obd2-bluetooth

1.0

Generado por Doxygen 1.8.13

Índice general

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:

Capítulo 3

Indice de archivos

3.1. Lista de archivos

Lista de todos los archivos documentados y con descripciones breves:

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.
- std::string getGeoPos (std::string serialPort)

  Método que obtiene la geolocalización del vehículo.
- bool hasGps ()

  Método para mostrar si tiene disponible el GPS.
- void Gps (bool installed)

  Método para indicar que está disponible el GPS.
- 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 45 del archivo alarmfile.cpp.

4.1.3. Documentación de las funciones miembro

4.1.3.1. getGeoPos()

```
std::string AlarmFile::getGeoPos (
            std::string serialPort )
```

Método que obtiene la geolocalización del vehículo.

Parámetros

serialPort	String con la ruta al dispositivo GPS.

Devuelve

String con la geolocalización del dispositivo GPS.

Definición en la línea 24 del archivo alarmfile.cpp.

4.1.3.2. Gps()

```
void AlarmFile::Gps (
            bool installed )
```

Método para indicar que está disponible el GPS.

Parámetros

installed	Booleano que indica si está instalado el GPS (true).

Definición en la línea 34 del archivo alarmfile.cpp.

4.1.3.3. hasGps()

```
bool AlarmFile::hasGps ( )
```

Método para mostrar si tiene disponible el GPS.

Devuelve

Booleano, true si está instalado y false en caso contrario.

Definición en la línea 29 del archivo alarmfile.cpp.

4.1.3.4. 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 63 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 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

4.3.1. 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 EL↩ M327 y el posterior envío y recepción de mensajes OBD.

Definición en la línea 73 del archivo Obd.hpp.

4.3.2. Documentación del constructor y destructor

4.3.2.1. 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.

4.3.3. Documentación de las funciones miembro

4.3.3.1. 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.

4.3.3.2. disconnectBluetooth()

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

Método de desconexión bluetooth con el dispositivo ELM327.

Cierra el socket e instancia epoll abiertas.

Definición en la línea 548 del archivo Obd.hpp.

4.3.3.3. 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.

4.3.3.4. 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 560 del archivo Obd.hpp.

4.3.3.5. 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 624 del archivo Obd.hpp.

4.3.3.6. 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 611 del archivo Obd.hpp.

4.3.3.7. 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 507 del archivo Obd.hpp.

4.3.3.8. 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 472 del archivo Obd.hpp.

4.3.3.9. 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 651 del archivo Obd.hpp.

4.3.3.10. 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 292 del archivo Obd.hpp.

4.3.3.11. 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 579 del archivo Obd.hpp.

4.3.3.12. 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 599 del archivo Obd.hpp.

4.3.3.13. 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 228 del archivo Obd.hpp.

4.3.3.14. 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 248 del archivo Obd.hpp.

4.3.4. Documentación de los campos

4.3.4.1. 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

4.4. 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

4.4.1. 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.

4.4.2. Documentación de los campos

4.4.2.1. A

```
float OxigenoResponse::A
```

Valor A en la formula de decodificación

Definición en la línea 26 del archivo decoders.hpp.

4.4.2.2. 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

4.5. 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

4.5.1. 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.

4.5.2. Documentación de los campos

4.5.2.1. A

```
int RelacionesResponse::A
```

Valor A en la formula de decodificación

Definición en la línea 35 del archivo decoders.hpp.

4.5.2.2. B

```
int RelacionesResponse::B
```

Valor B en la formula de decodificación

Definición en la línea 36 del archivo decoders.hpp.

4.5.2.3. C

```
int RelacionesResponse::C
```

Valor C en la formula de decodificación

Definición en la línea 37 del archivo decoders.hpp.

4.5.2.4. 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

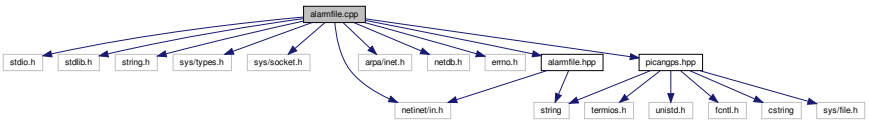
Documentación de archivos

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"
#include "picangps.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

Definición en el archivo alarmfile.cpp.

5.2. alarmfile.cpp

```
00001
00008 #include <stdio.h>
00009 #include <stdlib.h>
00010 #include <string.h>
00011 #include <sys/types.h>
00012 #include <sys/socket.h>
00013 #include <netinet/in.h>
00014 #include <arpa/inet.h>
00015 #include <netdb.h>
00016 #include <errno.h>
00017 #include <stdio.h>
00018 #include "alarmfile.hpp"
00019
00020 #include "picangps.hpp"
00021
00022 bool AlarmFile::gps_ = false;
00023
00024 std::string AlarmFile::getGeoPos(std::string serialPort)
00025 {
00026   return PicanGetGPS(serialPort);
00027 }
00028
00029 bool AlarmFile::hasGps()
00030 {
00031   return AlarmFile::gps_;
00032 }
00033
00034 void AlarmFile::Gps(bool installed)
00035 {
00036   AlarmFile::gps_ = installed;
00037 }
00038
00039 void AlarmFile::shit (const char *mens)
00040 {
00041   fprintf(stderr, "%s - %d\n", mens, errno);
00042   perror("Error is ");
00043 }
00044
00045 AlarmFile::AlarmFile(std::string AlarmHost,
00046                      std::string AlarmPort,
00047                      std::string AlarmFilename,
00048                      std::string LastAlarmFilename):
00049   alarmHost_(AlarmHost),
00050   alarmPort_(atoi(AlarmPort.c_str())),
00051   alarmFilename_(AlarmFilename),
00052   lastAlarmFilename_(LastAlarmFilename)
00053 {
00054   if ((s_ = socket (AF_INET, SOCK_DGRAM, 0)) < 0) shit ("socket");
00055
00056   memset ((char *)&iTu_, 0, sizeof(struct sockaddr_in));
00057   iTu_.sin_family      = AF_INET;
00058   iTu_.sin_addr.s_addr = inet_addr(alarmHost_.c_str());
00059   iTu_.sin_port        = htons(alarmPort_);
00060 }
00061
00062
00063 bool AlarmFile::sendAlarm(std::string msg)
00064 {
00065   if ( "" != msg ){
00066     int cc =  sendto(s_, msg.c_str(), strlen(msg.c_str()), 0,(struct sockaddr *)&iTu_, sizeof(iTu_));
00067
00068     if (cc < 0){
00069       perror("Error is ");
00070       fprintf(stderr, "Value of errno: %d\n", errno);
00071     }
00072
00073     fprintf(stderr, "Alarma enviada (%d): %s a %s:%d\n", cc, msg.c_str(), inet_ntoa(iTu_.sin_addr), (int)
      ntohs(iTu_.sin_port));
00074
00075     return true;
00076   }
00077   return false;
00078 }
00079
```

5.3. 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:

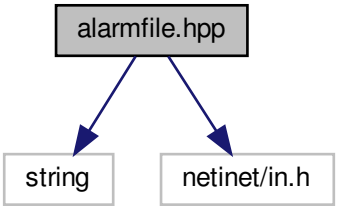
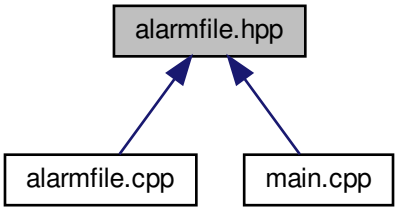


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- class AlarmFile

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

5.3.1. 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

Definición en el archivo alarmfile.hpp.

5.4. alarmfile.hpp

```
00001
00009 #ifndef ALARMFILE_HPP
00010 #define ALARMFILE_HPP
00011
00012 #include <string>
00013 #include <netinet/in.h>
00014
00021 class AlarmFile{
00022 public:
00023
00033   AlarmFile(std::string AlarmHost,
00034             std::string AlarmPort,
00035             std::string AlarmFilename,
00036             std::string LastAlarmFilename);
00037
00044   std::string getGeoPos(std::string serialPort);
00045
00051   bool hasGps();
00052
00058   void Gps(bool installed);
00059
00066   bool sendAlarm(std::string msg);
00067 private:
00068
00074   void shit (const char *mens);
00075   static bool gps_;
00076   std::string alarmHost_;
00077   int alarmPort_;
00078   std::string alarmFilename_;
00079   std::string lastAlarmFilename_;
00080   int s_;
00081   struct sockaddr_in iTu_;
00082 };
00083
00084 #endif
```

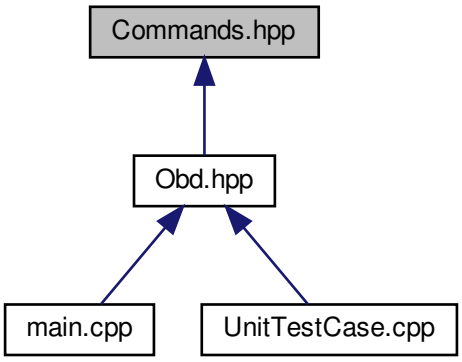
5.5. Referencia del Archivo Commands.hpp

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.

5.5.1. 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

Definición en el archivo Commands.hpp.

5.6. Commands.hpp

```
00001
00008 #ifndef COMMANDS_HPP
00009 #define COMMANDS_HPP
00010
00011 #include <iostream>
00012 #include <any>
00013
00014 #include "external/json.hpp"
00015 #include "decoders.hpp"
00016
00021 using json = nlohmann::json;
00022
00029 class Commands {
00030 public:
00031
00038     Commands(json data)
00039     :   m_name(data["name"]),
00040         m_description(data["description"]),
00041         m_cmd(data["cmd"]),
00042         m_bytes_response(data["bytes_response"]),
00043         m_decoder(data["decoder"]),
00044         m_min_unit(data["min_unit"]),
00045         m_max_unit(data["max_unit"]),
00046         m_units(data["units"]),
00047         m_type_data(data["type_data"])
00048     {}
00049
00055     std::string getName(){ return this->m_name; }
00056
00062     std::string getDescription(){ return this->m_description; }
00063
00069     std::string getCMD(){ return this->m_cmd; }
00070
00076     int getBytesResponse(){ return this->m_bytes_response; }
00077
00083     std::string getDecoder(){ return this->m_decoder; }
00084
00090     float getMIN(){ return this->m_min_unit; }
00091
00097     float getMAX(){ return this->m_max_unit; }
00098
00104     std::string getUnits(){ return this->m_units; }
00105
00115     std::string getTypeData(){ return this->m_type_data; }
00116
00122     std::any getResValue(){ return this->m_resValue; }
00123
00132     json getJson(){
00133         json data;
00134
00135         try{
00136             if (getTypeData() == "int"){
00137                 auto resValue = std::any_cast<int>(this->m_resValue);
00138                 data["value"] = std::to_string(resValue);
00139             }
00140             else if (getTypeData() == "float"){
00141                 auto resValue = std::any_cast<float>(this->m_resValue);
00142                 data["value"] = std::to_string(resValue);
00143             }
00144             else if (getTypeData() == "string"){
00145                 auto resValue = std::any_cast<std::string>(this->m_resValue);
00146                 data["value"] = resValue;
00147             }
00148             else if (getTypeData() == "vectorStr"){
00149                 auto resValue = std::any_cast<std::vector<std::string»(this->m_resValue);
00150                 data["value"] = resValue;
00151             }
00152             else if (getTypeData() == "vectorInt"){
00153                 auto resValue = std::any_cast<std::vector<int»(this->m_resValue);
00154                 data["value"] = resValue;
00155             }
00156             else if (getTypeData() == "map"){
00157                 auto resValue = std::any_cast<std::map<std::string, std::string»(this->m_resValue);
00158                 data["value"] = resValue;
00159             }
00160             else if (getTypeData() == "OxigenoResponse"){
00161                 auto resValue = std::any_cast<struct OxigenoResponse>(this->m_resValue);
00162                 std::map<std::string, float> mapResValue;
00163                 mapResValue["A"] = resValue.A;
00164                 mapResValue["B"] = resValue.B;
00165                 data["value"] = mapResValue;
00166             }
00167             else if (getTypeData() == "RelacionesResponse"){
00168                 auto resValue = std::any_cast<struct RelacionesResponse>(this->m_resValue
```

```
    );
00169                 std::map<std::string, int> mapResValue;
00170                 mapResValue["A"] = resValue.A;
00171                 mapResValue["B"] = resValue.B;
00172                 mapResValue["C"] = resValue.C;
00173                 mapResValue["D"] = resValue.D;
00174                 data["value"] = mapResValue;
00175             }
00176 
00177         } catch(const std::bad_any_cast& e) {
00178             std::cerr << e.what() << std::endl;
00179         }
00180         data["name"] = this->m_name;
00181         data["description"] = this->m_description;
00182         data["units"] = this->m_units;
00183 
00184 
00185         return data;
00186     }
00187 
00195     std::string getCMDResponse() {
00196         std::string CMDResponse;
00197         CMDResponse = this->m_cmd;
00198         CMDResponse.replace(0, 1, "4");
00199         return CMDResponse;
00200     }
00201 
00207     void setName(std::string name) { this->m_name = name; }
00208 
00214     void setDescription(std::string description) { this->m_description = description; }
00215 
00221     void setCMD(std::string cmd) { this->m_cmd = cmd; }
00222 
00228     void setBytesResponse(int bytes_response) { this->m_bytes_response = bytes_response; }
00229 
00235     void setDecoder(std::string decoder) { this->m_decoder = decoder; }
00236 
00242     void setMIN(float min_unit) { this->m_min_unit = min_unit; }
00243 
00249     void setMAX(float max_unit) { this->m_max_unit = max_unit; }
00250 
00256     void setUnits(std::string units) { this->m_units = units; }
00257 
00263     void setTypeData(std::string type_data) { this->m_type_data = type_data; }
00264 
00272     void setResValue(auto resValue) { this->m_resValue = resValue; }
00273 private:
00274     // Atributos privados de la clase "Commands"
00275     std::string m_name;
00276     std::string m_description;
00277     std::string m_cmd;
00278     int m_bytes_response;
00279     std::string m_decoder;
00280     float m_min_unit;
00281     float m_max_unit;
00282     std::string m_units;
00283     std::string m_type_data;
00284     std::any m_resValue;
00285 };
00286 
00287 
00288 #endif
```

5.7. 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:

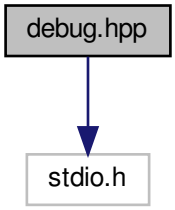
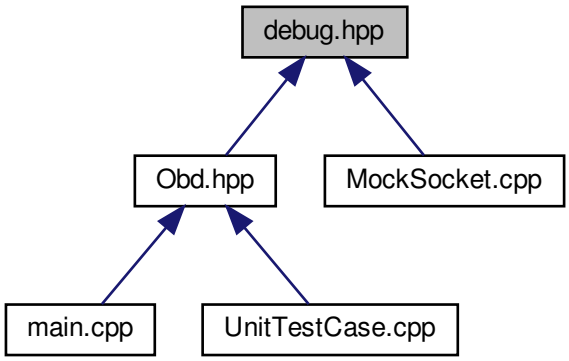


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



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.

