpp.

4.1.2. Documentación del constructor y destructor

4.1.2.1. AlarmFile()

```
AlarmFile::AlarmFile (
            std::string AlarmHost,
            std::string AlarmPort,
            std::string AlarmFilename,
            std::string LastAlarmFilename )
```

Constructor de la clase AlarmFile.

Parámetros

AlarmHost	String con la dirección IP del servidor remoto.
AlarmPort	String con el puerto de conexión del servidor remoto.
AlarmFilename	String con el nombre del archivo de almacenamiento de la alarma.
LastAlarmFilename	String con el nombre del último archivo de almacenamiento de la alarma.

Devuelve

Devuelve una instancia de la clase AlarmFile.

Definición en la línea 26 del archivo alarmfile.cpp.

4.1.3. Documentación de las funciones miembro

4.1.3.1. sendAlarm()

```
bool AlarmFile::sendAlarm (
            std::string msg )
```

Método para enviar el mensaje/alarma al servidor remoto.

Parámetros

msg	String con el mensaje a enviar al servidor remoto.

Devuelve

Booleano, true si el mensaje fue enviado correctamente y false en caso contrario.

Definición en la línea 44 del archivo alarmfile.cpp.

La documentación para esta clase fue generada a partir de los siguientes ficheros:

- alarmfile.hpp
- alarmfile.cpp

4.2. Referencia de la Clase Commands

Clase que representa los comandos AT y PIDS que se necesitan en el intercambio de mensajes con el dispositivo ELM327.

```
#include <Commands.hpp>
```

Métodos públicos

- Commands (json data)

  Constructor de la clase Commands.
- std::string getName ()

  Método que obtiene el nombre del comando.
- std::string getDescription ()

  Método que obtiene la descripción del comando.
- std::string getCMD ()

  Método que obtiene el comando en hexadecimal que se envía al dispositivo ELM327.
- int getBytesResponse ()

  Método que obtiene el número de bytes de la respuesta del comando a enviar.
- std::string getDecoder ()

  Método que obtiene la función de decodificación que se debe ejecutar en la respuesta.
- float getMIN ()

  Método que obtiene el valor mínimo que puede tener la respuesta al comando.
- float getMAX ()

  Método que obtiene el valor máximo que puede tener la respuesta al comando.
- std::string getUnits ()

  Método que obtiene en qué unidades se mide la respuesta del comando.
- std::string getTypeData ()

  Método que obtiene el tipo de dato que se obtiene en la respuesta del comando.
- std::any getResValue ()

  Método que obtiene el valor decodificado de la respuesta del comando.
- json getJson ()

  Método que obtiene información del comando y el valor de la respuesta decodificado en formato JSON.
- std::string getCMDResponse ()

  Método para obtener el comando de respuesta al PID solicitado.
- void setName (std::string name)

  Método para asignar un nombre a un comando.
- void setDescription (std::string description)

  Método para asignar una descripción a un comando.
- void setCMD (std::string cmd)

  Método para asignar el comando en hexadecimal a enviar al dispositivo ELM327.
- void setBytesResponse (int bytes_response)

  Método para asignar el número de bytes de respuesta a un comando.
- void setDecoder (std::string decoder)

  Método para asignar un decodificador a un comando.
- void setMIN (float min_unit)

  Método para asignar el valor mínimo de la respuesta a un comando.
- void setMAX (float max_unit)

  Método para asignar el valor máximo de la respuesta a un comando.
- void setUnits (std::string units)

  Método para asignar las unidades de medida de la respuesta a un comando.
- void setTypeData (std::string type_data)

  Método para asignar el tipo de dato que se debe de obtener en la respuesta del comando.
- void setResValue (auto resValue)

  Método para asignar el valor decodificado de la respuesta al comando.

4.2.1. Descripción detallada

Clase que representa los comandos AT y PIDS que se necesitan en el intercambio de mensajes con el dispositivo ELM327.

Clase utilizada por la clase Obd con la información relativa a los comandos OBD.

Definición en la línea 29 del archivo Commands.hpp.

4.2.2. Documentación del constructor y destructor

4.2.2.1. Commands()

```
Commands::Commands (
            json data )  [inline]
```

Constructor de la clase Commands.

Parámetros

data	Tipo de datos json con la lista de comandos AT y OBD genéricos.

Devuelve

Devuelve una instancia de la clase Commands.

Definición en la línea 38 del archivo Commands.hpp.

4.2.3. Documentación de las funciones miembro

4.2.3.1. getBytesResponse()

```
int Commands::getBytesResponse ( )  [inline]
```

Método que obtiene el número de bytes de la respuesta del comando a enviar.

Devuelve

Entero con el número de bytes de la respuesta del comando a enviar.

Definición en la línea 76 del archivo Commands.hpp.

4.2.3.2. getCMD()

```
std::string Commands::getCMD ( )  [inline]
```

Método que obtiene el comando en hexadecimal que se envía al dispositivo ELM327.

Devuelve

String con el comando en hexadecimal que se envía al dispositivo ELM327.

Definición en la línea 69 del archivo Commands.hpp.

4.2.3.3. getCMDResponse()

```
std::string Commands::getCMDResponse ( )  [inline]
```

Método para obtener el comando de respuesta al PID solicitado.

Devuelve

String con la cadena de respuesta sustituyendo el 0 por el 4 en el mensaje OBD.

Se utiliza para identificar los bytes útiles de la respuesta que se encuentran tras esta cadena.

Definición en la línea 195 del archivo Commands.hpp.

4.2.3.4. getDecoder()

```
std::string Commands::getDecoder ( )  [inline]
```

Método que obtiene la función de decodificación que se debe ejecutar en la respuesta.

Devuelve

String del decodificador que se debe ejecutar en la respuesta.

Definición en la línea 83 del archivo Commands.hpp.

4.2.3.5. getDescription()

```
std::string Commands::getDescription ( )  [inline]
```

Método que obtiene la descripción del comando.

Devuelve

String con la descripción del comando.

Definición en la línea 62 del archivo Commands.hpp.

4.2.3.6. getJson()

```
json Commands::getJson ( )  [inline]
```

Método que obtiene información del comando y el valor de la respuesta decodificado en formato JSON.

Devuelve

Tipo json definido con el valor de la respuesta decodificado del comando, su nombre, descripción y unidades.

Función desarrollada con el fin de facilitar el envío de información en formato JSON a un servidor remoto o para el almacenamiento local.

Definición en la línea 132 del archivo Commands.hpp.

4.2.3.7. getMAX()

```
float Commands::getMAX ( )  [inline]
```

Método que obtiene el valor máximo que puede tener la respuesta al comando.

Devuelve

Flotante con el valor máximo que puede tener la respuesta al comando.

Definición en la línea 97 del archivo Commands.hpp.

4.2.3.8. getMIN()

```
float Commands::getMIN ( )  [inline]
```

Método que obtiene el valor mínimo que puede tener la respuesta al comando.

Devuelve

Flotante con el valor mínimo que puede tener la respuesta al comando.

Definición en la línea 90 del archivo Commands.hpp.

4.2.3.9. getName()

```
std::string Commands::getName ( )  [inline]
```

Método que obtiene el nombre del comando.

Devuelve

String con el nombre del comando.

Definición en la línea 55 del archivo Commands.hpp.

4.2.3.10. getResValue()

```
std::any Commands::getResValue ( )  [inline]
```

Método que obtiene el valor decodificado de la respuesta del comando.

Devuelve

El tipo de dato correspondiente con el comando y el valor decodificado de la respuesta.

Definición en la línea 122 del archivo Commands.hpp.

4.2.3.11. getTypeData()

```
std::string Commands::getTypeData ( )  [inline]
```

Método que obtiene el tipo de dato que se obtiene en la respuesta del comando.

Devuelve

String del tipo de dato de la respuesta del comando.

El tipo de dato se utiliza para filtrar entre los tipos de decodificadores de respuesta. Sus valores pueden ser: string, float, vector<int>, vector<string>, map<string, string>, struct OxigenoResponse y struct RelacionesResponse.

Definición en la línea 115 del archivo Commands.hpp.

4.2.3.12. getUnits()

```
std::string Commands::getUnits ( )  [inline]
```

Método que obtiene en qué unidades se mide la respuesta del comando.

Devuelve

String de la unidad de medida de la respuesta del comando.

Definición en la línea 104 del archivo Commands.hpp.

4.2.3.13. setBytesResponse()

```
void Commands::setBytesResponse (
            int bytes_response )  [inline]
```

Método para asignar el número de bytes de respuesta a un comando.

Parámetros

bytes_response	Entero con el número de bytes de respuesta a asignar al comando.

Definición en la línea 228 del archivo Commands.hpp.

4.2.3.14. setCMD()

```
void Commands::setCMD (
            std::string cmd )  [inline]
```

Método para asignar el comando en hexadecimal a enviar al dispositivo ELM327.

Parámetros

cmd	String con el comando en hexadecimal a enviar al dispositivo ELM327.

Definición en la línea 221 del archivo Commands.hpp.

4.2.3.15. setDecoder()

```
void Commands::setDecoder (
            std::string decoder )  [inline]
```

Método para asignar un decodificador a un comando.

Parámetros

decoder	String con el nombre del decodificador que utiliza el comando.

Definición en la línea 235 del archivo Commands.hpp.

4.2.3.16. setDescription()

```
void Commands::setDescription (
            std::string description )  [inline]
```

Método para asignar una descripción a un comando.

Parámetros

description	String con la descripción a asignar al comando.

Definición en la línea 214 del archivo Commands.hpp.

4.2.3.17. setMAX()

```
void Commands::setMAX (
            float max_unit )  [inline]
```

Método para asignar el valor máximo de la respuesta a un comando.

Parámetros

max_unit	Flotante con el valor máximo de la respuesta a un comando.

Definición en la línea 249 del archivo Commands.hpp.

4.2.3.18. setMIN()

```
void Commands::setMIN (
            float min_unit )  [inline]
```

Método para asignar el valor mínimo de la respuesta a un comando.

Parámetros

min_unit	Flotante con el valor mínimo de la respuesta a un comando.

Definición en la línea 242 del archivo Commands.hpp.

4.2.3.19. setName()

```
void Commands::setName (
            std::string name )  [inline]
```

Método para asignar un nombre a un comando.

Parámetros

name	String con el nombre a asignar al comando.

Definición en la línea 207 del archivo Commands.hpp.

4.2.3.20. setResValue()

```
void Commands::setResValue (
            auto resValue )  [inline]
```

Método para asignar el valor decodificado de la respuesta al comando.

Parámetros

resValue	Tipo de dato dependiente del tipo de dato del comando con el valor de la respuesta decodificada.

