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.
- bool sendAlarm (std::string msg)

  Método para enviar el mensaje/alarma al servidor remoto.

4.1.1. Descripción detallada

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

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

Definición en la línea 21 del archivo alarmfile.hpp.

4.1.2. Documentación del constructor y destructor

4.1.2.1. AlarmFile()

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

Constructor de la clase AlarmFile.

Parámetros

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

Devuelve

Devuelve una instancia de la clase AlarmFile.

Definición en la línea 26 del archivo alarmfile.cpp.

4.1.3. Documentación de las funciones miembro

4.1.3.1. sendAlarm()

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

Método para enviar el mensaje/alarma al servidor remoto.

Parámetros

msg	String con el mensaje a enviar al servidor remoto.

Devuelve

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

Definición en la línea 44 del archivo alarmfile.cpp.

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

- alarmfile.hpp
- alarmfile.cpp

4.2. Referencia de la Clase Commands

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

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

Métodos públicos

- Commands (json data)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2.1. Descripción detallada

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

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

Definición en la línea 29 del archivo Commands.hpp.

4.2.2. Documentación del constructor y destructor

4.2.2.1. Commands()

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

Constructor de la clase Commands.

Parámetros

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

Devuelve

Devuelve una instancia de la clase Commands.

Definición en la línea 38 del archivo Commands.hpp.

4.2.3. Documentación de las funciones miembro

4.2.3.1. getBytesResponse()

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

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

Devuelve

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

Definición en la línea 76 del archivo Commands.hpp.

4.2.3.2. getCMD()

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

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

Devuelve

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

Definición en la línea 69 del archivo Commands.hpp.

4.2.3.3. getCMDResponse()

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

Método para obtener el comando de respuesta al PID solicitado.

Devuelve

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

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

Definición en la línea 195 del archivo Commands.hpp.

4.2.3.4. getDecoder()

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

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

Devuelve

String del decodificador que se debe ejecutar en la respuesta.

Definición en la línea 83 del archivo Commands.hpp.

4.2.3.5. getDescription()

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

Método que obtiene la descripción del comando.

Devuelve

String con la descripción del comando.

Definición en la línea 62 del archivo Commands.hpp.

4.2.3.6. getJson()

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

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

Devuelve

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

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

Definición en la línea 132 del archivo Commands.hpp.

4.2.3.7. getMAX()

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

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

Devuelve

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

Definición en la línea 97 del archivo Commands.hpp.

4.2.3.8. getMIN()

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

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

Devuelve

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

Definición en la línea 90 del archivo Commands.hpp.

4.2.3.9. getName()

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

Método que obtiene el nombre del comando.

Devuelve

String con el nombre del comando.

Definición en la línea 55 del archivo Commands.hpp.

4.2.3.10. getResValue()

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

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

Devuelve

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

Definición en la línea 122 del archivo Commands.hpp.

4.2.3.11. getTypeData()

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

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

Devuelve

String del tipo de dato de la respuesta del comando.

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

Definición en la línea 115 del archivo Commands.hpp.

4.2.3.12. getUnits()

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

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

Devuelve

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

Definición en la línea 104 del archivo Commands.hpp.

4.2.3.13. setBytesResponse()

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

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

Parámetros

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

Definición en la línea 228 del archivo Commands.hpp.

4.2.3.14. setCMD()

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

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

Parámetros

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

Definición en la línea 221 del archivo Commands.hpp.

4.2.3.15. setDecoder()

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

Método para asignar un decodificador a un comando.

Parámetros

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

Definición en la línea 235 del archivo Commands.hpp.

4.2.3.16. setDescription()

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

Método para asignar una descripción a un comando.

Parámetros

description	String con la descripción a asignar al comando.

Definición en la línea 214 del archivo Commands.hpp.

4.2.3.17. setMAX()

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

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

Parámetros

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

Definición en la línea 249 del archivo Commands.hpp.

4.2.3.18. setMIN()

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

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

Parámetros

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

Definición en la línea 242 del archivo Commands.hpp.

4.2.3.19. setName()

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

Método para asignar un nombre a un comando.

Parámetros

name	String con el nombre a asignar al comando.

Definición en la línea 207 del archivo Commands.hpp.

4.2.3.20. setResValue()

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

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

Parámetros

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

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

Definición en la línea 272 del archivo Commands.hpp.

4.2.3.21. setTypeData()

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

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

Parámetros

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

Definición en la línea 263 del archivo Commands.hpp.

4.2.3.22. setUnits()

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

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

Parámetros

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

Definición en la línea 256 del archivo Commands.hpp.

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

- Commands.hpp

4.3. Referencia de la Clase GpsClient

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

`#include <gpsclient.hpp>`

Métodos públicos

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

4.3.1. Descripción detallada

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

Clase utilizada para la obtención de coordenadas GPS.

Definición en la línea 28 del archivo gpsclient.hpp.

4.3.2. Documentación del constructor y destructor

4.3.2.1. GpsClient()

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

Constructor de la clase GpsClient.

Parámetros

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

Devuelve

Devuelve una instancia de la clase GpsClient.

Definición en la línea 13 del archivo gpsclient.cpp.

4.3.3. Documentación de las funciones miembro

4.3.3.1. getGPS()

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

Método que obtiene una string con las coordenadas GPS.

Devuelve

String con las coordenadas GPS.

Definición en la línea 37 del archivo gpsclient.cpp.

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

- gpsclient.hpp
- gpsclient.cpp

4.4. Referencia de la Clase Obd

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

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

Métodos públicos

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

Campos de datos

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

4.4.1. Descripción detallada

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

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

Definición en la línea 73 del archivo Obd.hpp.