5.7.1. 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

Definición en el archivo debug.hpp.

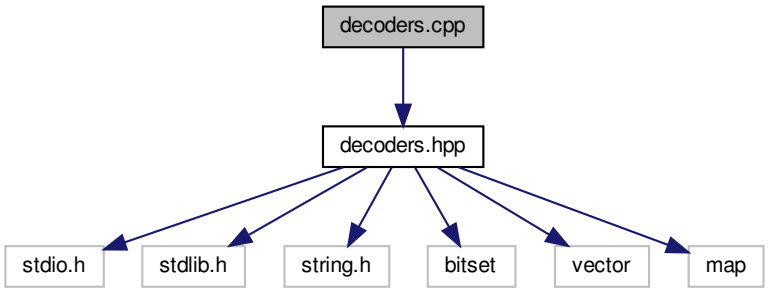
5.8. debug.hpp

```
00001
00008 #include <stdio.h>
00009
00010 #ifndef DEBUG_HPP
00011 #define DEBUG_HPP
00012 #ifdef DEBUG
00013
00016     #define debugLog(info, args...) \
00017         fprintf (stderr, "[%s %s][LOG][%s][%s][Line %i] ", __DATE__, __TIME__, __FILE__, __FUNCTION__,
      __LINE__); \
00018         fprintf (stderr, info "\n", ##args);
00019
00022     #define debugError(info, args...) \
00023         fprintf (stderr, "[%s %s][ERROR][%s][%s][Line %i] ", __DATE__, __TIME__, __FILE__, __FUNCTION__,
      __LINE__); \
00024         fprintf (stderr, info "\n", ##args);
00025 #else
00026
00029     #define debugLog(format, args...) do{} while(0);
00030
00033     #define debugError(format, args...) do{} while(0);
00034 #endif
00035 #endif
```

5.9. 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).
- 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.

5.9.1. 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

Definición en el archivo decoders.cpp.

5.9.2. Documentación de las funciones

5.9.2.1. 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 47 del archivo decoders.cpp.

5.9.2.2. 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 284 del archivo decoders.cpp.

5.9.2.3. 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 316 del archivo decoders.cpp.

5.9.2.4. 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 270 del archivo decoders.cpp.

5.9.2.5. 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.

5.9.2.6. 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 86 del archivo decoders.cpp.

5.9.2.7. 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 298 del archivo decoders.cpp.

5.9.2.8. 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 112 del archivo decoders.cpp.

5.9.2.9. 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 384 del archivo decoders.cpp.

5.9.2.10. 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 291 del archivo decoders.cpp.

5.9.2.11. 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 391 del archivo decoders.cpp.

5.9.2.12. 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 442 del archivo decoders.cpp.

5.9.2.13. 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 398 del archivo decoders.cpp.

5.9.2.14. 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 453 del archivo decoders.cpp.

5.9.2.15. 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 496 del archivo decoders.cpp.

5.9.2.16. 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 520 del archivo decoders.cpp.

5.9.2.17. 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 305 del archivo decoders.cpp.

5.9.2.18. 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 341 del archivo decoders.cpp.

5.9.2.19. 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 128 del archivo decoders.cpp.

5.9.2.20. 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 476 del archivo decoders.cpp.

5.9.2.21. 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 277 del archivo decoders.cpp.

5.9.2.22. 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 327 del archivo decoders.cpp.

5.9.2.23. decodeVIN()

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

Función de decodificación del Número de Identificación del Vehículo (VIN).

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.

5.9.2.24. 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 485 del archivo decoders.cpp.