Función utilizada para el almacenamiento en memoria del valor solicitado con un comando.

Definición en la línea 272 del archivo Commands.hpp.

4.2.3.21. setTypeData()

```
void Commands::setTypeData (
            std::string type_data )  [inline]
```

Método para asignar el tipo de dato que se debe de obtener en la respuesta del comando.

Parámetros

type_data	String con el tipo de dato que se debe de obtener en la respuesta del comando.

Definición en la línea 263 del archivo Commands.hpp.

4.2.3.22. setUnits()

```
void Commands::setUnits (
            std::string units )  [inline]
```

Método para asignar las unidades de medida de la respuesta a un comando.

Parámetros

units	String con las unidades de medida de la respuesta del comando.

Definición en la línea 256 del archivo Commands.hpp.

La documentación para esta clase fue generada a partir del siguiente fichero:

- Commands.hpp

4.3. Referencia de la Clase GpsClient

Clase que representa la conexión con el servicio gpsd para obtener las coordenadas GPS.

```
#include <gpsclient.hpp>
```

Métodos públicos

- GpsClient (std::string GpsPort, std::string validity)
  Constructor de la clase GpsClient.
- std::string getGPS ()
  Método que obtiene una string con las coordenadas GPS.

4.3.1. Descripción detallada

Clase que representa la conexión con el servicio gpsd para obtener las coordenadas GPS.

Clase utilizada para la obtención de coordenadas GPS.

Definición en la línea 28 del archivo gpsclient.hpp.

4.3.2. Documentación del constructor y destructor

4.3.2.1. GpsClient()

```
GpsClient::GpsClient (
            std::string GpsPort,
            std::string validity )
```

Constructor de la clase GpsClient.

Parámetros

GpsPort	String del puerto UDP de conexión con el servicio gpsd.
validity	String con el tiempo máximo de espera en segundos en la recepción del dato GPS.

Devuelve

Devuelve una instancia de la clase GpsClient.

Definición en la línea 13 del archivo gpsclient.cpp.

4.3.3. Documentación de las funciones miembro

4.3.3.1. getGPS()

```
std::string GpsClient::getGPS ( )
```

Método que obtiene una string con las coordenadas GPS.

Devuelve

String con las coordenadas GPS.

Definición en la línea 37 del archivo gpsclient.cpp.

La documentación para esta clase fue generada a partir de los siguientes ficheros:

- gpsclient.hpp
- gpsclient.cpp

4.4. Referencia de la Clase Obd

Clase que representa el acceso a la conexión con el dispositivo ELM327.

```
#include <Obd.hpp>
```

Métodos públicos

- Obd (const char *deviceName)
  Constructor de la clase Obd.
- void discoverDeviceAddress (const char *deviceName, char *deviceAddress)
  Método que realiza el descubrimiento bluetooth del dispositivo ELM327.
- void connectBluetooth ()
  Método que realiza la conexión con el dispositivo bluetooth ELM327.
- void readFileData ()
  Método de lectura del fichero de PIDS en formato json.
- void send (Commands command)
  Método de envío de mensajes AT y OBD al dispositivo ELM327.
- void polling (Commands command)
  Método de recepción de mensajes enviados por el dispositivo ELM327.
- void initMessages ()
  Método de inicialización de parámetros de conexión con ELM327.
- void initDecoderFunctions ()
  Método de inicialización de funciones de decodificación de mensajes OBD.
- void disconnectBluetooth ()
  Método de desconexión bluetooth con el dispositivo ELM327.
- bool existPID (std::string command)
  Método de comprobación de existencia de un PID implemetado en el vehículo.
- void printPIDs ()
  Método de impresión de la lista de PIDS implementados en el vehículo.
- void printStatus ()
  Método de impresión de las pruebas realizadas en el vehículo.
- std::string getVIN ()
  Método que permite obtener el Número de Identificación del Vehículo (VIN).
- std::vector< std::string > getDTCs ()
  Método que permite obtener los DTC activos en el vehículo.
- bool isValid ()
  Método de validación del estado de la conexion.

Campos de datos

- std::map< std::string, Commands > map_commands

4.4.1. Descripción detallada

Clase que representa el acceso a la conexión con el dispositivo ELM327.

Clase principal que contiene los atributos y métodos necesarios para la conexión bluetooth con el dispositivo ELM327 y el posterior envío y recepción de mensajes OBD.

Definición en la línea 73 del archivo Obd.hpp.

4.4.2. Documentación del constructor y destructor

4.4.2.1. Obd()

```
Obd::Obd (
            const char * deviceName )  [inline]
```

Constructor de la clase Obd.

Parámetros

deviceName	Cadena de caracteres con el nombre del dispositivo bluetooth OBDII al que conectar.

Devuelve

Devuelve una instancia de la clase Obd.

Definición en la línea 83 del archivo Obd.hpp.

4.4.3. Documentación de las funciones miembro

4.4.3.1. connectBluetooth()

```
void Obd::connectBluetooth ( )  [inline]
```

Método que realiza la conexión con el dispositivo bluetooth ELM327.

Función que lleva a cabo la conexión con la interfaz bluetooth de ELM327. Crea un socket del tipo AF_BLUET↩ OOTH y configura los parámetros de conexión de éste con la dirección física obtenida tras el descubrimiento. Se crea una instancia epoll que permite monitorizar descriptores de ficheros y obtener notificaciones de ellos, en este caso para el socket de conexión bluetooth.

Definición en la línea 167 del archivo Obd.hpp.

4.4.3.2. disconnectBluetooth()

```
void Obd::disconnectBluetooth ( )  [inline]
```

Método de desconexión bluetooth con el dispositivo ELM327.

Cierra el socket e instancia epoll abiertas.

Definición en la línea 552 del archivo Obd.hpp.

4.4.3.3. discoverDeviceAddress()

```
void Obd::discoverDeviceAddress (
            const char * deviceName,
            char * deviceAddress )  [inline]
```

Método que realiza el descubrimiento bluetooth del dispositivo ELM327.

Parámetros

deviceName	Cadena de caracteres con el nombre del dispositivo bluetooth OBDII del que obtener la dirección física de conexión.
deviceAddress	Dirección física del dispositivo al que conectar tras el descubrimiento.

Función que realiza un escaneo de todos los dispositivos bluetooth disponibles y mediante un bucle filtra la dirección física del dispositivo bluetooth ELM327 pasado como primera parámetro.

Definición en la línea 112 del archivo Obd.hpp.

4.4.3.4. existPID()

```
bool Obd::existPID (
            std::string command )  [inline]
```

Método de comprobación de existencia de un PID implemetado en el vehículo.

Parámetros

command	String del comando a comprobar de su existencia entre los comandos disponibles.

Devuelve

Devuelve true si existe y false en caso contrario.

Definición en la línea 564 del archivo Obd.hpp.

4.4.3.5. getDTCs()

```
std::vector<std::string> Obd::getDTCs ( )  [inline]
```

Método que permite obtener los DTC activos en el vehículo.

Devuelve

Vector de strings con los DTC activos en el vehículo

Realiza la comprobación de existencia del número de DTC con el comando STATUS y si existen, obtiene su DTC con el comando GET_DTC.

Definición en la línea 628 del archivo Obd.hpp.

4.4.3.6. getVIN()

```
std::string Obd::getVIN ( )  [inline]
```

Método que permite obtener el Número de Identificación del Vehículo (VIN).

Devuelve

String del VIN de 17 dígitos del vehículo.

Definición en la línea 615 del archivo Obd.hpp.

4.4.3.7. initDecoderFunctions()

```
void Obd::initDecoderFunctions ( )  [inline]
```

Método de inicialización de funciones de decodificación de mensajes OBD.

Función que agrupa los decodificadores dependiendo del tipo de dato a obtener para poder utilizarlos en la función polling y obtener el dato solicitado.

Definición en la línea 511 del archivo Obd.hpp.

4.4.3.8. initMessages()

```
void Obd::initMessages ( )  [inline]
```

Método de inicialización de parámetros de conexión con ELM327.

Se realiza una secuencia de paso de mensajes que permiten obtener los datos en un formato normalizado. En primer lugar, se hace un RESET del dispositivo ELM327, se establecen los valores por defecto, se configura las respuestas sin eco, sin cabecera y sin espacio y se establece el protocolo automático. Por último, se realiza un escaneo general del estado del vehículo con distintas pruebas establecidas por el comando STATUS, se obtiene el VIN del vehículo y se obtiene el número de comandos disponibles tras un escaneo con los PIDS específicos para ello.

Definición en la línea 476 del archivo Obd.hpp.

4.4.3.9. isValid()

```
bool Obd::isValid ( )  [inline]
```

Método de validación del estado de la conexion.

Devuelve

Devuelve true si la conexión está establecida correctamente y false en caso contrario.

Definición en la línea 655 del archivo Obd.hpp.

4.4.3.10. polling()

```
void Obd::polling (
            Commands command )  [inline]
```

Método de recepción de mensajes enviados por el dispositivo ELM327.

Parámetros

command	Objeto del tipo Commands con la información del comando a recepcionar.

Función que se encarga de mantenerse a la espera del mensaje de respuesta del dispositivo ELM327 al mensaje anteriormente enviado por la función send. Mediante un bucle y la instancia epoll creada se recogen los eventos de mensajes recibidos, y se filtra su contenido para conocer la finalización del mensaje. Tras esto, se realiza una búsqueda de la información útil del mensaje y una decodificación dependiendo del tipo de dato a recibir. Por útimo, se almacena la respuesta en el propio objeto Commands para poder recuperarla posteriormente.

Definición en la línea 295 del archivo Obd.hpp.

4.4.3.11. printPIDs()

```
void Obd::printPIDs ( )  [inline]
```

Método de impresión de la lista de PIDS implementados en el vehículo.

Realiza una búsqueda iterativa que obtiene por consola los PIDS disponibles en el vehículo encontrados en la inicialización del dispositivo ELM327.

Definición en la línea 583 del archivo Obd.hpp.

4.4.3.12. printStatus()

```
void Obd::printStatus ( )  [inline]
```

Método de impresión de las pruebas realizadas en el vehículo.

Muestra por consola cada una de las pruebas realizadas en el vehículo y su resultado.

Definición en la línea 603 del archivo Obd.hpp.

4.4.3.13. readFileData()

```
void Obd::readFileData ( )  [inline]
```

Método de lectura del fichero de PIDS en formato json.

Utiliza la librería externa json.hpp para la lectura de los PIDS en formato JSON que permite obtener a la clase Obd los Commands a ejecutar.

Definición en la línea 229 del archivo Obd.hpp.

4.4.3.14. send()

```
void Obd::send (
            Commands command )  [inline]
```

Método de envío de mensajes AT y OBD al dispositivo ELM327.