5.10. decoders.cpp

```
00001
00008 #include "decoders.hpp"
00009
00010 /*
00011 Definición de la función
00012 */
00013
00014 //Modo AT
00015 void noDecodeAT(){}
00016 std::string decodeDescribeProtocol(char * response){
00017     std::string protocol(response);
00018     return protocol;
00019 }
00020 //Modo 09
00021 std::string decodeVIN(char * response){
00022     std::string bytes_res(response);
00023     std::string vin;
00024
00025     //División de orden y datos
00026     std::string order = bytes_res.substr(0,2);
00027     std::string vin_bytes = bytes_res.substr(2,42);
00028
00029
00030     std::size_t found = vin_bytes.find("\n");
00031     while(found!=std::string::npos){
00032         vin_bytes.erase(found,3);
00033         found = vin_bytes.find("\n");
00034     }
00035
00036
00037     //Conversión en ASCII
00038     for (uint32_t i = 0; i < vin_bytes.size(); i+=2){
00039         std::string vin_char = vin_bytes.substr(i,2);
00040         //Conversión de bytes en char
00041         vin.push_back((char) stoi(vin_char,nullptr,16));
00042     }
00043
00044     return vin;
00045 }
00046
00047 std::string convertDTCs(std::string dtc){
00048     if(dtc[0] == '0'){
00049         dtc.replace(0,1,"P0");
00050     } else if (dtc[0] == '1'){
00051         dtc.replace(0,1,"P1");
00052     } else if (dtc[0] == '2'){
00053         dtc.replace(0,1,"P2");
00054     } else if (dtc[0] == '3'){
00055         dtc.replace(0,1,"P3");
00056     } else if (dtc[0] == '4'){
00057         dtc.replace(0,1,"C0");
```

```
00058     } else if (dtc[0] == '5'){
00059         dtc.replace(0,1,"C1");
00060     } else if (dtc[0] == '6'){
00061         dtc.replace(0,1,"C2");
00062     } else if (dtc[0] == '7'){
00063         dtc.replace(0,1,"C3");
00064     } else if (dtc[0] == '8'){
00065         dtc.replace(0,1,"B0");
00066     } else if (dtc[0] == '9'){
00067         dtc.replace(0,1,"B1");
00068     } else if (dtc[0] == 'A'){
00069         dtc.replace(0,1,"B2");
00070     } else if (dtc[0] == 'B'){
00071         dtc.replace(0,1,"B3");
00072     } else if (dtc[0] == 'C'){
00073         dtc.replace(0,1,"U0");
00074     } else if (dtc[0] == 'D'){
00075         dtc.replace(0,1,"U1");
00076     } else if (dtc[0] == 'E'){
00077         dtc.replace(0,1,"U2");
00078     } else if (dtc[0] == 'F'){
00079         dtc.replace(0,1,"U3");
00080     }
00081
00082     return dtc;
00083 }
00084
00085 //Modo 03
00086 std::vector<std::string> decodeDTCs(char *response){
00087     std::vector<std::string> vec_dtcs;
00088     std::string bytes_res(response);
00089
00090     std::string dtc_1 = bytes_res.substr(0,4);
00091     if (dtc_1.compare("0000")){
00092         dtc_1 = convertDTCs(dtc_1);
00093         vec_dtcs.push_back(dtc_1);
00094     }
00095     std::string dtc_2 = bytes_res.substr(4,4);
00096     if (dtc_2.compare("0000")){
00097         dtc_2 = convertDTCs(dtc_2);
00098         vec_dtcs.push_back(dtc_2);
00099     }
00100     std::string dtc_3 = bytes_res.substr(8,4);
00101     if (dtc_3.compare("0000")){
00102         dtc_3 = convertDTCs(dtc_3);
00103         vec_dtcs.push_back(dtc_3);
00104     }
00105
00106     return vec_dtcs;
00107 }
00108
00109     //Modo 01-> Descripcion - PID - Valor Mínimo - Valor Máximo - Unidad - Fórmula
00110 //00 - PIDs implementados [01 - 20] -Cada bit indica si los siguientes 32 PID están implementados (1) o no
       (0): [A7..D0] == [PID 01..20]
00111
00112 std::vector<int> decodePIDS(char *response){
00113     //Conversión a long para poder convertirlo a bitset
00114     long value_rcv = std::stol(response, nullptr, 16);
00115     //Conversión a bitset
00116     std::bitset<PID_BITS> setBit (value_rcv);
00117     std::vector<int> vec_pids;
00118     //Comprobación de PIDs disponibles(bitset lectura al reves)
00119     for (int i = PID_BITS-1; i >= 0; i--){
00120         if(setBit[i]){
00121             vec_pids.push_back(PID_BITS-i);
00122         }
00123     }
00124     return vec_pids;
00125 }
00126
00127 //01 - Estado de los monitores de diagnóstico desde que se borraron los códigos de fallas DTC; incluye el
       estado de la luz indicadora de fallas, MIL, y la cantidad de códigos de fallas DTC
00128 std::map<std::string, std::string> decodeStatus(char *response){
00129     std::map<std::string, std::string> status;
00130
00131     std::string bytes_res(response);
00132     std::string responseA = bytes_res.substr(0,2);
00133     std::string responseB = bytes_res.substr(2,2);
00134     std::string responseC = bytes_res.substr(4,2);
00135     std::string responseD = bytes_res.substr(6,2);
00136
00137     int intA = std::stoi(responseA, nullptr, 16);
00138     int intB = std::stoi(responseB, nullptr, 16);
00139     int intC = std::stoi(responseC, nullptr, 16);
00140     int intD = std::stoi(responseD, nullptr, 16);
00141
00142     std::bitset<STATUS_BITS> byteA (intA);
```

```
00143     std::bitset<STATUS_BITS> byteB (intB);
00144     std::bitset<STATUS_BITS> byteC (intC);
00145     std::bitset<STATUS_BITS> byteD (intD);
00146 
00147     if (byteA[7]){
00148         status["MIL"] = "Encendida";
00149         status["DTC_CNT"] = std::to_string(intA-128);
00150     } else {
00151         status["MIL"] = "Apagada";
00152         status["DTC_CNT"] = std::to_string(intA);
00153     }
00154 
00155     if (byteB[0]){
00156         if (byteB[4])
00157             status["Sistema de detección de condiciones inadecuadas de ignición en cilindros"] = "Prueba
      Incorrecta";
00158         else
00159             status["Sistema de detección de condiciones inadecuadas de ignición en cilindros"] = "Prueba
      Correcta";
00160     }
00161     if (byteB[1]){
00162         if (byteB[5])
00163             status["Sistema de combustible"] = "Prueba Incorrecta";
00164         else
00165             status["Sistema de combustible"] = "Prueba Correcta";
00166     }
00167     if (byteB[2]){
00168         if (byteB[6])
00169             status["Sistema de componentes integrales"] = "Prueba Incorrecta";
00170         else
00171             status["Sistema de componentes integrales"] = "Prueba Correcta";
00172     }
00173 
00174     if (byteB[3]){
00175         status["IGNICION"] = "Chispa";
00176         if (byteC[0]){
00177             if (byteD[0])
00178                 status["Sistema de eficiencia del convertidor catalítico"] = "Prueba Incorrecta";
00179             else
00180                 status["Sistema de eficiencia del convertidor catalítico"] = "Prueba Correcta";
00181         }
00182         if (byteC[1]){
00183             if (byteD[1])
00184                 status["Sistema de calentamiento de convertidor catalítico"] = "Prueba Incorrecta";
00185             else
00186                 status["Sistema de calentamiento de convertidor catalítico"] = "Prueba Correcta";
00187         }
00188         if (byteC[2]){
00189             if (byteD[2])
00190                 status["Sistema evaporativo"] = "Prueba Incorrecta";
00191             else
00192                 status["Sistema evaporativo"] = "Prueba Correcta";
00193         }
00194         if (byteC[3]){
00195             if (byteD[3])
00196                 status["Sistema secundario de aire"] = "Prueba Incorrecta";
00197             else
00198                 status["Sistema secundario de aire"] = "Prueba Correcta";
00199         }
00200         if (byteC[4]){
00201             if (byteD[4])
00202                 status["Sistema de fugas de aire acondicionado"] = "Prueba Incorrecta";
00203             else
00204                 status["Sistema de fugas de aire acondicionado"] = "Prueba Correcta";
00205         }
00206         if (byteC[5]){
00207             if (byteD[5])
00208                 status["Sistema de sensores de oxígeno"] = "Prueba Incorrecta";
00209             else
00210                 status["Sistema de sensores de oxígeno"] = "Prueba Correcta";
00211         }
00212         if (byteC[6]){
00213             if (byteD[6])
00214                 status["Sistema de calentamiento del sensor de oxígeno"] = "Prueba Incorrecta";
00215             else
00216                 status["Sistema de calentamiento del sensor de oxígeno"] = "Prueba Correcta";
00217         }
00218         if (byteC[7]){
00219             if (byteD[7])
00220                 status["Sistema de recirculación de los gases de escape (Exhaust Gas Recicrulation, EGR)"]
      = "Prueba Incorrecta";
00221             else
00222                 status["Sistema de recirculación de los gases de escape (Exhaust Gas Recicrulation, EGR)"]
      = "Prueba Correcta";
00223         }
00224     } else {
00225         status["IGNICION"] = "Compresión";
```

```
00226         if (byteC[0]){
00227             if (byteD[0])
00228                 status["Sistema de catalizador NMHC"] = "Prueba Incorrecta";
00229             else
00230                 status["Sistema de catalizador NMHC"] = "Prueba Correcta";
00231         }
00232         if (byteC[1]){
00233             if (byteD[1])
00234                 status["Sistema monitor de NOx/SCR"] = "Prueba Incorrecta";
00235             else
00236                 status["Sistema monitor de NOx/SCR"] = "Prueba Correcta";
00237         }
00238         if (byteC[3]){
00239             if (byteD[3])
00240                 status["Sistema de presión de impulso"] = "Prueba Incorrecta";
00241             else
00242                 status["Sistema de presión de impulso"] = "Prueba Correcta";
00243         }
00244         if (byteC[5]){
00245             if (byteD[5])
00246                 status["Sistema del sensor de gases de escape"] = "Prueba Incorrecta";
00247             else
00248                 status["Sistema del sensor de gases de escape"] = "Prueba Correcta";
00249         }
00250         if (byteC[6]){
00251             if (byteD[6])
00252                 status["Sistema de monitor del filtro de partículas (Particular Matter, PM)"] = "Prueba
      Incorrecta";
00253             else
00254                 status["Sistema de monitor del filtro de partículas (Particular Matter, PM)"] = "Prueba
      Correcta";
00255         }
00256         if (byteC[7]){
00257             if (byteD[7])
00258                 status["Sistema de recirculación de gases de escape (Exhaust Gas Recirculation, EGR) y/o
      VVT"] = "Prueba Incorrecta";
00259             else
00260                 status["Sistema de recirculación de gases de escape (Exhaust Gas Recirculation, EGR) y/o
      VVT"] = "Prueba Correcta";
00261         }
00262     }
00263 
00264     return status;
00265 }
00266 //02
00267 //03
00268 //04 - Carga calculada del motor , 0 , 100 , % , A/2.55
00269 
00270 float decodeCargaPosicionEGR(char *response){
00271     int dec = (int)strtol(response, NULL, 16);
00272     return dec/2.55;
00273 }
00274 
00275 //05 - Temperatura del líquido de enfriamiento del motor , -40 , 215 , °C , A-40
00276 
00277 float decodeTempGeneral(char *response){
00278     int dec = (int)strtol(response, NULL, 16);
00279     return dec-40;
00280 }
00281 
00282 //06,07,08,09 - Ajuste de combustible a corto/largo plazo--Banco [1,2] , -100 (Reduccción de combustible:
      muy rico) , 99.2 (Aumento de combustible: muy magro) , % , A/1.28-100
00283 
00284 float decodeAjusteCombustibleEGR(char *response){
00285     int dec = (int)strtol(response, NULL, 16);
00286     return (dec/1.28)-100;
00287 }
00288 
00289 //0a - Presión del combustible , 0 , 765 , kPa , 3A
00290 
00291 float decodePresionCombustible(char *response){
00292     int dec = (int)strtol(response, NULL, 16);
00293     return 3*dec;
00294 }
00295 
00296 //0b - Presión absoluta del colector de admisión , 0 , 255 , kPa , A
00297 
00298 float decodeHexToDec(char *response){
00299     int dec = (int)strtol(response, NULL, 16);
00300     return dec;
00301 }
00302 
00303 //0c - RPM del motor , 0 , 16,383.75 , rpm , (256A+B)/4
00304 
00305 float decodeRPM(char *response){
00306     int dec = (int)strtol(response, NULL, 16);
00307     return dec/4.0;
```

```
00308 }
00309
00310 //0d - Velocidad del vehículo , 0 , 255 , km/h , A
00311
00312 //decodeHexToDec
00313
00314 //0e - Avance del tiempo , -64 , 63.5 , ° antes TDC , A/2-64
00315
00316 float decodeAvanceTiempo(char *response){
00317     int dec = (int)strtol(response, NULL, 16);
00318     return (dec/2.0)-64;
00319 }
00320
00321 //0f - Temperatura del aire del colector de admisión , -40 , 215 , °C, A-40
00322
00323 //decodeTempGeneral
00324
00325 //10 - Velocidad del flujo del aire MAF , 0 , 655.35  , gr/sec , (256A+B)/100
00326
00327 float decodeVelocidadMAF(char *response){
00328     int dec = (int)strtol(response, NULL, 16);
00329     return dec/100.0;
00330 }
00331
00332 //11 - Posición del acelerador , 0 , 100 , % , A/2.55
00333
00334 //decodeCargaPosicionEGR
00335
00336
00337 //12 - Estado del aire secundario controlado
00338 //13 - Presencia de sensores de oxígeno (en 2 bancos)
00339 //14,15,16,17,18,19,1a,1b - Sensor de oxígeno 1-8 A: Voltaje B: Ajuste de combustible a corto plazo , 0 100
       , 1.275 99.2 , voltios % , A: A/200 B: B/1.28-100 (Si B==FF, entonces el sensor no se usa en el cálculo del
      ajuste)
00340
00341 struct OxigenoResponse decodeSensorOxigeno(char *response){
00342     char AResponse[3], BResponse[3];
00343     float A, B;
00344     struct OxigenoResponse datos;
00345     //Añade caracter \0 al final de la cadena AResponse
00346     memset(AResponse, '\0', sizeof(AResponse));
00347     //Divide los bytes de respuesta
00348     strncpy( AResponse, response, 2);
00349     strcpy( BResponse, response + 2 );
00350
00351     A = (int)strtol(AResponse, NULL, 16);
00352     datos.A = A/200;
00353     /*
00354     Según la documentación si B==FF, entonces el sensor no se usa en el cálculo del ajuste
00355     if (strcmp(BResponse, "FF") == 0){
00356         //Si B==FF, entonces el sensor no se usa en el cálculo del ajuste
00357         datos.B=0;
00358     } else {
00359         //Si B!=FF, se aplica fórmula
00360         B = (int)strtol(BResponse, NULL, 16);
00361         datos.B = (B/1.28)-100;
00362     }
00363     */
00364     B = (int)strtol(BResponse, NULL, 16);
00365     datos.B = (B/1.28)-100;
00366
00367     return datos;
00368 }
00369
00370 //1c - Estándar OBD implementado en este vehículo ,
00371 //1d - Sensores de oxígenos presentes en el banco 4
00372 //1e - Estado de las entradas auxiliares
00373 //1f - Tiempo desde que se puso en marcha el motor , 0 , 65,535 , sec , 256A+B
00374
00375 //decodeHexToDec
00376
00377 //20 - PID implementados [21 - 40]
00378 //21 - Distancia recorrida con la luz indicadora de falla (Malfunction Indicator Lamp, MIL) encendida , 0 ,
       65,535 , km
00379
00380 //decodeHexToDec
00381
00382 //22 - Presión del tren de combustible, relativa al colector de vacío , 0 , 5177.265 , kPa, 0.079(256A+B)
00383
00384 float decodePresionCombColector(char *response){
00385     int dec = (int)strtol(response, NULL, 16);
00386     return 0.079*dec;
00387 }
00388
00389 //23 - Presión del medidor del tren de combustible (Diesel o inyección directa de gasolina) , 0 , 655,350 ,
       kPa , 10(256A+B)
00390
```

```
00391 float decodePresionMedidorCombustible(char *response){
00392     int dec = (int)strtol(response, NULL, 16);
00393     return 10*dec;
00394 }
00395
00396 //24,25,26,27,28,29,2a,2b - Sensor de oxígeno 1 AB: Relación equivalente de combustible - aire CD: Voltaje
       , 0 0 , <2 <8 , prop. V , A, B: (256A+B)/32768 C, D: (256C+D)/8192
00397
00398 struct OxigenoResponse decodeRelacionCombAire(char *response){
00399     char ABResponse[5], CDResponse[5];
00400     float AB, CD;
00401     struct OxigenoResponse datos;
00402     //Añade caracter \0 al final de la cadena AResponse
00403     memset(ABResponse, '\0', sizeof(ABResponse));
00404     //Divide los bytes de respuesta
00405     strncpy( ABResponse, response, 4);
00406     strcpy( CDResponse, response + 4 );
00407
00408     AB = (int)strtol(ABResponse, NULL, 16);
00409     datos.A = AB/32768;
00410     CD = (int)strtol(CDResponse, NULL, 16);
00411     datos.B = CD/8192;
00412
00413     return datos;
00414 }
00415
00416 //2c - EGR comandado , 0 , 100 , % , A/2.55
00417
00418 //decodeCargaPosicionEGR
00419
00420 //2d - falla EGR , -100 , 99.2 , % , A/1.28-100
00421
00422 //decodeAjusteCombustibleEGR(
00423
00424 //2e - Purga evaporativa comandada , 0 , 100 , % , A/2.55
00425
00426 //decodeCargaPosicionEGR
00427
00428 //2f - Nivel de entrada del tanque de combustible , 0 , 100 , % , A/2.55
00429
00430 //decodeCargaPosicionEGR
00431
00432 //30 - Cantidad de calentamientos desde que se borraron los fallas , 0 , 255 , cuenta, A
00433
00434 //decodeHexToDec
00435
00436 //31 - Distancia recorrida desde que se borraron los fallas , 0 , 65,535 , km , 256A+B
00437
00438 //decodeHexToDec
00439
00440 //32 - Presión de vapor del sistema evaporativo , -8,192 , 8191.75 , Pa , (256A + B) / 4 - 8192
00441
00442 float decodePresionVapor(char *response){
00443     int dec = (int)strtol(response, NULL, 16);
00444     return (dec/4.0)-8192.0;
00445 }
00446
00447 //33 - Presión barométrica absoluta , 0 , 255 , kPa , A
00448
00449 //decodeHexToDec
00450
00451 //34,35,36,37,38,39,3a,3b - Sensor de oxígeno 8 AB: Relación equivalente de combustible - aire CD: Actual ,
       0 -128 , <2 <128, prop. mA, A, B: (256A+B)/32768 C, D: C+D/256-128
00452
00453 struct OxigenoResponse decodeRelacionCombAireActual(char *
      response){
00454     char ABResponse[5], CResponse[3], DResponse[3];
00455     float AB, C, D;
00456     struct OxigenoResponse datos;
00457     //Añade caracter \0 al final de la cadena AResponse
00458     memset(ABResponse, '\0', sizeof(ABResponse));
00459     memset(CResponse, '\0', sizeof(CResponse));
00460     //Divide los bytes de respuesta
00461     strncpy( ABResponse, response, 4);
00462     strncpy( CResponse, response + 4, 2);
00463     strcpy( DResponse, response + 6);
00464
00465     AB = (int)strtol(ABResponse, NULL, 16);
00466     datos.A = AB/32768;
00467     C = (int)strtol(CResponse, NULL, 16);
00468     D = (int)strtol(DResponse, NULL, 16);
00469     datos.B = C+(D/256)-128;
00470
00471     return datos;
00472 }
00473
00474 //3c,3d,3e,3f - Temperatura del catalizador: Banco 1-4, Sensor , -40 , 6,513.5 , °C , (256A+B)/10-40
```

```
00475
00476 float decodeTempCatalizador(char *response){
00477     int dec = (int)strtol(response, NULL, 16);
00478     return (dec/10.0)-40;
00479 }
00480
00481 //40 - PID implementados [41 - 60]
00482 //41 - Estado de los monitores en este ciclo de manejo
00483 //42 - Voltaje del módulo de control , 0 , 65.535 , V , (256A+B)/1000
00484
00485 float decodeVoltajeControl(char *response){
00486     int dec = (int)strtol(response, NULL, 16);
00487     return dec/1000.0;
00488 }
00489
00490 //43 - Valor absoluta de carga , 0 , 25,700 , % , (256A+B)/2.55
00491
00492 //decodeCargaPosicionEGR
00493
00494 //44 - Relación equivaliente comandada de combustible - aire , 0 , <2 , prop. , (256A+B)/32768
00495
00496 float decodeRelacionCombAireBasica(char *response){
00497     int dec = (int)strtol(response, NULL, 16);
00498     return dec/32768.0;
00499 }
00500
00501 //45 - Posición relativa del acelerador , 0 , 100 , % , A/2.55
00502
00503 //decodeCargaPosicionEGR
00504
00505 //46 - Temperatura del aire ambiental , -40 , 215 , °C , A-40
00506
00507 //decodeTempGeneral
00508
00509 //47,48,49,4a,4b 4c - Posición absoluta del acelerador B,C,D,E,F  Actuador comandando del acelerador , 0 ,
      100 , % , A/2.55
00510
00511 //decodeCargaPosicionEGR
00512
00513 //4d, 4e - Tiempo transcurrido con MIL encendido Tiempo transcurrido desde que se borraron los códigops de
      fallas , 0 , 65,535 , min , 256A+B
00514
00515 /*4f Valor máximo de la relación de equivalencia de combustible - aire,
00516  voltaje del sensor de oxígenos, corriente del sensor de oxígenos
00517   y presión absoluta del colector de entrada , 0 0 0 0 , 255 255 255 2550, prop. V mA kPa, A B C D*10
00518 */
00519
00520 struct RelacionesResponse decodeRelaciones(char *response){
00521     char AResponse[3], BResponse[3], CResponse[3], DResponse[3];
00522     struct RelacionesResponse datos;
00523     //Añade caracter \0 al final de la cadena AResponse
00524     memset(AResponse, '\0', sizeof(AResponse));
00525     memset(BResponse, '\0', sizeof(BResponse));
00526     memset(CResponse, '\0', sizeof(CResponse));
00527     //Divide los bytes de respuesta
00528     strncpy( AResponse, response, 2);
00529     strncpy( BResponse, response + 2, 2);
00530     strncpy( CResponse, response + 4, 2);
00531     strcpy( DResponse, response + 6);
00532
00533     datos.A = (int)strtol(AResponse, NULL, 16);
00534     datos.B = (int)strtol(BResponse, NULL, 16);
00535     datos.C = (int)strtol(CResponse, NULL, 16);
00536     datos.D = 10*(int)strtol(DResponse, NULL, 16);
00537
00538     return datos;
00539 }
```

5.11. 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:

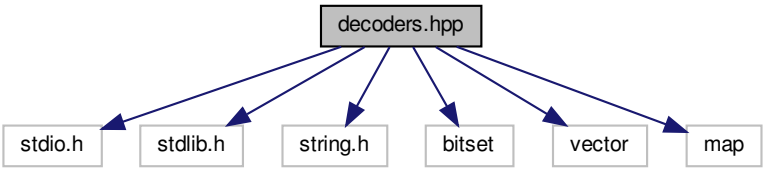
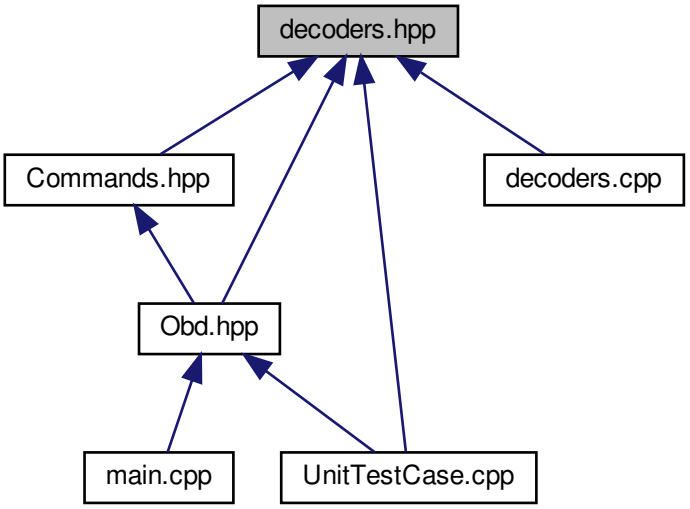


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



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).
- 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.

5.11.1. 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

Definición en el archivo decoders.hpp.

5.11.2. Documentación de los 'defines'

5.11.2.1. 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.

5.11.2.2. 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.

5.11.3. Documentación de las funciones

5.11.3.1. 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 47 del archivo decoders.cpp.

5.11.3.2. 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 284 del archivo decoders.cpp.

5.11.3.3. 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 316 del archivo decoders.cpp.

5.11.3.4. 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 270 del archivo decoders.cpp.

5.11.3.5. 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.

5.11.3.6. 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 86 del archivo decoders.cpp.

5.11.3.7. 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 298 del archivo decoders.cpp.

5.11.3.8. 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 112 del archivo decoders.cpp.

5.11.3.9. 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 384 del archivo decoders.cpp.

5.11.3.10. 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 291 del archivo decoders.cpp.

5.11.3.11. 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 391 del archivo decoders.cpp.

5.11.3.12. 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 442 del archivo decoders.cpp.

5.11.3.13. 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 398 del archivo decoders.cpp.

5.11.3.14. 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 453 del archivo decoders.cpp.

5.11.3.15. 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 496 del archivo decoders.cpp.

5.11.3.16. 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 520 del archivo decoders.cpp.

5.11.3.17. 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 305 del archivo decoders.cpp.

5.11.3.18. 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 341 del archivo decoders.cpp.

5.11.3.19. 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 128 del archivo decoders.cpp.

5.11.3.20. 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 476 del archivo decoders.cpp.

5.11.3.21. 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 277 del archivo decoders.cpp.

5.11.3.22. 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 327 del archivo decoders.cpp.

5.11.3.23. decodeVIN()

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

Función de decodificación del Número de Identificación del Vehículo (VIN).

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.

5.11.3.24. 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 485 del archivo decoders.cpp.

5.12. decoders.hpp

```
00001
00008 #ifndef DECODERS_HPP
00009 #define DECODERS_HPP
00010
00011 #include <stdio.h>
00012 #include <stdlib.h>
00013 #include <string.h>
00014 #include <bitset>
00015 #include <vector>
00016 #include <map>
00017
00018 #define PID_BITS 32
00019 #define STATUS_BITS 8
00025 struct OxigenoResponse {
00026     float A;
00027     float B;
00028 };
00029
00034 struct RelacionesResponse {
00035     int A;
00036     int B;
00037     int C;
00038     int D;
00039 };
00040
00045 void noDecodeAT();
00046
00053 std::map<std::string, std::string> decodeStatus(char *response);
00054
00061 std::vector<int> decodePIDS(char *response);
00062
00069 std::vector<std::string> decodeDTCs(char *response);
00070
00077 std::string convertDTCs(std::string dtc);
00078
00085 std::string decodeVIN(char * response);
00086
00093 std::string decodeDescribeProtocol(char * response);
00094
00101 float decodeCargaPosicionEGR(char *response);
00102
00109 float decodeTempGeneral(char *response);
00110
00117 float decodeAjusteCombustibleEGR(char *response);
00118
00125 float decodePresionCombustible(char *response);
00126
00133 float decodeHexToDec(char *response);
00134
00141 float decodeRPM(char *response);
00142
00149 float decodeAvanceTiempo(char *response);
00150
00157 float decodeVelocidadMAF(char *response);
00158
00165 float decodePresionCombColector(char *response);
00166
00173 float decodePresionMedidorCombustible(char *response);
00174
00181 float decodePresionVapor(char *response);
00182
00189 float decodeTempCatalizador(char *response);
00190
00197 float decodeVoltajeControl(char *response);
00198
00205 float decodeRelacionCombAireBasica(char *response);
00206
00213 struct OxigenoResponse decodeSensorOxigeno(char *response);
00214
00221 struct OxigenoResponse decodeRelacionCombAire(char *response);
00222
00229 struct OxigenoResponse decodeRelacionCombAireActual(char *
      response);
00230
00239 struct RelacionesResponse decodeRelaciones(char *response);
00240
00241
00242 #endif
```

5.13. 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.

5.13.1. 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

Definición en el archivo loadcfg.cpp.

5.13.2. Documentación de las funciones

5.13.2.1. 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 59 del archivo loadcfg.cpp.

5.13.2.2. 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.

5.13.2.3. 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.

5.14. loadcfg.cpp

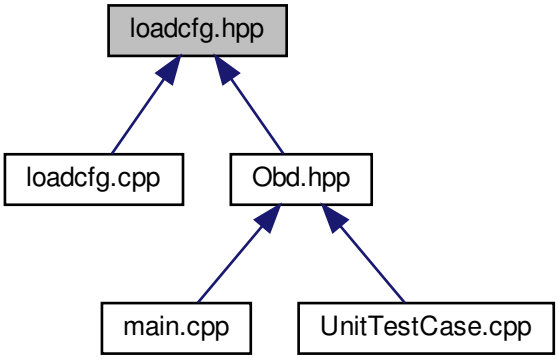
```
00001
00008 #include <stdio.h>
00009 #include <stdlib.h>
00010 #include <string.h>
00011 #include <sys/time.h>
00012 #include <time.h>
00013 #include <sys/types.h>
00014 #include <sys/socket.h>
00015 #include <netinet/in.h>
00016 #include <arpa/inet.h>
00017 #include <netdb.h>
00018 #include <errno.h>
00019 #include <iostream>
00020 #include <fstream>
00021 #include <algorithm>
00022 #include <map>
00023 #include <string>
00024 #include <ifaddrs.h>
00025 #include <ctype.h>
00026 #include <unistd.h>
00027 #include <stdexcept>
00028 #include <sstream>
00029 #include <netpacket/packet.h>
00030 #include "loadcfg.hpp"
00031
00032 void shit (const char* mens)
00033 {
00034 std::cerr « "ABORTING: " « mens « " - " « errno « "\n";
00035 perror("Error is ");
00036 exit(1);
00037 }
00038
00039 void loadCfg (const char* filename, cfgType* pcfg)
00040 {
00041    std::ifstream cFile (filename);
00042    if (cFile.is_open()){
00043      std::string line;
00044      while(getline(cFile, line)){
00045        line.erase(std::remove_if(line.begin(), line.end(), ::isspace), line.end());
00046        if(line[0] == '#' || line.empty())
00047          continue;
00048        int delimiterPos = line.find("=");
00049        std::string name = line.substr(0, delimiterPos).c_str();
00050        std::string value = line.substr(delimiterPos + 1).c_str();
00051        (*pcfg)[name] = value;
00052      }
00053    }
00054    else {
00055      shit("Couldn't open config file for reading.\n");
00056    }
00057 }
00058
00059 std::string getmac (const char* name)
00060 {
00061    int i;
00062    struct ifaddrs *addrs,*tmp;
00063    std::stringstream macaddress;
00064    char mymac[18];
00065    getifaddrs(&addrs);
00066    tmp = addrs;
00067
00068    memset (mymac, 0, 18);
00069    while (tmp) {
00070      if (!strcmp(name, tmp->ifa_name)){
00071           struct sockaddr_ll *s = (struct sockaddr_ll*)tmp->ifa_addr;
00072           for (i=0; i <s->sll_halen; i++){
00073          sprintf(mymac, "%s%02x%c",
00074                                  mymac,
00075                                  (s->sll_addr[i]),
00076                                  (i+1!=s->sll_halen)?':':0);
```

```
00077           }
00078           macaddress « mymac;
00079           return macaddress.str();
00080         }
00081       tmp = tmp->ifa_next;
00082     }
00083     freeifaddrs(addrs);
00084     return std::string("ff:ff:ff:ff:ff:ff");
00085 }
```

5.15. 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.

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.

5.15.1. 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

Definición en el archivo loadcfg.hpp.

5.15.2. Documentación de las funciones

5.15.2.1. 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 59 del archivo loadcfg.cpp.

5.15.2.2. 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.

5.15.2.3. 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.

5.16. loadcfg.hpp

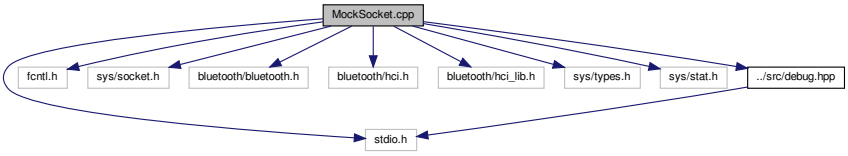
```
00001
00008 #ifndef LOADCFG_HPP
00009 #define LOADCFG_HPP
00010
00015 typedef  std::map <std::string, std::string> cfgType;
00016
00022 void shit (const char* mens);
00023
00030 void loadCfg (const char* filename, cfgType* pcfg);
00031
00038 std::string getmac (const char* name);
00039
00040 #endif
```

5.17. 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"
```
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** ()

5.17.1. Descripción detallada

Archivo que contiene las funciones mock bluetooth para poder realizar las pruebas de integración.

Autor

Sergio Román González

Fecha

05/09/2020

Definición en el archivo MockSocket.cpp.

5.17.2. Documentación de las funciones

5.17.2.1. 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 25 del archivo MockSocket.cpp.

5.18. MockSocket.cpp