Parámetros

command	Objeto del tipo Commands con la información del comando a enviar.

Función que se encarga de la creación de un hilo de ejecución que ejecute la función polling para la recepción del comando a enviar y del formateo de éste a través del socket creado al conectar con el dispositivo ELM327.

Definición en la línea 249 del archivo Obd.hpp.

4.4.4. Documentación de los campos

4.4.4.1. map_commands

```
std::map<std::string, Commands> Obd::map_commands
```

Map para asignación del nombre al comando correspondiente

Definición en la línea 75 del archivo Obd.hpp.

La documentación para esta clase fue generada a partir del siguiente fichero:

- Obd.hpp

4.5. Referencia de la Estructura OxigenoResponse

Estructura de datos para las respuesta de dos valores en PIDS relacionados con gases de escape.

```
#include <decoders.hpp>
```

Campos de datos

- float A
- float B

4.5.1. Descripción detallada

Estructura de datos para las respuesta de dos valores en PIDS relacionados con gases de escape.

Definición en la línea 25 del archivo decoders.hpp.

4.5.2. Documentación de los campos

4.5.2.1. A

```
float OxigenoResponse::A
```

Valor A en la formula de decodificación

Definición en la línea 26 del archivo decoders.hpp.

4.5.2.2. B

```
float OxigenoResponse::B
```

Valor B en la formula de decodificación

Definición en la línea 27 del archivo decoders.hpp.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- decoders.hpp

4.6. Referencia de la Estructura RelacionesResponse

Estructura de datos para las respuesta de cuatro valores en PIDS relacionados con gases de escape.

```
#include <decoders.hpp>
```

Campos de datos

- int A
- int B
- int C
- int D

4.6.1. Descripción detallada

Estructura de datos para las respuesta de cuatro valores en PIDS relacionados con gases de escape.

Definición en la línea 34 del archivo decoders.hpp.

4.6.2. Documentación de los campos

4.6.2.1. A

```
int RelacionesResponse::A
```

Valor A en la formula de decodificación

Definición en la línea 35 del archivo decoders.hpp.

4.6.2.2. B

```
int RelacionesResponse::B
```

Valor B en la formula de decodificación

Definición en la línea 36 del archivo decoders.hpp.

4.6.2.3. C

```
int RelacionesResponse::C
```

Valor C en la formula de decodificación

Definición en la línea 37 del archivo decoders.hpp.

4.6.2.4. D

```
int RelacionesResponse::D
```

Valor D en la formula de decodificación

Definición en la línea 38 del archivo decoders.hpp.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- decoders.hpp

Capítulo 5

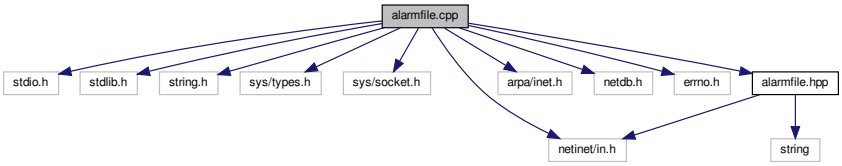
Documentación de archivos

5.1. Referencia del Archivo alarmfile.cpp

Archivo que contiene la definición de la clase para la configuración y envío de un mensaje a un servidor remoto.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include "alarmfile.hpp"
```

Dependencia gráfica adjunta para alarmfile.cpp:



5.1.1. Descripción detallada

Archivo que contiene la definición de la clase para la configuración y envío de un mensaje a un servidor remoto.

Autor

Juan Manuel Vozmediano Torres

Fecha

09/04/2019

5.2. Referencia del Archivo alarmfile.hpp

Archivo que contiene la declaración de la clase para la configuración y envío de un mensaje a un servidor remoto.

```
#include <string>
#include <netinet/in.h>
```

Dependencia gráfica adjunta para alarmfile.hpp:

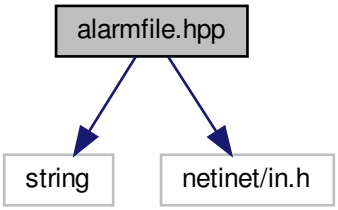
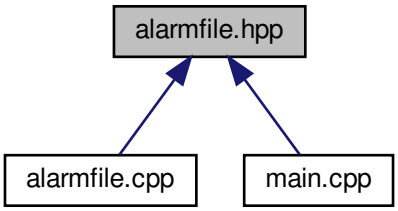


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- class AlarmFile

  Clase que representa la conexión con el servidor remoto para el envío de un mensaje (alarma).

5.2.1. Descripción detallada

Archivo que contiene la declaración de la clase para la configuración y envío de un mensaje a un servidor remoto.

Autor

Juan Manuel Vozmediano Torres

Fecha

09/04/2019

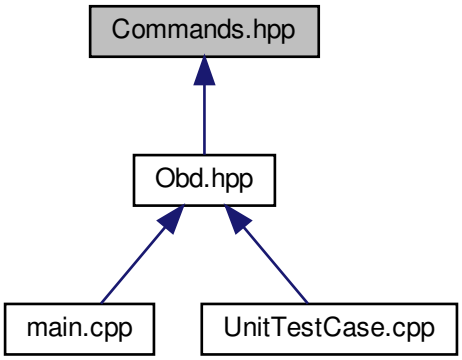
5.3. Referencia del Archivo Commands.hpp

Archivo que contiene la clase con la definición de la estructura de los comandos AT y OBD.

```
#include <iostream>
#include <any>
#include "external/json.hpp"
#include "decoders.hpp"
```

Dependencia gráfica adjunta para Commands.hpp:

Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- class Commands

  Clase que representa los comandos AT y PIDS que se necesitan en el intercambio de mensajes con el dispositivo ELM327.

typedefs

- using json = nlohmann::json

  Utilización de la librería externa nlohmann::json a través del tipo definido json.

5.3.1. Descripción detallada

Archivo que contiene la clase con la definición de la estructura de los comandos AT y OBD.

Autor

Sergio Román González

Fecha

05/09/2020

5.4. Referencia del Archivo debug.hpp

Archivo que contiene las funciones de debug en la salida estándar y de error del sistema.

```
#include <stdio.h>
```

Dependencia gráfica adjunta para debug.hpp:

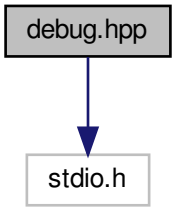
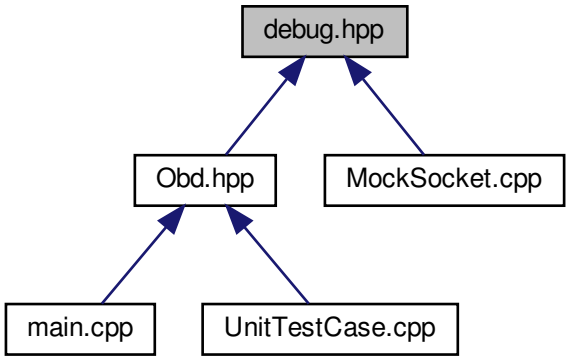


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



defines

- #define debugLog(format, args...) do{} while(0);

  Macro de función vacía para el debug del nivel de Log.

- #define debugError(format, args...) do{} while(0);

  Macro de función vacía para el debug del nivel de Error.

5.4.1. Descripción detallada

Archivo que contiene las funciones de debug en la salida estándar y de error del sistema.

Autor

Sergio Román González

Fecha

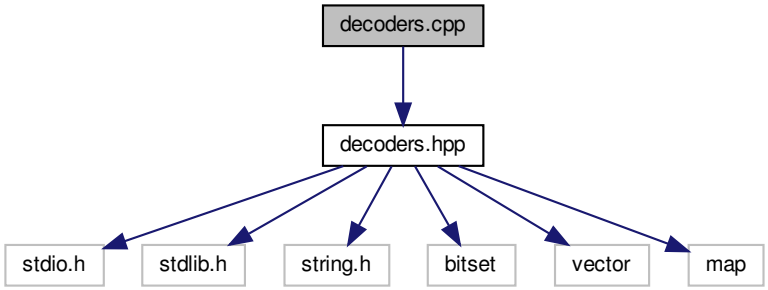
05/09/2020

5.5. Referencia del Archivo decoders.cpp

Archivo que contiene la definición de las funciones de decodificación de las respuestas del dispositivo ELM327.

```
#include "decoders.hpp"
```

Dependencia gráfica adjunta para decoders.cpp:



Funciones

- void noDecodeAT ()

  Función que no realiza decodificación para comandos AT.
- std::string decodeDescribeProtocol (char ∗response)

  Función de decodificación del protocolo de funcionamiento actual.
- std::string decodeVIN (char ∗response)

  Función de decodificación del Número de Identificación del Vehículo (VIN) para ISO15765-4 CAN.
- std::string convertDTCs (std::string dtc)

  Función de conversión del primer byte del DTC en su valor correspondiente.
- std::vector< std::string > decodeDTCs (char ∗response)

  Función de decodificación de los DTC activos en el vehículo.
- std::vector< int > decodePIDS (char ∗response)

  Función de decodificación de los PIDS disponibles en el vehículo.
- std::map< std::string, std::string > decodeStatus (char ∗response)

  Función de decodificación del PID STATUS.

- float decodeCargaPosicionEGR (char ∗response)

  Función de decodificación de la posición EGR.
- float decodeTempGeneral (char ∗response)

  Función de decodificación de la temperatura.
- float decodeAjusteCombustibleEGR (char ∗response)

  Función de decodificación del ajuste de combustible EGR.
- float decodePresionCombustible (char ∗response)

  Función de decodificación de la presión del combustible.
- float decodeHexToDec (char ∗response)

  Función de decodificación de hexadecimal a decimal.
- float decodeRPM (char ∗response)

  Función de decodificación de las RPM del motor.
- float decodeAvanceTiempo (char ∗response)

  Función de decodificación del avance del tiempo.
- float decodeVelocidadMAF (char ∗response)

  Función de decodificación de la tasa de flujo del aire (MAF).
- struct OxigenoResponse decodeSensorOxigeno (char ∗response)

  Función de decodificación de los sensores de oxígeno.
- float decodePresionCombColector (char ∗response)

  Función de decodificación de la presión del combustible del colector de vacío.
- float decodePresionMedidorCombustible (char ∗response)

  Función de decodificación de la presión del medidor del tren de combustible.
- struct OxigenoResponse decodeRelacionCombAire (char ∗response)

  Función de decodificación de los sensores de oxígeno y la relación de combustible.
- float decodePresionVapor (char ∗response)

  Función de decodificación de la presión de vapor del sistema evaporativo .
- struct OxigenoResponse decodeRelacionCombAireActual (char ∗response)

  Función de decodificación de los sensores de oxígeno y la relación de combustible actual.
- float decodeTempCatalizador (char ∗response)

  Función de decodificación de la temperatura del catalizador.
- float decodeVoltajeControl (char ∗response)

  Función de decodificación del voltaje del módulo de control.
- float decodeRelacionCombAireBasica (char ∗response)

  Función de decodificación de la relación equivaliente comandada de combustible - aire.
- struct RelacionesResponse decodeRelaciones (char ∗response)

  Función de decodificación del alor máximo de la relación de equivalencia de combustible - aire, voltaje del sensor de oxígenos, corriente del sensor de oxígenos y presión absoluta del colector de entrada.

5.5.1. Descripción detallada

Archivo que contiene la definición de las funciones de decodificación de las respuestas del dispositivo ELM327.

Autor

Sergio Román González

Fecha

05/09/2020

5.5.2. Documentación de las funciones

5.5.2.1. convertDTCs()

```
std::string convertDTCs (
            std::string dtc )
```

Función de conversión del primer byte del DTC en su valor correspondiente.

Parámetros

dtc	String con los bytes del DTC.

Devuelve

String de DTC con el primer byte convertido.

Definición en la línea 71 del archivo decoders.cpp.

5.5.2.2. decodeAjusteCombustibleEGR()

```
float decodeAjusteCombustibleEGR (
            char * response )
```

Función de decodificación del ajuste de combustible EGR.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 308 del archivo decoders.cpp.

5.5.2.3. decodeAvanceTiempo()

```
float decodeAvanceTiempo (
            char * response )
```

Función de decodificación del avance del tiempo.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 340 del archivo decoders.cpp.

5.5.2.4. decodeCargaPosicionEGR()

```
float decodeCargaPosicionEGR (
            char * response )
```

Función de decodificación de la posición EGR.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 294 del archivo decoders.cpp.

5.5.2.5. decodeDescribeProtocol()

```
std::string decodeDescribeProtocol (
            char * response )
```

Función de decodificación del protocolo de funcionamiento actual.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

String del protocolo de funcionamiento actual.

Definición en la línea 16 del archivo decoders.cpp.

5.5.2.6. decodeDTCs()

```
std::vector<std::string> decodeDTCs (
            char * response )
```

Función de decodificación de los DTC activos en el vehículo.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Vector de strings con los DTC activos en el vehículo.

Definición en la línea 110 del archivo decoders.cpp.

5.5.2.7. decodeHexToDec()

```
float decodeHexToDec (
            char * response )
```

Función de decodificación de hexadecimal a decimal.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 322 del archivo decoders.cpp.

5.5.2.8. decodePIDS()

```
std::vector<int> decodePIDS (
            char * response )
```

Función de decodificación de los PIDS disponibles en el vehículo.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Vector de enteros con los PIDS disponibles en el vehículo.

Definición en la línea 136 del archivo decoders.cpp.

5.5.2.9. decodePresionCombColector()

```
float decodePresionCombColector (
            char * response )
```

Función de decodificación de la presión del combustible del colector de vacío.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 408 del archivo decoders.cpp.

5.5.2.10. decodePresionCombustible()

```
float decodePresionCombustible (
            char * response )
```

Función de decodificación de la presión del combustible.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 315 del archivo decoders.cpp.

5.5.2.11. decodePresionMedidorCombustible()

```
float decodePresionMedidorCombustible (
            char * response )
```

Función de decodificación de la presión del medidor del tren de combustible.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 415 del archivo decoders.cpp.

5.5.2.12. decodePresionVapor()

```
float decodePresionVapor (
            char * response )
```

Función de decodificación de la presión de vapor del sistema evaporativo .

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 466 del archivo decoders.cpp.

5.5.2.13. decodeRelacionCombAire()

```
struct OxigenoResponse decodeRelacionCombAire (
            char * response )
```

Función de decodificación de los sensores de oxígeno y la relación de combustible.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 422 del archivo decoders.cpp.

5.5.2.14. decodeRelacionCombAireActual()

```
struct OxigenoResponse decodeRelacionCombAireActual (
            char * response )
```

Función de decodificación de los sensores de oxígeno y la relación de combustible actual.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 477 del archivo decoders.cpp.

5.5.2.15. decodeRelacionCombAireBasica()

```
float decodeRelacionCombAireBasica (
            char * response )
```

Función de decodificación de la relación equivaliente comandada de combustible - aire.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 520 del archivo decoders.cpp.

5.5.2.16. decodeRelaciones()

```
struct RelacionesResponse decodeRelaciones (
            char * response )
```

Función de decodificación del alor máximo de la relación de equivalencia de combustible - aire, voltaje del sensor de oxígenos, corriente del sensor de oxígenos y presión absoluta del colector de entrada.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 544 del archivo decoders.cpp.

5.5.2.17. decodeRPM()

```
float decodeRPM (
            char * response )
```

Función de decodificación de las RPM del motor.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 329 del archivo decoders.cpp.

5.5.2.18. decodeSensorOxigeno()

```
struct OxigenoResponse decodeSensorOxigeno (
            char * response )
```

Función de decodificación de los sensores de oxígeno.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 365 del archivo decoders.cpp.

5.5.2.19. decodeStatus()

```
std::map<std::string, std::string> decodeStatus (
            char * response )
```

Función de decodificación del PID STATUS.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Mapa string/string con el estado de los monitores de diagnóstico.

Definición en la línea 152 del archivo decoders.cpp.

5.5.2.20. decodeTempCatalizador()

```
float decodeTempCatalizador (
            char * response )
```

Función de decodificación de la temperatura del catalizador.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 500 del archivo decoders.cpp.

5.5.2.21. decodeTempGeneral()

```
float decodeTempGeneral (
            char * response )
```

Función de decodificación de la temperatura.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 301 del archivo decoders.cpp.

5.5.2.22. decodeVelocidadMAF()

```
float decodeVelocidadMAF (
            char * response )
```

Función de decodificación de la tasa de flujo del aire (MAF).

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 351 del archivo decoders.cpp.

5.5.2.23. decodeVIN()

```
std::string decodeVIN (
            char * response )
```

Función de decodificación del Número de Identificación del Vehículo (VIN) para ISO15765-4 CAN.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

String con el Número de Identificación del Vehículo (VIN).

Definición en la línea 21 del archivo decoders.cpp.

5.5.2.24. decodeVoltajeControl()

```
float decodeVoltajeControl (
            char * response )
```

Función de decodificación del voltaje del módulo de control.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 509 del archivo decoders.cpp.

5.6. Referencia del Archivo decoders.hpp

Archivo que contiene la declaración de las funciones de decodificación de las respuestas del dispositivo ELM327.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <bitset>
#include <vector>
#include <map>
```

Dependencia gráfica adjunta para decoders.hpp:

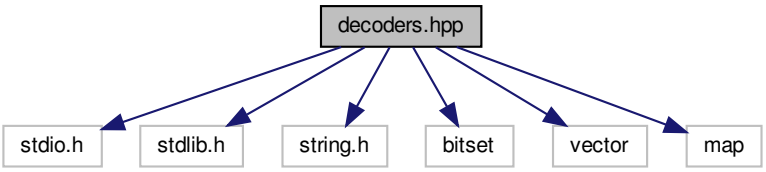
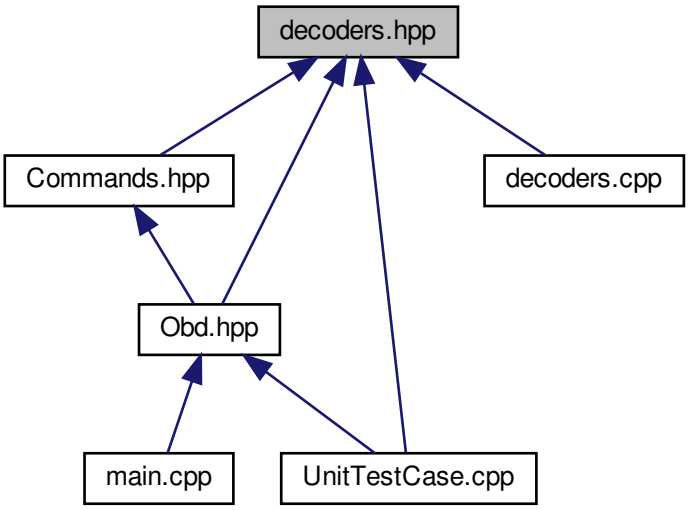


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- struct OxigenoResponse

  Estructura de datos para las respuesta de dos valores en PIDS relacionados con gases de escape.
- struct RelacionesResponse

  Estructura de datos para las respuesta de cuatro valores en PIDS relacionados con gases de escape.

defines

- #define PID_BITS 32
- #define STATUS_BITS 8

Funciones

- void noDecodeAT ()

  Función que no realiza decodificación para comandos AT.
- std::map< std::string, std::string > decodeStatus (char *response)

  Función de decodificación del PID STATUS.
- std::vector< int > decodePIDS (char *response)

  Función de decodificación de los PIDS disponibles en el vehículo.
- std::vector< std::string > decodeDTCs (char *response)

  Función de decodificación de los DTC activos en el vehículo.
- std::string convertDTCs (std::string dtc)

  Función de conversión del primer byte del DTC en su valor correspondiente.
- std::string decodeVIN (char *response)

  Función de decodificación del Número de Identificación del Vehículo (VIN) para ISO15765-4 CAN.
- std::string decodeDescribeProtocol (char *response)

  Función de decodificación del protocolo de funcionamiento actual.
- float decodeCargaPosicionEGR (char *response)

  Función de decodificación de la posición EGR.
- float decodeTempGeneral (char *response)

  Función de decodificación de la temperatura.
- float decodeAjusteCombustibleEGR (char *response)

  Función de decodificación del ajuste de combustible EGR.
- float decodePresionCombustible (char *response)

  Función de decodificación de la presión del combustible.
- float decodeHexToDec (char *response)

  Función de decodificación de hexadecimal a decimal.
- float decodeRPM (char *response)

  Función de decodificación de las RPM del motor.
- float decodeAvanceTiempo (char *response)

  Función de decodificación del avance del tiempo.
- float decodeVelocidadMAF (char *response)

  Función de decodificación de la tasa de flujo del aire (MAF).
- float decodePresionCombColector (char *response)

  Función de decodificación de la presión del combustible del colector de vacío.
- float decodePresionMedidorCombustible (char *response)

  Función de decodificación de la presión del medidor del tren de combustible.
- float decodePresionVapor (char *response)

  Función de decodificación de la presión de vapor del sistema evaporativo .