```
00001
00009 #include <stdio.h>
00010 #include <fcntl.h>
00011 #include <sys/socket.h>
00012 #include <bluetooth/bluetooth.h>
00013 #include <bluetooth/hci.h>
00014 #include <bluetooth/hci_lib.h>
00015 //Para mkfifo
00016 #include <sys/types.h>
00017 #include <sys/stat.h>
00018 #include "../src/debug.hpp"
00019
00025 std::string findDevPTS(){
00026     system("ls /dev/pts | tail -2 | head -1 > tmpPTSfile.txt");
00027
00028     int ultPts;
00029     int tempVar;
00030     std::ifstream input_file("tmpPTSfile.txt");
00031     while ( input_file » tempVar )
00032     {
00033         ultPts = tempVar;
00034     }
00035
00036     std::string devFile = "/dev/pts/" + std::to_string(ultPts);
00037     remove("tmpPTSfile.txt");
00038
00039     return devFile;
00040 }
00041
00042 int mock_socket(int domain, int type, int protocol){
00043
00044     std::string devFile = findDevPTS();
00045
00046     int filedesc = open(devFile.c_str(), O_RDWR);
00047     if (filedesc < 0) {
00048         debugError("Error al abrir socket %d.", filedesc);
00049     }
00050
00051     return filedesc;
00052 }
00053
00054 int hci_get_route( bdaddr_t *bdaddr ){
00055     int value = 0;
00056     return value;
00057 }
00058
00059 int hci_open_dev( int dev_id ){
00060     int filedesc = open("/dev/null", O_RDONLY);
00061     if(filedesc < 0)
00062         filedesc = -1;
00063     return filedesc;
00064 }
00065
00066 int hci_inquiry(int dev_id, int len, int max_rsp, const uint8_t *lap, inquiry_info **ii, long flags){
00067     int value = 1;
00068     return value;
00069 }
00070
00071
00072 int hci_read_remote_name(int sock, const bdaddr_t *ba, int len, char *name, int timeout){
00073     int value = 1;
00074     strcpy(name, "OBDII");
00075     return value;
00076 }
00077
00078 int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen){
00079     int value = 0;
00080     return value;
00081 }
00082
00083 void writeSocket(){
00084
00085 }
```

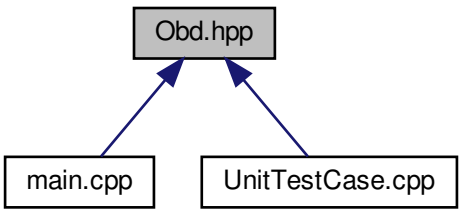
5.19. Referencia del Archivo Obd.hpp

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.

5.19.1. 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

Definición en el archivo Obd.hpp.

5.19.2. Documentación de los 'defines'

5.19.2.1. 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.

5.20. Obd.hpp

```
00001
00008 #ifndef OBD_HPP
00009 #define OBD_HPP
00010
00011 #include <iostream>
00012 #include <fstream>
00013 #include <thread>
00014
00015 #include <bitset>
00016 #include <vector>
00017 #include <sstream>
00018 #include <algorithm>
00019
00020 #include <utility>
00021 #include <map>
00022
00023 #include <typeinfo>
00024
00025 #include <netinet/in.h>
00026 #include <arpa/inet.h>
00027 #include <sys/types.h>
00028 #include <sys/socket.h>
00029 #include <sys/epoll.h>
00030
00031 #include <bluetooth/bluetooth.h>
00032 #include <bluetooth/rfcomm.h>
00033 #include <bluetooth/hci.h>
00034 #include <bluetooth/hci_lib.h>
00035
00036 #include <unistd.h>
00037
00038 #include <ctime>
00039
00040 #include "Commands.hpp"
00041 #include "decoders.hpp"
00042 #include "loadcfg.hpp"
00043 #include "debug.hpp"
00044
00045 #ifdef TEST
00046     #define socket mock_socket
00047     #include "../test/MockSocket.cpp"
00048 #endif
00049
00050
00051
00052 #define MAX_EP_EVTS 20
00058 using json = nlohmann::json;
00059
00064 typedef std::pair<std::string, Commands> tupla;
00065
00073 class Obd {
00074 public:
00075     std::map<std::string, Commands> map_commands;
00083     Obd(const char *deviceName){
00084         // Comenzamos el descubrimiento del dipositivo Bluetooth
00085         debugLog("Iniciando descubrimiento del dispositivo %s", deviceName);
00086         this->discoverDeviceAddress(deviceName, this->dest);
00087         if(this->m_deviceFound){
00088             // Si lo encontramos nos conectamos
00089             this->connectBluetooth();
00090             if (this->m_status){
00091                 // Si la conexión tiene éxito, iniciamos los decodificadores
00092                 this->initDecoderFunctions();
00093                 // Leemos el archivo de PIDS
00094                 this->readFileData();
00095                 // Comenzamos el envío de mensajes de inicio
00096                 this->initMessages();
00097             }
00098         } else {
00099             debugLog("Dispositivo %s no encontrado.", deviceName);
00100         }
00101     }
00102
00112     void discoverDeviceAddress(const char * deviceName, char *deviceAddress){
00113         inquiry_info *ii = NULL;
00114         int max_rsp, num_rsp;
00115         int dev_id, sock, len, flags;
00116         int i;
00117         char addr[19] = { 0 };
00118         char name[248] = { 0 };
00119
00120         //Identificamos la interfaz bluetooth del dispositivo
00121         dev_id = hci_get_route(NULL);
00122         //Abrimos socket para esta interfaz
```

```
00123         sock = hci_open_dev( dev_id );
00124         if (dev_id < 0 || sock < 0) {
00125             perror("Abriendo socket");
00126             exit(1);
00127         }
00128 
00129         len  = 8;
00130         max_rsp = 255;
00131         flags = IREQ_CACHE_FLUSH;
00132         ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));
00133 
00134         //Iniciamos el descubrimiento de dispositivos bluetooth
00135         num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags);
00136         if( num_rsp < 0 ) perror("hci_inquiry");
00137 
00138         //Entre todas las respuestas buscamos el dispositivo bluetooth de OBDII
00139         for (i = 0; i < num_rsp; i++) {
00140             ba2str(&(ii+i)->bdaddr, addr);
00141             memset(name, 0, sizeof(name));
00142             if (hci_read_remote_name(sock, &(ii+i)->bdaddr, sizeof(name), name, 0) < 0)
00143                 strcpy(name, "[unknown]");
00144             debugLog("%s  %s", addr, name);
00145             //Si la cadena introducida a la función es igual al dispositivo encontrado guardamos la
    dirección
00146             if(strcmp(deviceName, name) == 0){
00147                 this->m_deviceFound = true;
00148                 strcpy(deviceAddress, addr);
00149                 debugLog("Dispositivo %s encontrado", deviceName);
00150                 break;
00151             }
00152         }
00153 
00154         free( ii );
00155         close( sock );
00156     }
00157 
00167     void connectBluetooth(){
00168         try{
00169             struct sockaddr_rc addr;
00170             int statusConnection;
00171 
00172             // Abrimos socket bluetooh
00173             this->m_cli_s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
00174 
00175             debugLog("socket: %d", this->m_cli_s);
00176             if (this->m_cli_s < 0) {
00177                 throw std::string("error abriendo socket BT/RFCOMM");
00178             }
00179 
00180             addr.rc_family = AF_BLUETOOTH;
00181             str2ba(this->dest, &addr.rc_bdaddr );
00182             addr.rc_channel = (uint8_t) 1;
00183 
00184             debugLog("Conectando con %s (canal %d)", this->dest, addr.rc_channel);
00185             //Iniciamos la conexión
00186             statusConnection = connect(this->m_cli_s, (struct sockaddr *)&addr, sizeof(addr));
00187 
00188             if (statusConnection) {
00189                 close(this->m_cli_s);
00190                 throw std::string("No se ha podido conectar");
00191             }
00192 
00193             debugLog("Conectado!");
00194             this->m_status = true;
00195 
00196             //Creamos instancia epoll para la recepción de datos en el socket
00197             this->epoll_fd = epoll_create(1);
00198             if (this->epoll_fd < 0) {
00199                 perror("No se ha podido crear epoll");
00200                 close(this->m_cli_s);
00201             }
00202 
00203             this->ev.events = EPOLLIN;
00204             this->ev.data.fd = this->m_cli_s;
00205 
00206             //Añadimos el socket de conexión a la instancia de epoll creada
00207             int err = epoll_ctl(this->epoll_fd, EPOLL_CTL_ADD, this->m_cli_s, &ev);
00208 
00209             if (err) {
00210                 perror("No se ha podido añadir el socket cliente a la instancia epoll");
00211                 close(this->m_cli_s);
00212                 close(this->epoll_fd);
00213             }
00214 
00215 
00216         } catch(std::string e) {
00217             std::cerr « e « std::endl;
```

```
00218         }
00219     }
00220
00228     void readFileData(){
00229         std::ifstream ifs("data/PIDS.json");
00230         auto j = json::parse(ifs);
00231
00232         //Convertimos todos los PIDS en objetos del tipo Commands y los añadimos a Obd
00233         for (int i = 0; i < (int)j.size(); ++i)
00234         {
00235             this->map_commands.insert(tupla(j[i]["name"], Commands(j[i])));
00236         }
00237     }
00238
00248     void send(Commands command){
00249         //Iniciamos en un hilo de ejecución la función polling de recepción de datos
00250         std::thread t1(&Obd::polling, this, command);
00251
00252         char *p;
00253         char buf[1024];
00254         int len;
00255
00256         //Comando a enviar
00257         std::string message = command.getCMD();
00258         strcpy(buf, message.c_str());
00259
00260         len = strlen(buf);
00261         buf[len] = '\n';
00262         buf[len+1] = '\0';
00263
00264         // Todo los mensajes a ELM327  deben terminar con el caracter retorno de carro (hex  '0D', \r).
00265         p = buf;
00266         while (*p) {
00267             if (*p == '\n')
00268                 *p = '\r';
00269             p++;
00270         }
00271
00272         debugLog("Mensaje a enviar: %s", buf);
00273         debugLog("Enviando mensaje...");
00274         write(this->m_cli_s, buf, strlen(buf));
00275
00276         //Queda a la espera de finalización de ejecución del hilo de recepción del mensaje OBD
00277         t1.join();
00278     }
00279
00292     void polling(Commands command){
00293         struct epoll_event events[MAX_EP_EVTS];
00294         int nfds;
00295         bool continuar = true;
00296
00297         debugLog("Polling function");
00298
00299         // Bucle infinito para el envío de datos por bluetooth al conector OBD
00300         while(continuar) {
00301         // Buffer para enviar y recibir
00302             char message_rcv[1024], buf[1024], *p;
00303             ssize_t len;
00304             int i;
00305             //Quedamos a la espera de recepción de eventos en la instancia epoll (socket)
00306             nfds = epoll_wait(this->epoll_fd, events, MAX_EP_EVTS, -1);
00307             if (nfds < 0) {
00308                 perror("epoll error");
00309                 break;
00310             }
00311             for (i = 0; i < nfds; i++) {
00312                 if ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP)) {
00313                     debugError("epoll error");
00314                 }
00315                 //Si los eventos detectados corresponden al socket de conexión con el vehículo, tratamos el
    mensaje
00316                 if (events[i].data.fd == this->m_cli_s) {
00317                     len = read(this->m_cli_s, &buf, sizeof(buf) - 1);
00318                     if (len < 0) {
00319                         perror("socket read error");
00320                         continue;
00321                     }
00322                     debugLog("Evento leído: %s", buf);
00323                     strcat(message_rcv, buf);
00324                     //Si se detecta el caracter ">" se ha finalizado el mensaje
00325                     if(strstr(buf, ">") != NULL) {
00326                         len = strlen(message_rcv);
00327                         message_rcv[len] = '\0';
00328
00329                         p = message_rcv;
00330                         //Conversión inversa del mensaje ELM327 enviado en el último carácter
00331                         while(*p) {
```

```
00332                             if (*p == '\r')
00333                                 *p = '\n';
00334                             p++;
00335                         }
00336                         //Transformar respuesta
00337                         debugLog("Mensaje recibido:\n%s", message_rcv);
00338 
00339                         char * ocurrencia = message_rcv;
00340                         if((ocurrencia=strstr(ocurrencia, command.getCMDResponse().c_str()))
      != NULL){
00341                             while((ocurrencia=strstr(ocurrencia, command.
      getCMDResponse().c_str())) != NULL){
00342                                 debugLog("Ocurrencia encontrada");
00343                                 char info[1024];
00344                                 memset(info, '\0', sizeof(info));
00345                                 strncpy(info, ocurrencia + command.getCMD().size() , command.
      getBytesResponse());
00346                                 debugLog("Información: %s", info);
00347                                 std::string type_data = command.getTypeData();
00348                                 //Dependiendo del tipo de dato de la respuesta se busca el decodificador
       correspondiente
00349                                 if (!type_data.compare("float")){
00350                                     auto varResultado = this->decoderFunctionsFloat[command.
      getDecoder().c_str()](info);
00351                                     std::cout << command.getName() << " - " << command.
      getDescription() << " - Min=" << command.getMIN() << " Max=" << command.
      getMAX() << std::endl;
00352                                     std::cout << "-> " << varResultado << " "<< command.
      getUnits() << std::endl;
00353                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00354                                 } else if(!type_data.compare("OxigenoResponse")){
00355                                     auto varResultado = this->decoderFunctionsStructOx[command.
      getDecoder().c_str()](info);
00356                                     std::cout << command.getName() << " - " << command.
      getDescription() << " - Min=" << command.getMIN() << " Max=" << command.
      getMAX() << std::endl;
00357                                     std::cout << "-> " << varResultado.A << "/" << varResultado.B << " "<<
      command.getUnits() << std::endl;
00358                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00359 
00360                                 } else if (!type_data.compare("RelacionesResponse")) {
00361                                     auto varResultado = this->decoderFunctionsStructRel[command.
      getDecoder().c_str()](info);
00362                                     std::cout << command.getName() << " - " << command.
      getDescription() << " - Min=" << command.getMIN() << " Max=" << command.
      getMAX() << std::endl;
00363                                     std::cout << "-> " << varResultado.A << "/" << varResultado.B << "/" <<
       varResultado.C << "/" << varResultado.D << " "<< command.getUnits() << std::endl;
00364                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00365 
00366                                 } else if(!type_data.compare("vectorInt")){
00367                                     auto varResultado = this->decoderFunctionsVectorInt[command.
      getDecoder().c_str()](info);
00368                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00369 
00370                                     std::cout << command.getName() << " - " << command.
      getDescription() << " - Min=" << command.getMIN() << " Max=" << command.
      getMAX() << std::endl;
00371                                     //Tratamiento para los PIDs disponibles
00372                                     for (uint32_t i = 0; i < varResultado.size(); ++i){
00373                                         std::string substr_cmd = command.getCMD().substr(2,2);
00374                                         int sum_pid = stoi(substr_cmd,nullptr,16);
00375                                         std::stringstream stream;
00376                                         stream << std::hex << sum_pid+varResultado[i];
00377                                         std::string result(stream.str());
00378                                         if(result.size() == 1)
00379                                         //Si el resultado solo tiene un caracter se añade un 0 al principio
00380                                             result.insert(0,"0");
00381                                         result.insert(0,"01");
00382                                         std::transform(result.begin(), result.end(),result.begin(),
      ::toupper);
00383 
00384                                         //Almacenamos el resultado de los PIDs disponibles
00385                                         this->vecPIDs.push_back(result);
00386                                     }
00387                                 } else if (!type_data.compare("vectorStr")) {
00388                                     auto varResultado = this->decoderFunctionsVectorStr[command.
      getDecoder().c_str()](info);
00389                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00390                                     //Decodificador para DTC
00391                                     if (varResultado.empty()){
00392                                         debugLog("No hay DTC en el vehículo");
```