- float decodeTempCatalizador (char ∗response)

  Función de decodificación de la temperatura del catalizador.
- float decodeVoltajeControl (char ∗response)

  Función de decodificación del voltaje del módulo de control.
- float decodeRelacionCombAireBasica (char ∗response)

  Función de decodificación de la relación equivaliente comandada de combustible - aire.
- struct OxigenoResponse decodeSensorOxigeno (char ∗response)

  Función de decodificación de los sensores de oxígeno.
- struct OxigenoResponse decodeRelacionCombAire (char ∗response)

  Función de decodificación de los sensores de oxígeno y la relación de combustible.
- struct OxigenoResponse decodeRelacionCombAireActual (char ∗response)

  Función de decodificación de los sensores de oxígeno y la relación de combustible actual.
- struct RelacionesResponse decodeRelaciones (char ∗response)

  Función de decodificación del alor máximo de la relación de equivalencia de combustible - aire, voltaje del sensor de oxígenos, corriente del sensor de oxígenos y presión absoluta del colector de entrada.

5.6.1. Descripción detallada

Archivo que contiene la declaración de las funciones de decodificación de las respuestas del dispositivo ELM327.

Autor

Sergio Román González

Fecha

05/09/2020

5.6.2. Documentación de los 'defines'

5.6.2.1. PID_BITS

```
#define PID_BITS 32
```

Macro con el número de bits de respuesta para la solicitud de PIDs disponibles

Definición en la línea 18 del archivo decoders.hpp.

5.6.2.2. STATUS_BITS

```
#define STATUS_BITS 8
```

Macro con el número de bits de respuesta para las pruebas del PID STATUS

Definición en la línea 19 del archivo decoders.hpp.

5.6.3. Documentación de las funciones

5.6.3.1. convertDTCs()

```
std::string convertDTCs (
            std::string dtc )
```

Función de conversión del primer byte del DTC en su valor correspondiente.

Parámetros

dtc	String con los bytes del DTC.

Devuelve

String de DTC con el primer byte convertido.

Definición en la línea 71 del archivo decoders.cpp.

5.6.3.2. decodeAjusteCombustibleEGR()

```
float decodeAjusteCombustibleEGR (
            char * response )
```

Función de decodificación del ajuste de combustible EGR.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 308 del archivo decoders.cpp.

5.6.3.3. decodeAvanceTiempo()

```
float decodeAvanceTiempo (
            char * response )
```

Función de decodificación del avance del tiempo.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 340 del archivo decoders.cpp.

5.6.3.4. decodeCargaPosicionEGR()

```
float decodeCargaPosicionEGR (
            char * response )
```

Función de decodificación de la posición EGR.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 294 del archivo decoders.cpp.

5.6.3.5. decodeDescribeProtocol()

```
std::string decodeDescribeProtocol (
            char * response )
```

Función de decodificación del protocolo de funcionamiento actual.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

String del protocolo de funcionamiento actual.

Definición en la línea 16 del archivo decoders.cpp.

5.6.3.6. decodeDTCs()

```
std::vector<std::string> decodeDTCs (
            char * response )
```

Función de decodificación de los DTC activos en el vehículo.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Vector de strings con los DTC activos en el vehículo.

Definición en la línea 110 del archivo decoders.cpp.

5.6.3.7. decodeHexToDec()

```
float decodeHexToDec (
            char * response )
```

Función de decodificación de hexadecimal a decimal.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 322 del archivo decoders.cpp.

5.6.3.8. decodePIDS()

```
std::vector<int> decodePIDS (
            char * response )
```

Función de decodificación de los PIDS disponibles en el vehículo.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Vector de enteros con los PIDS disponibles en el vehículo.

Definición en la línea 136 del archivo decoders.cpp.

5.6.3.9. decodePresionCombColector()

```
float decodePresionCombColector (
            char * response )
```

Función de decodificación de la presión del combustible del colector de vacío.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 408 del archivo decoders.cpp.

5.6.3.10. decodePresionCombustible()

```
float decodePresionCombustible (
            char * response )
```

Función de decodificación de la presión del combustible.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 315 del archivo decoders.cpp.

5.6.3.11. decodePresionMedidorCombustible()

```
float decodePresionMedidorCombustible (
            char * response )
```

Función de decodificación de la presión del medidor del tren de combustible.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 415 del archivo decoders.cpp.

5.6.3.12. decodePresionVapor()

```
float decodePresionVapor (
            char * response )
```

Función de decodificación de la presión de vapor del sistema evaporativo .

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 466 del archivo decoders.cpp.

5.6.3.13. decodeRelacionCombAire()

```
struct OxigenoResponse decodeRelacionCombAire (
            char * response )
```

Función de decodificación de los sensores de oxígeno y la relación de combustible.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 422 del archivo decoders.cpp.

5.6.3.14. decodeRelacionCombAireActual()

```
struct OxigenoResponse decodeRelacionCombAireActual (
            char * response )
```

Función de decodificación de los sensores de oxígeno y la relación de combustible actual.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 477 del archivo decoders.cpp.

5.6.3.15. decodeRelacionCombAireBasica()

```
float decodeRelacionCombAireBasica (
            char * response )
```

Función de decodificación de la relación equivaliente comandada de combustible - aire.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 520 del archivo decoders.cpp.

5.6.3.16. decodeRelaciones()

```
struct RelacionesResponse decodeRelaciones (
            char * response )
```

Función de decodificación del alor máximo de la relación de equivalencia de combustible - aire, voltaje del sensor de oxígenos, corriente del sensor de oxígenos y presión absoluta del colector de entrada.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 544 del archivo decoders.cpp.

5.6.3.17. decodeRPM()

```
float decodeRPM (
            char * response )
```

Función de decodificación de las RPM del motor.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 329 del archivo decoders.cpp.

5.6.3.18. decodeSensorOxigeno()

```
struct OxigenoResponse decodeSensorOxigeno (
            char * response )
```

Función de decodificación de los sensores de oxígeno.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Estructura OxigenoResponse con el valor A y B correspondiente al comando solicitado.

Definición en la línea 365 del archivo decoders.cpp.

5.6.3.19. decodeStatus()

```
std::map<std::string, std::string> decodeStatus (
            char * response )
```

Función de decodificación del PID STATUS.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Mapa string/string con el estado de los monitores de diagnóstico.

Definición en la línea 152 del archivo decoders.cpp.

5.6.3.20. decodeTempCatalizador()

```
float decodeTempCatalizador (
            char * response )
```

Función de decodificación de la temperatura del catalizador.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 500 del archivo decoders.cpp.

5.6.3.21. decodeTempGeneral()

```
float decodeTempGeneral (
            char * response )
```

Función de decodificación de la temperatura.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 301 del archivo decoders.cpp.

5.6.3.22. decodeVelocidadMAF()

```
float decodeVelocidadMAF (
            char * response )
```

Función de decodificación de la tasa de flujo del aire (MAF).

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

Definición en la línea 351 del archivo decoders.cpp.

5.6.3.23. decodeVIN()

```
std::string decodeVIN (
            char * response )
```

Función de decodificación del Número de Identificación del Vehículo (VIN) para ISO15765-4 CAN.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

String con el Número de Identificación del Vehículo (VIN).

Definición en la línea 21 del archivo decoders.cpp.

5.6.3.24. decodeVoltajeControl()

```
float decodeVoltajeControl (
            char * response )
```

Función de decodificación del voltaje del módulo de control.

Parámetros

response	Cadena de caracteres con los bytes útiles de la respuesta del dispositivo ELM327.

Devuelve

Float del valor de la respuesta decodificada.

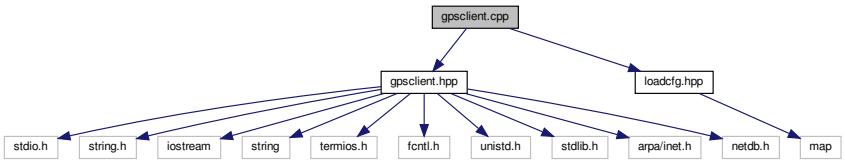
Definición en la línea 509 del archivo decoders.cpp.

5.7. Referencia del Archivo gpsclient.cpp

Archivo que contiene la definición de la clase para la conexión con el servicio gpsd de obtención de coordenadas GPS.

```
#include "gpsclient.hpp"
#include "loadcfg.hpp"
```

Dependencia gráfica adjunta para gpsclient.cpp:



defines

- #define NOGPSDATA "000000,010170,0000.000,N,00000.000,E"

5.7.1. Descripción detallada

Archivo que contiene la definición de la clase para la conexión con el servicio gpsd de obtención de coordenadas GPS.

Autor

Juan Manuel Vozmediano Torres

Fecha

09/08/2019

5.7.2. Documentación de los 'defines'

5.7.2.1. NOGPSDATA

```
#define NOGPSDATA "000000,010170,0000.000,N,00000.000,E"
```

Macro con la string en el caso de que no haya datos de GPS disponibles.

Definición en la línea 11 del archivo gpsclient.cpp.

5.8. Referencia del Archivo gpsclient.hpp

Archivo que contiene la declaración de la clase para la conexión con el servicio gpsd de obtención de coordenadas GPS.

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <string>
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <netdb.h>
```

Dependencia gráfica adjunta para gpsclient.hpp:

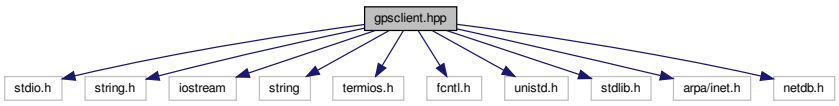
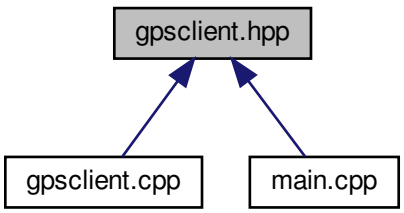


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- class GpsClient

  Clase que representa la conexión con el servicio gpsd para obtener las coordenadas GPS.

5.8.1. Descripción detallada

Archivo que contiene la declaración de la clase para la conexión con el servicio gpsd de obtención de coordenadas GPS.

Autor

Juan Manuel Vozmediano Torres

Fecha

09/08/2019

5.9. Referencia del Archivo loadcfg.cpp

Archivo que contiene la definición de las funciones para la lectura de un fichero de configuración del tipo clave=valor.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <map>
#include <string>
#include <ifaddrs.h>
#include <ctype.h>
#include <unistd.h>
#include <stdexcept>
#include <sstream>
#include <netpacket/packet.h>
#include "loadcfg.hpp"
```

Dependencia gráfica adjunta para loadcfg.cpp:

Funciones

- void shit (const char ∗mens)

  Función para indicar error en el código y terminar la ejecución.
- void loadCfg (const char ∗filename, cfgType ∗pcfg)

  Función cargar la configuración y almacenarla para su utilización.
- std::string getmac (const char ∗name)

  Función que obtiene la MAC de una interfaz de red indicada.

5.9.1. Descripción detallada

Archivo que contiene la definición de las funciones para la lectura de un fichero de configuración del tipo clave=valor.

Autor

Juan Manuel Vozmediano Torres

Fecha

09/04/2019

5.9.2. Documentación de las funciones

5.9.2.1. getmac()

```
std::string getmac (
            const char * name )
```

Función que obtiene la MAC de una interfaz de red indicada.

Parámetros

name	Cadena de caracteres indicando el nombre de la interfaz de red de la que obtener su MAC.

Devuelve

String de la MAC de la interfaz de red indicada.

Definición en la línea 60 del archivo loadcfg.cpp.

5.9.2.2. loadCfg()

```
void loadCfg (
            const char * filename,
            cfgType * pcfg )
```

Función cargar la configuración y almacenarla para su utilización.

Parámetros

filename	Cadena de caracteres del archivo de configuración a leer.
pcfg	Variable de tipo puntero a cfgType para referenciar la variable donde se almacenará la configuración.

Definición en la línea 39 del archivo loadcfg.cpp.

5.9.2.3. shit()

```
void shit (
            const char * mens )
```

Función para indicar error en el código y terminar la ejecución.

Parámetros

mens	Cadena de caracteres para mostrar en el error producido.

Definición en la línea 32 del archivo loadcfg.cpp.

5.10. Referencia del Archivo loadcfg.hpp

Archivo que contiene la declaración de las funciones para la lectura de un fichero de configuración del tipo clave=valor.

```
#include <map>
```

Dependencia gráfica adjunta para loadcfg.hpp:

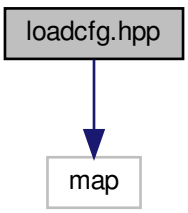
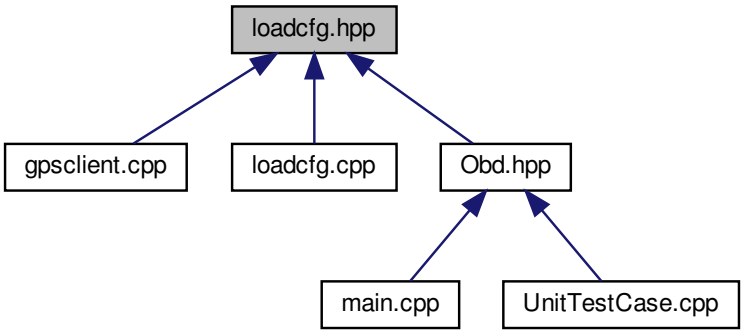


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



typedefs

- typedef std::map< std::string, std::string > cfgType

  Definición del tipo cfgType para referenciar los parámetros de configuración y sus valores.

Funciones

- void shit (const char *mens)

  Función para indicar error en el código y terminar la ejecución.
- void loadCfg (const char *filename, cfgType *pcfg)

  Función cargar la configuración y almacenarla para su utilización.
- std::string getmac (const char *name)

  Función que obtiene la MAC de una interfaz de red indicada.

5.10.1. Descripción detallada

Archivo que contiene la declaración de las funciones para la lectura de un fichero de configuración del tipo clave=valor.

Autor

Juan Manuel Vozmediano Torres

Fecha

09/04/2019

5.10.2. Documentación de las funciones

5.10.2.1. getmac()

```
std::string getmac (
            const char * name )
```

Función que obtiene la MAC de una interfaz de red indicada.

Parámetros

name	Cadena de caracteres indicando el nombre de la interfaz de red de la que obtener su MAC.

Devuelve

String de la MAC de la interfaz de red indicada.

Definición en la línea 60 del archivo loadcfg.cpp.

5.10.2.2. loadCfg()

```
void loadCfg (
             const char * filename,
             cfgType * pcfg )
```

Función cargar la configuración y almacenarla para su utilización.

Parámetros

filename	Cadena de caracteres del archivo de configuración a leer.
pcfg	Variable de tipo puntero a cfgType para referenciar la variable donde se almacenará la configuración.

Definición en la línea 39 del archivo loadcfg.cpp.

5.10.2.3. shit()

```
void shit (
             const char * mens )
```

Función para indicar error en el código y terminar la ejecución.

Parámetros

mens	Cadena de caracteres para mostrar en el error producido.

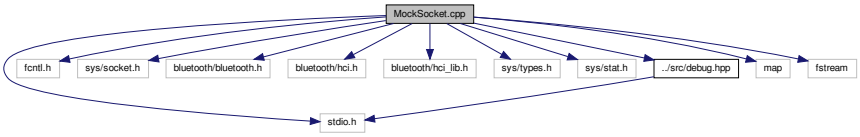
Definición en la línea 32 del archivo loadcfg.cpp.

5.11. Referencia del Archivo MockSocket.cpp

Archivo que contiene las funciones mock bluetooth para poder realizar las pruebas de integración.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "../src/debug.hpp"
#include <map>
#include <fstream>
```

Dependencia gráfica adjunta para MockSocket.cpp:



Funciones

- std::string findDevPTS ()

  Función de detección del dispositivo para la simulación con OBDSIM.
- int **mock_socket** (int domain, int type, int protocol)
- int **hci_get_route** (bdaddr_t ∗bdaddr)
- int **hci_open_dev** (int dev_id)
- int **hci_inquiry** (int dev_id, int len, int max_rsp, const uint8_t ∗lap, inquiry_info ∗∗ii, long flags)
- int **hci_read_remote_name** (int sock, const bdaddr_t ∗ba, int len, char ∗name, int timeout)
- int **connect** (int sockfd, const struct sockaddr ∗addr, socklen_t addrlen)
- void **writeSocket** ()

5.11.1. Descripción detallada

Archivo que contiene las funciones mock bluetooth para poder realizar las pruebas de integración.

Autor

Sergio Román González

Fecha

05/09/2020

5.11.2. Documentación de las funciones

5.11.2.1. findDevPTS()

```
std::string findDevPTS ( )
```

Función de detección del dispositivo para la simulación con OBDSIM.

Devuelve

String con la ruta del dispositivo al que conectarse para la simulación OBDSIM.

Definición en la línea 27 del archivo MockSocket.cpp.

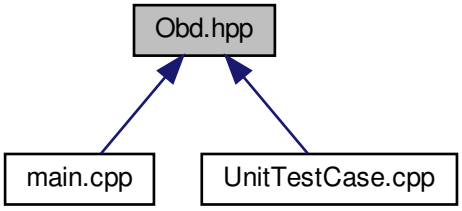
5.12. Referencia del Archivo Obd.hpp

Archivo que contiene la clase con la implementación de la conexión y envío de mensajes OBD con el dispositivo ELM327.

```
#include <iostream>
#include <fstream>
#include <thread>
#include <bitset>
#include <vector>
#include <sstream>
#include <algorithm>
#include <utility>
#include <map>
#include <typeinfo>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/epoll.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>
#include <unistd.h>
#include <ctime>
#include "Commands.hpp"
#include "decoders.hpp"
#include "loadcfg.hpp"
#include "debug.hpp"
```

Dependencia gráfica adjunta para Obd.hpp:

Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- class Obd

  Clase que representa el acceso a la conexión con el dispositivo ELM327.

defines

- #define MAX_EP_EVTS 20

typedefs

- using json = nlohmann::json

  Utilización de la librería externa nlohmann::json a través del tipo definido json.
- typedef std::pair< std::string, Commands > tupla

  Definición del tipo pair para la asignación de los objetos Commands a la clase Obd.

5.12.1. Descripción detallada

Archivo que contiene la clase con la implementación de la conexión y envío de mensajes OBD con el dispositivo ELM327.

Autor

Sergio Román González

Fecha

05/09/2020

5.12.2. Documentación de los 'defines'

5.12.2.1. MAX_EP_EVTS

```
#define MAX_EP_EVTS 20
```

Macro con el número máximo de eventos en la recepción de la instancia epoll

Definición en la línea 52 del archivo Obd.hpp.

5.13. Referencia del Archivo UnitTestCase.cpp

Archivo que contiene el conjunto de pruebas unitarias y de integración del sistema.