```
00393                                     } else {
00394                                         this->vecDTCs = varResultado;
00395                                         for (uint32_t i = 0; i < varResultado.size(); ++i)
00396                                         {
00397                                             debugLog("Enviar DTC: %s", varResultado[i].c_str());
00398                                         }
00399                                     }
00400                                 } else if (!type_data.compare("string")) {
00401                                     auto varResultado = this->decoderFunctionsStr[command.
      getDecoder().c_str()](info);
00402                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00403                                     //Decodificador para el número de identificación del vehículo
00404                                     if(!command.getDecoder().compare("decodeVIN"))
00405                                         this->vin.append(varResultado);
00406                                     std::cout « command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00407                                 } else if (!type_data.compare("map")) {
00408                                     auto varResultado = this->decoderFunctionsMap[command.
      getDecoder().c_str()](info);
00409                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00410                                     this->mapStatus = varResultado;
00411                                 } else {
00412                                     debugLog("Tipo de dato no reconocido");
00413                                 }
00414                                 ocurrencia++;
00415                             }
00416                             std::cout « "------------------------------------------------------------" <
      < std::endl;
00417                             memset(message_rcv, '\0', sizeof(message_rcv));
00418                             continuar = false;
00419                             //Respuestas de mensajes de AT de configuración
00420                         } else if((strstr(message_rcv, "OK")) != NULL){
00421                             debugLog("%s = OK.", command.getDescription().c_str());
00422                             memset(message_rcv, '\0', sizeof(message_rcv));
00423                             continuar = false;
00424                             //Vehículo sin el dato solicitado
00425                         } else if((strstr(message_rcv, "NO DATA")) != NULL){
00426                             debugLog("%s = No disponible.", command.
      getDescription().c_str());
00427                             memset(message_rcv, '\0', sizeof(message_rcv));
00428                             continuar = false;
00429                         } else {
00430                             //Para conocer el protocolo actual
00431                             if(!command.getName().compare("DESCRIBE_PROTOCOL")){
00432                                 char info[1024];
00433                                 char* token = strtok(message_rcv, "\n");
00434                                 strcpy(info, token);
00435                                 auto varResultado = this->decoderFunctionsStr[command.
      getDecoder().c_str()](info);
00436                                 this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00437                                 this->currentProtocol = varResultado;
00438                             }else if(!command.getName().compare("DESCRIBE_PROTOCOL_NUMBER")){
00439                                 char info[1024];
00440                                 char* token = strtok(message_rcv, "\n");
00441                                 strcpy(info, token);
00442                                 auto varResultado = this->decoderFunctionsStr[command.
      getDecoder().c_str()](info);
00443                                 this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00444                                 this->currentProtocolNumber = varResultado;
00445                             } else {
00446                                 debugLog("Mensaje recibido no entendido!");
00447                             }
00448                             memset(message_rcv, '\0', sizeof(message_rcv));
00449                             continuar = false;
00450                         }
00451                     }
00452
00453                     memset(buf, '\0', sizeof(buf));
00454                 } else {
00455                     debugError("Evento desconocido");
00456                 }
00457             }
00458         }
00459     }
00460
00472     void initMessages(){
00473         //Inicialización de la conexión con ELM327
00474         std::map<std::string, std::string> listPIDs = {
00475             {"PIDS_B", "0120"},
00476             {"PIDS_C", "0140"},
00477             {"PIDS_D", "0160"},
00478             {"PIDS_E", "0180"},
```

```
00479             {"PIDS_F", "01A0"},
00480             {"PIDS_G", "01C0"}
00481         };
00482         this->send(this->map_commands.find("RESET")->second);
00483         this->send(this->map_commands.find("DEFAULT_VALUES")->second);
00484         this->send(this->map_commands.find("RESP_SIN_ESPACIOS")->second);
00485         this->send(this->map_commands.find("SIN_ECO")->second);
00486         this->send(this->map_commands.find("SIN_HEADER")->second);
00487         this->send(this->map_commands.find("AUTO_PROTO")->second);
00488         this->send(this->map_commands.find("STATUS")->second);
00489         this->send(this->map_commands.find("GET_VIN")->second);
00490         this->send(this->map_commands.find("PIDS_A")->second);
00491         //Bucle para detectar PIDs disponibles
00492         for (std::map<std::string, std::string>::iterator it=listPIDs.begin(); it!=listPIDs.end(); ++it){
00493             if(this->existPID(it->second)){
00494                 this->send(this->map_commands.find(it->first)->second);
00495             }
00496         }
00497         debugLog("N° de comandos disponibles = %zu", vecPIDs.size());
00498     }
00499
00507     void initDecoderFunctions(){
00508     //Inicia las funciones dependiendo del tipo de dato de respuesta
00509     this->decoderFunctionsFloat = {
00510             { "decodeCargaPosicionEGR", decodeCargaPosicionEGR},
00511             { "decodeTempGeneral", decodeTempGeneral},
00512             { "decodeAjusteCombustibleEGR", decodeAjusteCombustibleEGR},
00513             { "decodePresionCombustible", decodePresionCombustible},
00514             { "decodeHexToDec", decodeHexToDec},
00515             { "decodeRPM", decodeRPM},
00516             { "decodeAvanceTiempo", decodeAvanceTiempo},
00517             { "decodeVelocidadMAF", decodeVelocidadMAF},
00518             { "decodePresionCombColector", decodePresionCombColector},
00519             { "decodePresionMedidorCombustible", decodePresionMedidorCombustible
      },
00520             { "decodePresionVapor", decodePresionVapor},
00521             { "decodeTempCatalizador", decodeTempCatalizador},
00522             { "decodeVoltajeControl", decodeVoltajeControl},
00523             { "decodeRelacionCombAireBasica", decodeRelacionCombAireBasica}
00524         };
00525         this->decoderFunctionsStructOx = {
00526             { "decodeSensorOxigeno", decodeSensorOxigeno},
00527             { "decodeRelacionCombAire", decodeRelacionCombAire},
00528             { "decodeRelacionCombAireActual", decodeRelacionCombAireActual}
00529         };
00530
00531         this->decoderFunctionsStructRel["decodeRelaciones"] = decodeRelaciones;
00532         this->decoderFunctionsVectorInt["decodePIDS"] = decodePIDS;
00533         this->decoderFunctionsVectorStr["decodeDTCs"] = decodeDTCs;
00534         this->decoderFunctionsStr = {
00535             {"decodeVIN", decodeVIN},
00536             {"decodeDescribeProtocol", decodeDescribeProtocol}
00537         };
00538         this->decoderFunctionsMap["decodeStatus"] = decodeStatus;
00539         this->noDecodeFunctionAT["noDecodeAT"] = noDecodeAT;
00540     }
00541
00548     void disconnectBluetooth(){
00549         debugLog("Desconectando dispositivo Bluetooth");
00550         close(this->m_cli_s);
00551         close(this->epoll_fd);
00552     }
00553
00560     bool existPID(std::string command){
00561         bool exists = false;
00562         //Comprueba en la lista de PIDs que el comando este implementado
00563         for (uint32_t i = 0; i < this->vecPIDs.size(); ++i){
00564             if(!this->vecPIDs[i].compare(command)){
00565                 exists = true;
00566                 break;
00567             }
00568         }
00569         return exists;
00570     }
00571
00579     void printPIDs(){
00580         //Iteración para imprimir por consola los PIDs disponibles
00581         for (std::map<std::string, Commands>::iterator it=this->map_commands.begin(); it!=this->
      map_commands.end(); ++it){
00582             Commands command = it->second;
00583             std::string str_cmd = command.getCMD();
00584             for (uint32_t i = 0; i < this->vecPIDs.size(); ++i){
00585                 if(!str_cmd.compare(this->vecPIDs[i])){
00586                     std::cout « str_cmd « " - "« command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00587                     break;
```

```
00588                 }
00589             }
00590         }
00591     }
00592
00599     void printStatus(){
00600         //Imprime por consola los resultados de las pruebas del comando STATUS
00601         for (std::map<std::string, std::string>::iterator it=this->mapStatus.begin(); it!=this->mapStatus.
      end(); ++it){
00602             std::cout « it->first « " -> " « it->second « std::endl;
00603         }
00604     }
00605
00611     std::string getVIN(){
00612         //Devuelve el número de identificación del vehículo
00613         return this->vin;
00614     }
00615
00624     std::vector<std::string> getDTCs(){
00625         time_t curr_time;
00626         tm *curr_tm;
00627         char date_string[100];
00628         char time_string[100];
00629         time(&curr_time);
00630
00631         curr_tm = localtime(&curr_time);
00632         strftime(date_string, 50, "%d/%m/%Y", curr_tm);
00633         strftime(time_string, 50, "%T", curr_tm);
00634         std::cout « date_string « " " « time_string « std::endl;
00635
00636         //Consulta de DTC del vehículo
00637         this->send(this->map_commands.find("STATUS")->second);
00638         if (this->mapStatus["DTC_CNT"].compare("0")){
00639             this->send(this->map_commands.find("GET_DTC")->second);
00640         } else {
00641             std::cout « "No hay DTC disponibles" « std::endl;
00642         }
00643         return this->vecDTCs;
00644     }
00645
00651     bool isValid(){
00652         //Bool del estado de la conexión bluetooth
00653         return this->m_status;
00654     }
00655 private:
00656     // Atributos privados de la clase "Obd"
00657     std::vector<std::string> vecPIDs;
00658     std::vector<std::string> vecDTCs;
00659     std::string vin;
00660     std::string currentProtocol;
00661     std::string currentProtocolNumber;
00662     std::map<std::string, std::string> mapStatus;
00664     std::map<std::string, std::function<void()» noDecodeFunctionAT;
00665     std::map<std::string, std::function<float(char *)» decoderFunctionsFloat;
00666     std::map<std::string, std::function<struct OxigenoResponse(char *)» decoderFunctionsStructOx;
00667     std::map<std::string, std::function<struct RelacionesResponse(char *)» decoderFunctionsStructRel;
00668     std::map<std::string, std::function<std::vector<int>(char *)» decoderFunctionsVectorInt;
00669     std::map<std::string, std::function<std::vector<std::string>(char *)» decoderFunctionsVectorStr;
00670     std::map<std::string, std::function<std::string(char *)» decoderFunctionsStr;
00671     std::map<std::string, std::function<std::map<std::string, std::string>(char *)» decoderFunctionsMap;
00673     char dest[19] = { 0 };
00674     int m_cli_s;
00675     bool m_deviceFound = false;
00676     bool m_status = false;
00677     int epoll_fd;
00678     struct epoll_event ev;
00679 };
00680
00681
00682 #endif
```

5.21. 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 15

Funciones

- void getMinicomCMD (char *cmd)

  Función que obtiene el comando Minicom para la primera conexión con el simulador OBDSIM.
- void initOBDSIM ()

  Función de inicialización del simulador OBDSIM.
- void closeOBDSIM ()

  Función de finalización del simulador OBDSIM.
- TEST_CASE ("Test OBD class", "[OBD]")

  Prueba de integración para el funcionamiento general del sistema.
- 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)", "[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.

5.21.1. Descripción detallada

Archivo que contiene el conjunto de pruebas unitarias y de integración del sistema.

Autor

Sergio Román González

Fecha

05/09/2020

Definición en el archivo UnitTestCase.cpp.

5.21.2. Documentación de los 'defines'

5.21.2.1. 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.

5.21.2.2. 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.

5.21.2.3. WAIT_OBDSIM

```
#define WAIT_OBDSIM 15
```

Macro con el tiempo de espera del simulador OBDSIM para introducir valores

Definición en la línea 16 del archivo UnitTestCase.cpp.

5.21.3. Documentación de las funciones

5.21.3.1. 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 26 del archivo UnitTestCase.cpp.

5.21.3.2. initOBDSIM()

```
void initOBDSIM ( )
```

Función de inicialización del simulador OBDSIM.

Inicializa el simulador con entorno gráfico permitiendo la introducción de DTC y otros valores para las pruebas.

Definición en la línea 40 del archivo UnitTestCase.cpp.

5.22. UnitTestCase.cpp