```
#include "../src/external/catch.hpp"
#include "../src/decoders.hpp"
#include "../src/Obd.hpp"
```

Dependencia gráfica adjunta para UnitTestCase.cpp:

defines

- #define CATCH_CONFIG_MAIN
- #define BUFSIZE 30
- #define WAIT_OBDSIM 1

Funciones

- void getMinicomCMD (char ∗cmd)

  Función que obtiene el comando Minicom para la primera conexión con el simulador OBDSIM.
- void initOBDSIM (std::string tipoSimulador, int tiempoEspera)

  Función de inicialización del simulador OBDSIM.
- void closeOBDSIM ()

  Función de finalización del simulador OBDSIM.
- TEST_CASE ("Test OBD class DTC", "[OBD]")

  Prueba de integración para el funcionamiento de la obtención de DTC y PIDS disponibles.
- TEST_CASE ("Test OBD class data SPEED", "[OBD]")

  Prueba de integración para el funcionamiento de solicitud de un dato continúo (velocidad).
- TEST_CASE ("Test Revoluciones Por Minuto", "[decoders]")

  Prueba unitaria del decodificador RPM.
- TEST_CASE ("Test Posición EGR", "[decoders]")

  Prueba unitaria del decodificador de posición EGR.
- TEST_CASE ("Test Temperatura General", "[decoders]")

  Prueba unitaria del decodificador de la temperatura general.
- TEST_CASE ("Test Ajuste Combustible EGR", "[decoders]")

  Prueba unitaria del decodificador de la temperatura general.
- TEST_CASE ("Test Presión Combustible", "[decoders]")

Prueba unitaria del decodificador de la presión del combustible.

- TEST_CASE ("Test Hexadecimal a Decimal", "[decoders]")

  Prueba unitaria del decodificador hexadecimal a decimal.
- TEST_CASE ("Test Avance Tiempo", "[decoders]")

  Prueba unitaria del decodificador del avance del tiempo.
- TEST_CASE ("Test Velocidad Flujo Aire MAF", "[decoders]")

  Prueba unitaria del decodificador de la tasa de flujo del aire (MAF).
- TEST_CASE ("Test Presión del Combustible, relativa al colector de vacío", "[decoders]")

  Prueba unitaria del decodificador de presión de combustible relativa al colector de vacío.
- TEST_CASE ("Test Presión del Combustible (Diesel o inyección directa de gasolina)", "[decoders]")

  Prueba unitaria del decodificador de presión de combusible (Diese o inyección directa de gasolina).
- TEST_CASE ("Test Presión de Vapor del Sistema Evaporativo", "[decoders]")

  Prueba unitaria del decodificador de Vapor del Sistema Evaporativo.
- TEST_CASE ("Test Temperatura del Catalizador", "[decoders]")

  Prueba unitaria del decodificador de Temperatura del Catalizador.
- TEST_CASE ("Test Voltaje del Módulo de Control", "[decoders]")

  Prueba unitaria del decodificador de Voltaje del Módulo de Control.
- TEST_CASE ("Test Relación Equivaliente Comandada de Combustible", "[decoders]")

  Prueba unitaria del decodificador de Relación Equivaliente Comandada de Combustible.
- TEST_CASE ("Comprobación Diagnostic Trouble Codes", "[DTC]")

  Prueba unitaria del conversor del primer byte DTC.
- TEST_CASE ("Test VIN (Vehicle Identification Number) ISO15765-4 CAN", "[decoders]")

  Prueba unitaria del Número de Identificación del vehículo.
- TEST_CASE ("Test Describir el Protocolo Actual", "[decoders]")

  Prueba unitaria del decodificador de descriptor del protocolo actual.
- SCENARIO ("Test de Sensores de Oxígeno", "[decoders]")

  Escenario de pruebas con distintos test de los sensores de oxígeno.
- SCENARIO ("Test de Relación Equivalente Combustible-Aire", "[decoders]")

  Escenario de pruebas con distintos test de la relación equivalente combustible-aire.
- SCENARIO ("Test de Relación Equivalente Combustible-Aire Actual", "[decoders]")

  Escenario de pruebas con distintos test de la relación equivalente combustible-aire actual.
- SCENARIO ("Test de Valores máximos relación de combustible-aire, voltaje, corriente y presión absoluta", "[decoders]")

  Escenario de pruebas con distintos test de valores máximo relación de combustible-aire, voltaje, corriente y presión absoluta.
- SCENARIO ("Test de decodificación de Data Trouble Codes (DTC)", "[decoders]")

  Escenario de pruebas con distintos test de DTC activos.
- SCENARIO ("Test de decodificación PIDs disponibles", "[decoders]")

  Escenario de pruebas con distintos test los PIDS implementados en el vehículo.
- SCENARIO ("Test de decodificación del estado del coche", "[decoders]")

  Escenario de pruebas con distintos valores de los monitores de diagnóstico tras las pruebas del vehículo.

5.13.1. Descripción detallada

Archivo que contiene el conjunto de pruebas unitarias y de integración del sistema.

Autor

Sergio Román González

Fecha

05/09/2020

5.13.2. Documentación de los 'defines'

5.13.2.1. BUFSIZE

```
#define BUFSIZE 30
```

Macro del tamaño del buffer de la cadena de caracteres para el comando Minicom a ejecutar

Definición en la línea 15 del archivo UnitTestCase.cpp.

5.13.2.2. CATCH_CONFIG_MAIN

```
#define CATCH_CONFIG_MAIN
```

Macro que permite a la librería catch proporcionar un main() para la ejecución del conjunto de pruebas

Definición en la línea 9 del archivo UnitTestCase.cpp.

5.13.2.3. WAIT_OBDSIM

```
#define WAIT_OBDSIM 1
```

Macro con el tiempo de espera en segundos del simulador OBDSIM para introducir valores

Definición en la línea 16 del archivo UnitTestCase.cpp.

5.13.3. Documentación de las funciones

5.13.3.1. getMinicomCMD()

```
void getMinicomCMD (
            char * cmd )
```

Función que obtiene el comando Minicom para la primera conexión con el simulador OBDSIM.

Parámetros

cmd	Puntero a cadena de caracteres para almacenar el comando de minicom a ejecutar.

Definición en la línea 27 del archivo UnitTestCase.cpp.

5.13.3.2. initOBDSIM()

```
void initOBDSIM (
             std::string tipoSimulador,
             int tiempoEspera )
```

Función de inicialización del simulador OBDSIM.

Parámetros

tipoSimulador	String para indicar el tipo de simulador a iniciar.
tiempoEspera	Entero con el número de segundos de espera del simulador.

Inicializa el simulador con entorno gráfico permitiendo la introducción de DTC y otros valores para las pruebas.

Definición en la línea 47 del archivo UnitTestCase.cpp.

Índice alfabético

A
OxigenoResponse, 24
RelacionesResponse, 25
AlarmFile, 7
AlarmFile, 7
sendAlarm, 8
alarmfile.cpp, 27
alarmfile.hpp, 28

B
OxigenoResponse, 25
RelacionesResponse, 26
BUFSIZE
UnitTestCase.cpp, 66

C
RelacionesResponse, 26
CATCH_CONFIG_MAIN
UnitTestCase.cpp, 66
Commands, 8
Commands, 10
getBytesResponse, 10
getCMDResponse, 11
getCMD, 10
getDecoder, 11
getDescription, 11
getJson, 12
getMAX, 12
getMIN, 12
getName, 13
getResValue, 13
getTypeData, 13
getUnits, 13
setBytesResponse, 14
setCMD, 14
setDecoder, 14
setDescription, 15
setMAX, 15
setMIN, 15
setName, 16
setResValue, 16
setTypeData, 16
setUnits, 17
Commands.hpp, 29
connectBluetooth
Obd, 20
convertDTCs
decoders.cpp, 33
decoders.hpp, 45

D
RelacionesResponse, 26
debug.hpp, 30
decodeAjusteCombustibleEGR
decoders.cpp, 33
decoders.hpp, 45
decodeAvanceTiempo
decoders.cpp, 33
decoders.hpp, 45
decodeCargaPosicionEGR
decoders.cpp, 34
decoders.hpp, 46
decodeDTCs
decoders.cpp, 34
decoders.hpp, 46
decodeDescribeProtocol
decoders.cpp, 34
decoders.hpp, 46
decodeHexToDec
decoders.cpp, 35
decoders.hpp, 47
decodePIDS
decoders.cpp, 35
decoders.hpp, 47
decodePresionCombColector
decoders.cpp, 36
decoders.hpp, 48
decodePresionCombustible
decoders.cpp, 36
decoders.hpp, 48
decodePresionMedidorCombustible
decoders.cpp, 36
decoders.hpp, 48
decodePresionVapor
decoders.cpp, 37
decoders.hpp, 49
decodeRPM
decoders.cpp, 39
decoders.hpp, 51
decodeRelacionCombAire
decoders.cpp, 37
decoders.hpp, 49
decodeRelacionCombAireActual
decoders.cpp, 37
decoders.hpp, 49
decodeRelacionCombAireBasica
decoders.cpp, 38
decoders.hpp, 50
decodeRelaciones

decoders.cpp, 38
decoders.hpp, 50
decodeSensorOxigeno
decoders.cpp, 39
decoders.hpp, 51
decodeStatus
decoders.cpp, 39
decoders.hpp, 51
decodeTempCatalizador
decoders.cpp, 40
decoders.hpp, 52
decodeTempGeneral
decoders.cpp, 40
decoders.hpp, 52
decodeVIN
decoders.cpp, 41
decoders.hpp, 53
decodeVelocidadMAF
decoders.cpp, 40
decoders.hpp, 52
decodeVoltajeControl
decoders.cpp, 41
decoders.hpp, 53
decoders.cpp, 31
convertDTCs, 33
decodeAjusteCombustibleEGR, 33
decodeAvanceTiempo, 33
decodeCargaPosicionEGR, 34
decodeDTCs, 34
decodeDescribeProtocol, 34
decodeHexToDec, 35
decodePIDS, 35
decodePresionCombColector, 36
decodePresionCombustible, 36
decodePresionMedidorCombustible, 36
decodePresionVapor, 37
decodeRPM, 39
decodeRelacionCombAire, 37
decodeRelacionCombAireActual, 37
decodeRelacionCombAireBasica, 38
decodeRelaciones, 38
decodeSensorOxigeno, 39
decodeStatus, 39
decodeTempCatalizador, 40
decodeTempGeneral, 40
decodeVIN, 41
decodeVelocidadMAF, 40
decodeVoltajeControl, 41
decoders.hpp, 42
convertDTCs, 45
decodeAjusteCombustibleEGR, 45
decodeAvanceTiempo, 45
decodeCargaPosicionEGR, 46
decodeDTCs, 46
decodeDescribeProtocol, 46
decodeHexToDec, 47
decodePIDS, 47
decodePresionCombColector, 48
decodePresionCombustible, 48
decodePresionMedidorCombustible, 48
decodePresionVapor, 49
decodeRPM, 51
decodeRelacionCombAire, 49
decodeRelacionCombAireActual, 49
decodeRelacionCombAireBasica, 50
decodeRelaciones, 50
decodeSensorOxigeno, 51
decodeStatus, 51
decodeTempCatalizador, 52
decodeTempGeneral, 52
decodeVIN, 53
decodeVelocidadMAF, 52
decodeVoltajeControl, 53
PID_BITS, 44
STATUS_BITS, 44
disconnectBluetooth
Obd, 20
discoverDeviceAddress
Obd, 20

existPID
Obd, 21

findDevPTS
MockSocket.cpp, 61

getBytesResponse
Commands, 10
getCMDResponse
Commands, 11
getCMD
Commands, 10
getDTCs
Obd, 21
getDecoder
Commands, 11
getDescription
Commands, 11
getGPS
GpsClient, 18
getJson
Commands, 12
getMAX
Commands, 12
getMIN
Commands, 12
getMinicomCMD
UnitTestCase.cpp, 66
getName
Commands, 13
getResValue
Commands, 13
getTypeData
Commands, 13
getUnits
Commands, 13
getVIN

Obd, 21
getmac
loadcfg.cpp, 57
loadcfg.hpp, 59
GpsClient, 17
getGPS, 18
GpsClient, 17
gpsclient.cpp, 54
NOGPSDATA, 54
gpsclient.hpp, 55

initDecoderFunctions
Obd, 22
initMessages
Obd, 22
initOBDSIM
UnitTestCase.cpp, 67
isValid
Obd, 22

loadCfg
loadcfg.cpp, 57
loadcfg.hpp, 60
loadcfg.cpp, 56
getmac, 57
loadCfg, 57
shit, 58
loadcfg.hpp, 58
getmac, 59
loadCfg, 60
shit, 60

MAX_EP_EVTS
Obd.hpp, 63
map_commands
Obd, 24
MockSocket.cpp, 60
findDevPTS, 61

NOGPSDATA
gpsclient.cpp, 54

Obd, 18
connectBluetooth, 20
disconnectBluetooth, 20
discoverDeviceAddress, 20
existPID, 21
getDTCs, 21
getVIN, 21
initDecoderFunctions, 22
initMessages, 22
isValid, 22
map_commands, 24
Obd, 19
polling, 22
printPIDs, 23
printStatus, 23
readFileData, 23
send, 23
Obd.hpp, 62
MAX_EP_EVTS, 63
OxigenoResponse, 24
A, 24
B, 25

PID_BITS
decoders.hpp, 44
polling
Obd, 22
printPIDs
Obd, 23
printStatus
Obd, 23

readFileData
Obd, 23
RelacionesResponse, 25
A, 25
B, 26
C, 26
D, 26

STATUS_BITS
decoders.hpp, 44
send
Obd, 23
sendAlarm
AlarmFile, 8
setBytesResponse
Commands, 14
setCMD
Commands, 14
setDecoder
Commands, 14
setDescription
Commands, 15
setMAX
Commands, 15
setMIN
Commands, 15
setName
Commands, 16
setResValue
Commands, 16
setTypeData
Commands, 16
setUnits
Commands, 17
shit
loadcfg.cpp, 58
loadcfg.hpp, 60

UnitTestCase.cpp, 64
BUFSIZE, 66
CATCH_CONFIG_MAIN, 66
getMinicomCMD, 66
initOBDSIM, 67
WAIT_OBDSIM, 66

WAIT_OBDSIM
UnitTestCase.cpp, 66

Índice de Figuras

1.1	Arquitectura general y componentes	2
3.1	Modulo de control de la transmisión (TCM)	8
3.2	Forma del conector OBDII	9
3.3	DLC para el protocolo ISO 15765-4/SAE J2480 (CAN)	9
3.4	Esquema básico del bus CAN entre ECUs	10
3.5	Esquema eléctrico genérico de un vehículo con bus CAN	10
3.6	ELM37 con interfaz Bluetooth	11
3.7	Formato de los bits de respuesta de los PIDs	12
3.8	Distintas representaciones de una MIL	12
3.9	Niveles de tensión del bus CAN	13
3.10	Trama CAN estándar	13
3.11	ISO 15765-4 (OBDII CAN BUS) e ISO 11898(CAN) en el modelo OSI	14
3.12	Trama OBDII	14
4.1	Diagrama de casos de uso general del sistema	21
4.2	Diagrama de casos de uso gestión de la conexión ELM327	22
4.3	Diagrama de casos de uso gestión del envío y recepción de mensajes OBD	25
4.4	Diagrama de casos de uso gestión de lectura de ficheros de configuración	28
4.5	Diagrama de casos de uso gestión de los datos	29
5.1	Diagrama de paquetes del sistema	39
5.2	Diagrama de componentes del sistema	40
5.3	Diagrama de clases del sistema	42
5.4	Diagrama de actividad del sistema	43
5.5	Diagrama de secuencia del sistema	44
6.1	Distribución de código del proyecto obd2-bluetooth	46
6.2	Distribución de código del proyecto obd2-server	47
6.3	Interfaz web de inicio del servidor web remoto	48
6.4	Interfaz web de DTC de vehículos del servidor web remoto	49
6.5	DTC recepcionados en la BBDD MongoDB en Compass	50
7.1	Interfaz gráfica del simulador OBDSIM	51
A.1	Arquitectura general y componentes para la instalación	63

Índice de Tablas

3.1	Modos de operación PIDs	12
3.2	Interpretación bit más significativo DTC	16
3.3	Comandos AT utilizados	17
4.1	ACT-01: Usuario	20
4.2	ACT-02: ELM327	20
4.3	ACT-03: Servidor Web (BBDD)	20
4.4	ACT-04: Archivos de configuración	20
4.5	CU-01: Descubrir dispositivo bluetooth	22
4.6	CU-02: Conectar con dispositivo ELM327	23
4.7	CU-03: Cargar comandos de inicialización	23
4.8	CU-04: Cargar los decodificadores PID y AT	23
4.9	CU-05: Consultar el estado de la conexión	24
4.10	CU-06: Leer archivo de parámetros de conexión	24
4.11	CU-07: Leer archivo JSON	24
4.12	CU-08: Enviar comando	25
4.13	CU-09: Consultar DTC	25
4.14	CU-10: Consultar VIN	26
4.15	CU-11: Solicitar pruebas de estado del vehículo	26
4.16	CU-12: Consultar lista de PID implementados	26
4.17	CU-13: Recibir respuesta	27
4.18	CU-14: Identificar los datos útiles de la respuesta	27
4.19	CU-15: Decodificar datos	27
4.20	CU-16: Almacenar respuesta	27
4.21	CU-17: Mostrar información del mensaje recibido	28
4.22	CU-18: Mostrar DTC	28
4.23	CU-19: Mostrar VIN	28
4.24	CU-20: Mostrar resultado de los test	29
4.25	CU-21: Mostrar PIDs disponibles y valores almacenados	29
4.26	CU-22: Enviar mensaje al servidor	30
4.27	CU-23: Enviar mensaje en formato JSON	30
4.28	RFI-01: Información del dispositivo bluetooth ELM327	30
4.29	RFI-02: Información de comandos AT y OBD	31
4.30	RFI-03: Información del servidor web	31
4.31	RFI-04: Información del mensaje de envío al servidor	31
4.32	RFI-05: Información relativa al vehículo	32
4.33	RFN-01: Intervalo de muestreo	32
4.34	RFN-02: Conexiones simultáneas al ELM327	32
4.35	RFN-03: Reconexión ante caída del servicio	33
4.36	RFN-04: Incrementar tiempo entre envíos tras detectar DTC	33
4.37	RFN-05: No debe interferir en el funcionamiento del vehículo	33

4.38	RFC-01: Mostrar evolución del intercambio de mensajes OBD	33
4.39	RNFF-01: Tiempo de recuperación del sistema menor de 1 minuto	34
4.40	RNFF-02: Reinicio del sistema no implica pérdida de datos	34
4.41	RNFF-03: Fallos en la respuesta del vehículo no implican corte en el servicio	34
4.42	RNFU-01: Utilización con un único fichero de cabecera	34
4.43	RNFU-02: Acceso rápido a datos del vehículo sin necesidad de comprensión del funcionamiento	35
4.44	RNFU-03: Utilización de comandos mediante su nombre descriptivo	35
4.45	RNFE-01: Tiempo de respuesta a una solicitud debe ser inferior a 1 segundo	35
4.46	RNFE-02: Disponibilidad de mantenimiento del servicio	35
4.47	RNFE-03: Independencia con el servidor web	36
4.48	RNFM-01: Amplia modularidad del sistema	36
4.49	RNFM-02: Cambios en el sistema a través de ficheros de configuración	36
4.50	RNFP-01: Autoconfiguración del servicio a través de script	36
4.51	RNFP-02: Distribuciones derivadas de Debian	37
4.52	RNFS-01: Acceso físico al dispositivo donde se ejecute el software debe estar restringido	37
4.53	RNFS-02: Envío de mensajes al servidor web	37
4.54	RT-01: Debe ser un sistema distribuido	37
4.55	RT-02: Características del cliente	38
4.56	RT-03: Características del servidor	38
4.57	RT-04: Lenguaje de programación C++	38
4.58	RI-01: Integración con librerías externas	38
7.1	PU-01: Decodificadores	52
7.2	PU-02: Conversor DTC	52
7.3	PU-03: Conversor VIN	52
7.4	PI-01: Mostrar DTC	53
7.5	PI-02: Mostrar VIN	53
7.6	PI-03: Mostrar resultados de los test	54
7.7	PI-04: Mostrar PIDs disponibles	54
7.8	PI-05: Obtener valor específico del vehículo	55

Índice de Códigos

6.1 Formato del mensaje JSON enviado al servido remoto 48

7.1 Ejemplo de ejecución de pruebas automatizadas y sus resultados 55
7.2 Ejemplo de ejecución de prueba VIN en vehículo 58

A.1 Descarga del código fuente obd2-bluetooth 63
A.2 Configuración del fichero configuration.cfg 63
A.3 Descripción de variables de configuration.cfg 64
A.4 Compilación con make 65
A.5 Descripción del fichero de configuración del servicio 66
A.6 Instalación de la aplicación como servicio en Linux 66
A.7 Descarga del código fuente de obd2-server 66
A.8 Instalación de Node.js 67
A.9 Instalación de MongoDB 67
A.10 Instalación obdsim y minicom 67
A.11 Generar ejecutable de las pruebas 67
A.12 Ejecución de las pruebas con el ejecutable test 67

B.1 Operaciones disponibles del servicio con systemd 69
B.2 Comprobación y arranque del servicio mongod 70
B.3 Iniciar el servidor remoto 70

Bibliografía

[1] N. Lohmann. (2020) Json for modern c++. [Online]. Available: https://github.com/nlohmann/json

[2] C. Org. (2020) Catch2, multi-paradigm test framework for c++. [Online]. Available: https://github.com/catchorg/Catch2

[3] B. S. I. G. (SIG). (2001) Bluez, official linux bluetooth protocol stack. [Online]. Available: https://www.bluez.org/about/

[4] I. of Electrical and E. E. (IEEE). (1995) Posix threads. [Online]. Available: https://en.wikipedia.org/wiki/POSIX_Threads

[5] S. H. P. Ltd. (2008) Sublime text. [Online]. Available: https://www.sublimetext.com/3

[6] M. Pedersen. (2016) Sublimegit. [Online]. Available: https://sublimegit.readthedocs.io/en/latest/

[7] quarnster. (2019) Sublimegdb. [Online]. Available: https://packagecontrol.io/packages/SublimeGDB

[8] S. F. Conservancy. (2005) Git. [Online]. Available: https://git-scm.com/

[9] I. Org. (2005) Obdsim, simulate an elm327 device. [Online]. Available: https://icculus.org/obdgpslogger/obdsim.html

[10] O. I. de Normalización. (2016) Road vehicles - diagnostics on controller area networks (can) - part 4: Requirements for emissions-related systems. [Online]. Available: https://www.iso.org/standard/67245.html

[11] S. of Automotive Engineers. (2002) Sae j2012: Diagnostic trouble code definitions. [Online]. Available: https://law.resource.org/pub/us/cfr/ibr/005/sae.j2012.2002.pdf

[12] ——. (2002) Sae j1979: E/e diagnostic test modes. [Online]. Available: https://law.resource.org/pub/us/cfr/ibr/005/sae.j1979.2002.pdf

[13] C. S. . S. E. S. Committee. (2011) Systems and software engineering — life cycle processes — requirements engineering. [Online]. Available: https://www.iso.org/standard/45171.html

[14] I. MongoDB. (2020) Install mongodb community edition on debian. [Online]. Available: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-debian/

[15] D. van Heesch. (2020) Doxygen. [Online]. Available: https://www.doxygen.nl/index.html