```
00001
00008 // Let Catch provide main():
00009 #define CATCH_CONFIG_MAIN
00011 #include "../src/external/catch.hpp"
00012 #include "../src/decoders.hpp"
00013 #include "../src/Obd.hpp"
00014
00015 #define BUFSIZE 30
00016 #define WAIT_OBDSIM 15
00019 using namespace Catch::literals;
00020
00026 void getMinicomCMD(char * cmd){
00027
00028     std::string devFile = findDevPTS();
00029
00030     snprintf(cmd, BUFSIZE, "minicom -p %s &", devFile.c_str());
00031 }
00032
00040 void initOBDSIM(){
00041
00042     char cmd[BUFSIZE];
00043
00044     system("obdsim -g gui_fltk &");
00045     // 5 segundos para configurar parámetros para el test
00046     sleep(WAIT_OBDSIM);
00047
00048     getMinicomCMD(cmd);
00049
00050     // Se abre terminal en el test con minicom, porque el primer mensaje obdsim no envía >
00051     // a nivel de código, pero si con un terminal con minicom
00052     //system("minicom -p /dev/pts/3 &");
00053     system(cmd);
00054
00055     sleep(1);
00056
00057     // Se mata el proceso minicom, ya que, no es necesario
00058     system("pkill minicom");
00059
00060     sleep(1);
00061 }
00062
00067 void closeOBDSIM(){
00068     system("pkill obdsim");
00069 }
00070
00075 TEST_CASE( "Test OBD class", "[OBD]" ) {
00076
00077     // Iniciamos el simulador OBDSIM para las pruebas
00078     initOBDSIM();
00079
00080
00081     Obd connection = Obd("OBDII");
00082
00083     REQUIRE (connection.isValid() == true);
00084
00085
00086     connection.getDTCs();
00087     connection.printPIDs();
00088
00089     connection.send(connection.map_commands.find("SPEED")->second);
00090
00091     std::cout « connection.map_commands.find("SPEED")->second.getJson().dump(4) « std::endl;
00092
00093     std::cout « connection.map_commands.find("GET_DTC")->second.getJson().dump(4) « std::endl
    ;
00094
00095     closeOBDSIM();
00096 }
00097
00102 TEST_CASE( "Test Revoluciones Por Minuto", "[decoders]" ) {
00103     REQUIRE( decodeRPM((char *)"0000") == 0);
00104     REQUIRE( decodeRPM((char *)"FFFF") == Approx(16383.75).epsilon(0.01));
00105     REQUIRE( decodeRPM((char *)"7FFF") == Approx(8191.75).epsilon(0.01));
00106     REQUIRE( decodeRPM((char *)"12F2") == Approx(1212.5).epsilon(0.01));
00107 }
00108
00113 TEST_CASE( "Test Posición EGR", "[decoders]" ) {
00114     REQUIRE( decodeCargaPosicionEGR((char *)"00") == Approx(0).epsilon(0.01));
00115     REQUIRE( decodeCargaPosicionEGR((char *)"FF") == Approx(100).epsilon(0.01));
00116     REQUIRE( decodeCargaPosicionEGR((char *)"7F") == Approx(50).epsilon(0.01));
00117 }
00118
00123 TEST_CASE( "Test Temperatura General", "[decoders]" ) {
00124     REQUIRE( decodeTempGeneral((char *)"00") == Approx(-40).epsilon(0.01));
```

```
00125     REQUIRE( decodeTempGeneral((char *)"FF") == Approx(215).epsilon(0.01));
00126     REQUIRE( decodeTempGeneral((char *)"7F") == Approx(87).epsilon(0.01));
00127 }
00128
00133 TEST_CASE( "Test Ajuste Combustible EGR", "[decoders]" ) {
00134     REQUIRE( decodeAjusteCombustibleEGR((char *)"00") == Approx(-100).epsilon(0.0
      1));
00135     REQUIRE( decodeAjusteCombustibleEGR((char *)"FF") == Approx(99.2).epsilon(0.0
      1));
00136     REQUIRE( decodeAjusteCombustibleEGR((char *)"7F") == Approx(-0.78125).epsilon
      (0.01));
00137 }
00138
00143 TEST_CASE( "Test Presión Combustible", "[decoders]" ) {
00144     REQUIRE( decodePresionCombustible((char *)"00") == Approx(0).epsilon(0.01));
00145     REQUIRE( decodePresionCombustible((char *)"FF") == Approx(765).epsilon(0.01));
00146     REQUIRE( decodePresionCombustible((char *)"7F") == Approx(381).epsilon(0.01));
00147 }
00148
00153 TEST_CASE( "Test Hexadecimal a Decimal", "[decoders]" ) {
00154     REQUIRE( decodeHexToDec((char *)"00") == Approx(0).epsilon(0.01));
00155     REQUIRE( decodeHexToDec((char *)"FF") == Approx(255).epsilon(0.01));
00156     REQUIRE( decodeHexToDec((char *)"7F") == Approx(127).epsilon(0.01));
00157 }
00158
00163 TEST_CASE( "Test Avance Tiempo", "[decoders]" ) {
00164     REQUIRE( decodeAvanceTiempo((char *)"00") == Approx(-64).epsilon(0.01));
00165     REQUIRE( decodeAvanceTiempo((char *)"FF") == Approx(63.5).epsilon(0.01));
00166     REQUIRE( decodeAvanceTiempo((char *)"7F") == Approx(-0.5).epsilon(0.01));
00167 }
00168
00173 TEST_CASE( "Test Velocidad Flujo Aire MAF", "[decoders]" ) {
00174     REQUIRE( decodeVelocidadMAF((char *)"0000") == Approx(0).epsilon(0.01));
00175     REQUIRE( decodeVelocidadMAF((char *)"FFFF") == Approx(655.35).epsilon(0.01));
00176     REQUIRE( decodeVelocidadMAF((char *)"7FFF") == Approx(327.67).epsilon(0.01));
00177 }
00178
00183 TEST_CASE( "Test Presión del Combustible, relativa al colector de vacío", "[decoders]" ) {
00184     REQUIRE( decodePresionCombColector((char *)"0000") == Approx(0).epsilon(0.01))
      ;
00185     REQUIRE( decodePresionCombColector((char *)"FFFF") == Approx(5177.265).epsilon
      (0.01));
00186     REQUIRE( decodePresionCombColector((char *)"7FFF") == Approx(2588.593).epsilon
      (0.01));
00187 }
00188
00193 TEST_CASE( "Test Presión del Combustible (Diesel o inyección directa de gasolina)", "[decoders]" )
       {
00194     REQUIRE( decodePresionMedidorCombustible((char *)"0000") == Approx(0).
      epsilon(0.01));
00195     REQUIRE( decodePresionMedidorCombustible((char *)"FFFF") == Approx(65535
      0).epsilon(0.01));
00196     REQUIRE( decodePresionMedidorCombustible((char *)"7FFF") == Approx(32767
      0).epsilon(0.01));
00197 }
00198
00203 TEST_CASE( "Test Presión de Vapor del Sistema Evaporativo", "[decoders]" ) {
00204     REQUIRE( decodePresionVapor((char *)"0000") == Approx(-8192).epsilon(0.01));
00205     REQUIRE( decodePresionVapor((char *)"FFFF") == Approx(8191.75).epsilon(0.01));
00206     REQUIRE( decodePresionVapor((char *)"7FFF") == Approx(-0.25).epsilon(0.01));
00207 }
00208
00213 TEST_CASE( "Test Temperatura del Catalizador", "[decoders]" ) {
00214     REQUIRE( decodeTempCatalizador((char *)"0000") == Approx(-40).epsilon(0.01));
00215     REQUIRE( decodeTempCatalizador((char *)"FFFF") == Approx(6513.5).epsilon(0.01));
00216     REQUIRE( decodeTempCatalizador((char *)"7FFF") == Approx(3236.7).epsilon(0.01));
00217 }
00218
00223 TEST_CASE( "Test Voltaje del Módulo de Control", "[decoders]" ) {
00224     REQUIRE( decodeVoltajeControl((char *)"0000") == Approx(0).epsilon(0.01));
00225     REQUIRE( decodeVoltajeControl((char *)"FFFF") == Approx(65.535).epsilon(0.01));
00226     REQUIRE( decodeVoltajeControl((char *)"7FFF") == Approx(32.767).epsilon(0.01));
00227 }
00228
00233 TEST_CASE( "Test Relación Equivaliente Comandada de Combustible", "[decoders]" ) {
00234     REQUIRE( decodeRelacionCombAireBasica((char *)"0000") == Approx(0).epsilon(
      0.01));
00235     REQUIRE( decodeRelacionCombAireBasica((char *)"FFFF") == Approx(2).epsilon(
      0.01));
00236     REQUIRE( decodeRelacionCombAireBasica((char *)"7FFF") == Approx(1).epsilon(
      0.01));
00237 }
00238
00243 TEST_CASE( "Comprobación Diagnostic Trouble Codes", "[DTC]" ) {
00244     REQUIRE( convertDTCs("0123") == "P0123");
00245     REQUIRE( convertDTCs("1234") == "P1234");
00246     REQUIRE( convertDTCs("2345") == "P2345");
```

```
00247     REQUIRE( convertDTCs("3456") == "P3456");
00248     REQUIRE( convertDTCs("4567") == "C0567");
00249     REQUIRE( convertDTCs("5678") == "C1678");
00250     REQUIRE( convertDTCs("6789") == "C2789");
00251     REQUIRE( convertDTCs("789A") == "C389A");
00252     REQUIRE( convertDTCs("89AB") == "B09AB");
00253     REQUIRE( convertDTCs("9ABC") == "B1ABC");
00254     REQUIRE( convertDTCs("ABCD") == "B2BCD");
00255     REQUIRE( convertDTCs("BCDE") == "B3CDE");
00256     REQUIRE( convertDTCs("CDEF") == "U0DEF");
00257     REQUIRE( convertDTCs("DEF0") == "U1EF0");
00258     REQUIRE( convertDTCs("EF01") == "U2F01");
00259     REQUIRE( convertDTCs("F012") == "U3012");
00260 }
00261
00266 TEST_CASE( "Test VIN (Vehicle Identification Number)", "[decoders]" ) {
00267     REQUIRE( decodeVIN((char *)"01573056\n1:3058455036384A\n2:34313430303530") == "
      W0V0XEP68J4140050");
00268     REQUIRE( decodeVIN((char *)"01314434\n1:47503030523535\n2:42313233343536") == "
      1D4GP00R55B123456");
00269 }
00270
00275 TEST_CASE( "Test Describir el Protocolo Actual", "[decoders]" ) {
00276     REQUIRE( decodeDescribeProtocol((char *)"AUTO, ISO 15765-4 (CAN 11/500)") == "
      AUTO, ISO 15765-4 (CAN 11/500)");
00277     REQUIRE( decodeDescribeProtocol((char *)"ISO 9141-2") == "ISO 9141-2");
00278 }
00279
00284 SCENARIO( "Test de Sensores de Oxígeno", "[decoders]" ) {
00285     GIVEN("La estructura OxigenoResponse con valores 0") {
00286         struct OxigenoResponse datosOX = {0,0};
00287
00288         REQUIRE( datosOX.A == 0 );
00289         REQUIRE( datosOX.B == 0 );
00290
00291         WHEN( "Valores de entrada mínimos 0x0000" ) {
00292             datosOX = decodeSensorOxigeno((char *)"0000");
00293
00294             THEN( "Resultados tienen que ser los mínimos" ) {
00295                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00296                 REQUIRE( datosOX.B == Approx(-100).epsilon(0.01));
00297             }
00298         }
00299         WHEN( "Valores de entrada máximos 0xFFFF" ) {
00300             datosOX = decodeSensorOxigeno((char *)"FFFF");
00301
00302             THEN( "Resultados tienen que ser los máximos" ) {
00303                 REQUIRE( datosOX.A == Approx(1.275).epsilon(0.01));
00304                 REQUIRE( datosOX.B == Approx(99.2).epsilon(0.01));
00305             }
00306         }
00307         WHEN( "Valores de entrada intermedio 0x7F7F" ) {
00308             datosOX = decodeSensorOxigeno((char *)"7F7F");
00309
00310             THEN( "Resultados tienen que ser intermedios" ) {
00311                 REQUIRE( datosOX.A == Approx(0.635).epsilon(0.01));
00312                 REQUIRE( datosOX.B == Approx(-0.78125).epsilon(0.01));
00313             }
00314         }
00315         WHEN( "Valores Voltaje Mínimo y Ajuste Combustible Máximo 0x00FF" ) {
00316             datosOX = decodeSensorOxigeno((char *)"00FF");
00317
00318             THEN( "Resultados tienen que ser Voltaje Mínimo y Ajuste Combustible Máximo" ) {
00319                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00320                 REQUIRE( datosOX.B == Approx(99.2).epsilon(0.01));
00321             }
00322         }
00323         WHEN( "Valores Ajuste Combustible Mínimo y Voltaje Máximo 0xFF00" ) {
00324             datosOX = decodeSensorOxigeno((char *)"FF00");
00325
00326             THEN( "Resultados tienen que ser Ajuste Combustible Mínimo y Voltaje Máximo" ) {
00327                 REQUIRE( datosOX.A == Approx(1.275).epsilon(0.01));
00328                 REQUIRE( datosOX.B == Approx(-100).epsilon(0.01));
00329             }
00330         }
00331     }
00332 }
00333
00338 SCENARIO( "Test de Relación Equivalente Combustible-Aire", "[decoders]" ) {
00339     GIVEN("La estructura OxigenoResponse con valores 0") {
00340         struct OxigenoResponse datosOX = {0,0};
00341
00342         REQUIRE( datosOX.A == 0 );
00343         REQUIRE( datosOX.B == 0 );
00344
00345         WHEN( "Valores de entrada mínimos 0x00000000" ) {
00346             datosOX = decodeRelacionCombAire((char *)"00000000");
```

```
00347
00348             THEN( "Resultados tienen que ser los mínimos" ) {
00349                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00350                 REQUIRE( datosOX.B == Approx(0).epsilon(0.01));
00351             }
00352         }
00353         WHEN( "Valores de entrada máximos 0xFFFFFFFF" ) {
00354             datosOX = decodeRelacionCombAire((char *)"FFFFFFFF");
00355
00356             THEN( "Resultados tienen que ser los máximos" ) {
00357                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00358                 REQUIRE( datosOX.B == Approx(8).epsilon(0.01));
00359             }
00360         }
00361         WHEN( "Valores de entrada intermedio 0x7FFF7FFF" ) {
00362             datosOX = decodeRelacionCombAire((char *)"7FFF7FFF");
00363
00364             THEN( "Resultados tienen que ser intermedios" ) {
00365                 REQUIRE( datosOX.A == Approx(1).epsilon(0.01));
00366                 REQUIRE( datosOX.B == Approx(4).epsilon(0.01));
00367             }
00368         }
00369         WHEN( "Valores Voltaje Mínimo y Relación Equivalente Máxima 0xFFFF0000" ) {
00370             datosOX = decodeRelacionCombAire((char *)"FFFF0000");
00371
00372             THEN( "Resultados tienen que ser Voltaje Mínimo y Relación Equivalente Máxima" ) {
00373                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00374                 REQUIRE( datosOX.B == Approx(0).epsilon(0.01));
00375             }
00376         }
00377         WHEN( "Valores Relación Equivalente Mínima y Voltaje Máximo 0x0000FFFF" ) {
00378             datosOX = decodeRelacionCombAire((char *)"0000FFFF");
00379
00380             THEN( "Resultados tienen que ser Relación Equivalente MínimA y Voltaje Máximo" ) {
00381                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00382                 REQUIRE( datosOX.B == Approx(8).epsilon(0.01));
00383             }
00384         }
00385     }
00386 }
00387
00392 SCENARIO( "Test de Relación Equivalente Combustible-Aire Actual", "[decoders]" ) {
00393     GIVEN("La estructura OxigenoResponse con valores 0") {
00394         struct OxigenoResponse datosOX = {0,0};
00395
00396         REQUIRE( datosOX.A == 0 );
00397         REQUIRE( datosOX.B == 0 );
00398
00399         WHEN( "Valores de entrada mínimos 0x00000000" ) {
00400             datosOX = decodeRelacionCombAireActual((char *)"00000000");
00401
00402             THEN( "Resultados tienen que ser los mínimos" ) {
00403                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00404                 REQUIRE( datosOX.B == Approx(-128).epsilon(0.01));
00405             }
00406         }
00407         WHEN( "Valores de entrada máximos 0xFFFFFFFF" ) {
00408             datosOX = decodeRelacionCombAireActual((char *)"FFFFFFFF");
00409
00410             THEN( "Resultados tienen que ser los máximos" ) {
00411                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00412                 REQUIRE( datosOX.B == Approx(128).epsilon(0.01));
00413             }
00414         }
00415         WHEN( "Valores de entrada intermedio 0x7FFF7FFF" ) {
00416             datosOX = decodeRelacionCombAireActual((char *)"7FFF7FFF");
00417
00418             THEN( "Resultados tienen que ser intermedios" ) {
00419                 REQUIRE( datosOX.A == Approx(1).epsilon(0.01));
00420                 REQUIRE( datosOX.B == Approx(-0.00390625).epsilon(0.01));
00421             }
00422         }
00423         WHEN( "Valores Corriente Mínimo y Relación Equivalente Máxima 0xFFFF0000" ) {
00424             datosOX = decodeRelacionCombAireActual((char *)"FFFF0000");
00425
00426             THEN( "Resultados tienen que ser Corriente Mínimo y Relación Equivalente Máxima" ) {
00427                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00428                 REQUIRE( datosOX.B == Approx(-128).epsilon(0.01));
00429             }
00430         }
00431         WHEN( "Valores Relación Equivalente Mínima y Corriente Máximo 0x0000FFFF" ) {
00432             datosOX = decodeRelacionCombAireActual((char *)"0000FFFF");
00433
00434             THEN( "Resultados tienen que ser Relación Equivalente MínimA y Corriente Máximo" ) {
00435                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00436                 REQUIRE( datosOX.B == Approx(128).epsilon(0.01));
00437             }
```

```
00438         }
00439     }
00440 }
00441
00446 SCENARIO( "Test de Valores máximos relación de combustible-aire, voltaje, corriente y presión
       absoluta", "[decoders]" ) {
00447     GIVEN("La estructura RelacionesResponse con valores 0") {
00448         struct RelacionesResponse datosREL = {0,0,0,0};
00449
00450         REQUIRE( datosREL.A == 0 );
00451         REQUIRE( datosREL.B == 0 );
00452         REQUIRE( datosREL.C == 0 );
00453         REQUIRE( datosREL.D == 0 );
00454
00455         WHEN( "Valores de entrada mínimos 0x00000000" ) {
00456             datosREL = decodeRelaciones((char *)"00000000");
00457
00458             THEN( "Resultados tienen que ser los mínimos" ) {
00459                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00460                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00461                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00462                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00463             }
00464         }
00465         WHEN( "Valores de entrada máximos 0xFFFFFFFF" ) {
00466             datosREL = decodeRelaciones((char *)"FFFFFFFF");
00467
00468             THEN( "Resultados tienen que ser los máximos" ) {
00469                 REQUIRE( datosREL.A == Approx(255).epsilon(0.01));
00470                 REQUIRE( datosREL.B == Approx(255).epsilon(0.01));
00471                 REQUIRE( datosREL.C == Approx(255).epsilon(0.01));
00472                 REQUIRE( datosREL.D == Approx(2550).epsilon(0.01));
00473             }
00474         }
00475         WHEN( "Valores de entrada intermedio 0x7F7F7F7F" ) {
00476             datosREL = decodeRelaciones((char *)"7F7F7F7F");
00477
00478             THEN( "Resultados tienen que ser intermedios" ) {
00479                 REQUIRE( datosREL.A == Approx(127).epsilon(0.01));
00480                 REQUIRE( datosREL.B == Approx(127).epsilon(0.01));
00481                 REQUIRE( datosREL.C == Approx(127).epsilon(0.01));
00482                 REQUIRE( datosREL.D == Approx(1270).epsilon(0.01));
00483             }
00484         }
00485         WHEN( "Valores de entrada Relación Máxima, resto mínimo 0xFF000000" ) {
00486             datosREL = decodeRelaciones((char *)"FF000000");
00487
00488             THEN( "Resultados tienen que ser Relación Máxima, resto mínimo" ) {
00489                 REQUIRE( datosREL.A == Approx(255).epsilon(0.01));
00490                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00491                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00492                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00493             }
00494         }
00495         WHEN( "Valores Voltaje Máximo, resto mínimo 0x00FF0000" ) {
00496             datosREL = decodeRelaciones((char *)"00FF0000");
00497
00498             THEN( "Resultados tienen que ser Voltaje Máximo, resto mínimo" ) {
00499                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00500                 REQUIRE( datosREL.B == Approx(255).epsilon(0.01));
00501                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00502                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00503             }
00504         }
00505         WHEN( "Valores Corriente Máxima, resto mínimo 0x0000FF00" ) {
00506             datosREL = decodeRelaciones((char *)"0000FF00");
00507
00508             THEN( "Resultados tienen que ser Corriente Máxima, resto mínimo" ) {
00509                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00510                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00511                 REQUIRE( datosREL.C == Approx(255).epsilon(0.01));
00512                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00513             }
00514         }
00515         WHEN( "Valores Presión Máxima, resto mínimo 0x000000FF" ) {
00516             datosREL = decodeRelaciones((char *)"000000FF");
00517
00518             THEN( "Resultados tienen que ser Presión Máxima, resto mínimo" ) {
00519                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00520                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00521                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00522                 REQUIRE( datosREL.D == Approx(2550).epsilon(0.01));
00523             }
00524         }
00525     }
00526 }
00527
```

```
00532 SCENARIO( "Test de decodificación de Data Trouble Codes (DTC)", "[decoders]" ) {
00533     GIVEN("Vector vacío cuyos componentes son strings (DTC's)") {
00534         std::vector<std::string> vec_dtcs;
00535 
00536         REQUIRE( vec_dtcs.empty() == 1 );
00537 
00538         WHEN( "Sólo un DTC" ) {
00539             vec_dtcs = decodeDTCs((char *)"013300000000");
00540 
00541             THEN( "Resultado de sólo un DTC" ) {
00542                 REQUIRE( vec_dtcs.size() == 1);
00543                 REQUIRE( vec_dtcs[0] == "P0133");
00544             }
00545         }
00546         WHEN( "Dos DTC's" ) {
00547             vec_dtcs = decodeDTCs((char *)"0133D0160000");
00548 
00549             THEN( "Resultado de dos DTC's" ) {
00550                 REQUIRE( vec_dtcs.size() == 2);
00551                 REQUIRE( vec_dtcs[0] == "P0133");
00552                 REQUIRE( vec_dtcs[1] == "U1016");
00553             }
00554         }
00555         WHEN( "Tres DTC's" ) {
00556             vec_dtcs = decodeDTCs((char *)"0133D0161131");
00557 
00558             THEN( "Resultado de tres DTC's" ) {
00559                 REQUIRE( vec_dtcs.size() == 3);
00560                 REQUIRE( vec_dtcs[0] == "P0133");
00561                 REQUIRE( vec_dtcs[1] == "U1016");
00562                 REQUIRE( vec_dtcs[2] == "P1131");
00563             }
00564         }
00565     }
00566 }
00567 
00572 SCENARIO( "Test de decodificación PIDs disponibles", "[decoders]" ) {
00573     GIVEN("Vector vacío cuyos componentes son int (DTC's)") {
00574         std::vector<int> vec_pids;
00575 
00576         REQUIRE( vec_pids.empty() == 1 );
00577 
00578         WHEN( "PIDs impares disponibles" ) {
00579             vec_pids = decodePIDS((char *)"AAAAAAAA");
00580 
00581             THEN( "Resultado con números impares del 1-20 (Hexadecimal, decimal 1-32)" ) {
00582                 REQUIRE( vec_pids.size() == 16);
00583                 int pids_impares = 1;
00584                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00585                     REQUIRE( vec_pids[i] == pids_impares);
00586                     pids_impares+=2;
00587 
00588                 }
00589             }
00590         }
00591         WHEN( "PIDs pares disponibles" ) {
00592             vec_pids = decodePIDS((char *)"55555555");
00593 
00594             THEN( "Resultado con números pares del 1-20 (Hexadecimal, decimal 1-32)" ) {
00595                 REQUIRE( vec_pids.size() == 16);
00596                 int pids_pares = 2;
00597                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00598                     REQUIRE( vec_pids[i] == pids_pares);
00599                     pids_pares+=2;
00600 
00601                 }
00602             }
00603         }
00604         WHEN( "PIDs primeros disponibles" ) {
00605             vec_pids = decodePIDS((char *)"FFFF0000");
00606 
00607             THEN( "Resultado con los primeros números del 1-20 (Hexadecimal, decimal 1-32)" ) {
00608                 REQUIRE( vec_pids.size() == 16);
00609                 int pids_primeros = 1;
00610                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00611                     REQUIRE( vec_pids[i] == pids_primeros);
00612                     pids_primeros++;
00613 
00614                 }
00615             }
00616         }
00617         WHEN( "PIDs ultimos disponibles" ) {
00618             vec_pids = decodePIDS((char *)"0000FFFF");
00619 
00620             THEN( "Resultado con los ultimos números del 1-20 (Hexadecimal, decimal 1-32)" ) {
00621                 REQUIRE( vec_pids.size() == 16);
00622                 int pids_ultimos = 17;
```

```
00623                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00624                     REQUIRE( vec_pids[i] == pids_ultimos);
00625                     pids_ultimos++;
00626 
00627                 }
00628             }
00629         }
00630     }
00631 }
00632 
00637 SCENARIO( "Test de decodificación del estado del coche", "[decoders]" ) {
00638     GIVEN("Map de <string, string> de estado vacío") {
00639         std::map<std::string, std::string> status;
00640 
00641         REQUIRE( status.empty() == 1 );
00642 
00643         WHEN( "La entrada es 0x81076504" ) {
00644             status = decodeStatus((char *)"81076504");
00645 
00646             THEN( "Resultado 1 DTC, Compresión, MIL Encendida y todas pruebas correctas" ) {
00647                 REQUIRE( status.size() == 9);
00648                 REQUIRE( status["MIL"] == "Encendida");
00649                 REQUIRE( status["DTC_CNT"] == "1");
00650                 REQUIRE( status["IGNICION"] == "Compresión");
00651                 REQUIRE( status["Sistema de catalizador NMHC"] == "Prueba Correcta");
00652                 REQUIRE( status["Sistema de detección de condiciones inadecuadas de ignición en cilindros"]
       == "Prueba Correcta");
00653                 REQUIRE( status["Sistema de componentes integrales"] == "Prueba Correcta");
00654                 REQUIRE( status["Sistema de combustible"] == "Prueba Correcta");
00655                 REQUIRE( status["Sistema del sensor de gases de escape"] == "Prueba Correcta");
00656                 REQUIRE( status["Sistema de monitor del filtro de partículas (Particular Matter, PM)"] == "
     Prueba Correcta");
00657             }
00658         }
00659     }
00660 }
```

Índice alfabético