4.4.2. Documentación del constructor y destructor

4.4.2.1. Obd()

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

Constructor de la clase Obd.

Parámetros

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

Devuelve

Devuelve una instancia de la clase Obd.

Definición en la línea 83 del archivo Obd.hpp.

4.4.3. Documentación de las funciones miembro

4.4.3.1. connectBluetooth()

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

Método que realiza la conexión con el dispositivo bluetooth ELM327.

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

Definición en la línea 167 del archivo Obd.hpp.

4.4.3.2. disconnectBluetooth()

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

Método de desconexión bluetooth con el dispositivo ELM327.

Cierra el socket e instancia epoll abiertas.

Definición en la línea 552 del archivo Obd.hpp.

4.4.3.3. discoverDeviceAddress()

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

Método que realiza el descubrimiento bluetooth del dispositivo ELM327.

Parámetros

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

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

Definición en la línea 112 del archivo Obd.hpp.

4.4.3.4. existPID()

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

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

Parámetros

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

Devuelve

Devuelve true si existe y false en caso contrario.

Definición en la línea 564 del archivo Obd.hpp.

4.4.3.5. getDTCs()

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

Método que permite obtener los DTC activos en el vehículo.

Devuelve

Vector de strings con los DTC activos en el vehículo

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

Definición en la línea 628 del archivo Obd.hpp.

4.4.3.6. getVIN()

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

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

Devuelve

String del VIN de 17 dígitos del vehículo.

Definición en la línea 615 del archivo Obd.hpp.

4.4.3.7. initDecoderFunctions()

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

Método de inicialización de funciones de decodificación de mensajes OBD.

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

Definición en la línea 511 del archivo Obd.hpp.

4.4.3.8. initMessages()

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

Método de inicialización de parámetros de conexión con ELM327.

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

Definición en la línea 476 del archivo Obd.hpp.

4.4.3.9. isValid()

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

Método de validación del estado de la conexion.

Devuelve

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

Definición en la línea 655 del archivo Obd.hpp.

4.4.3.10. polling()

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

Método de recepción de mensajes enviados por el dispositivo ELM327.

Parámetros

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

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

Definición en la línea 295 del archivo Obd.hpp.

4.4.3.11. printPIDs()

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

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

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

Definición en la línea 583 del archivo Obd.hpp.

4.4.3.12. printStatus()

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

Método de impresión de las pruebas realizadas en el vehículo.

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

Definición en la línea 603 del archivo Obd.hpp.

4.4.3.13. readFileData()

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

Método de lectura del fichero de PIDS en formato json.

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

Definición en la línea 229 del archivo Obd.hpp.

4.4.3.14. send()

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

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

Parámetros

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

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

Definición en la línea 249 del archivo Obd.hpp.

4.4.4. Documentación de los campos

4.4.4.1. map_commands

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

Map para asignación del nombre al comando correspondiente

Definición en la línea 75 del archivo Obd.hpp.

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

- Obd.hpp

4.5. Referencia de la Estructura OxigenoResponse

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

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

Campos de datos

- float A
- float B

4.5.1. Descripción detallada

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

Definición en la línea 25 del archivo decoders.hpp.

4.5.2. Documentación de los campos

4.5.2.1. A

```
float OxigenoResponse::A
```

Valor A en la formula de decodificación

Definición en la línea 26 del archivo decoders.hpp.

4.5.2.2. B

```
float OxigenoResponse::B
```

Valor B en la formula de decodificación

Definición en la línea 27 del archivo decoders.hpp.

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

- decoders.hpp

4.6. Referencia de la Estructura RelacionesResponse

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

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

Campos de datos

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

4.6.1. Descripción detallada

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

Definición en la línea 34 del archivo decoders.hpp.

4.6.2. Documentación de los campos

4.6.2.1. A

```
int RelacionesResponse::A
```

Valor A en la formula de decodificación

Definición en la línea 35 del archivo decoders.hpp.

4.6.2.2. B

```
int RelacionesResponse::B
```

Valor B en la formula de decodificación

Definición en la línea 36 del archivo decoders.hpp.

4.6.2.3. C

```
int RelacionesResponse::C
```

Valor C en la formula de decodificación

Definición en la línea 37 del archivo decoders.hpp.

4.6.2.4. D

```
int RelacionesResponse::D
```

Valor D en la formula de decodificación

Definición en la línea 38 del archivo decoders.hpp.

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

- decoders.hpp

Capítulo 5

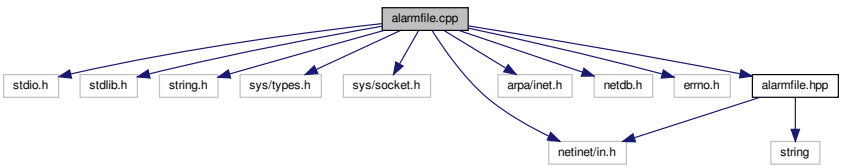
Documentación de archivos

5.1. Referencia del Archivo alarmfile.cpp

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

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

Dependencia gráfica adjunta para alarmfile.cpp:



5.1.1. Descripción detallada

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

Autor

Juan Manuel Vozmediano Torres

Fecha

09/04/2019

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 void AlarmFile::shit (const char *mens)
00021 {
00022   fprintf(stderr, "%s - %d\n", mens, errno);
00023   perror("Error is ");
00024 }
00025
00026 AlarmFile::AlarmFile(std::string AlarmHost,
00027                      std::string AlarmPort,
00028                      std::string AlarmFilename,
00029                      std::string LastAlarmFilename):
00030   alarmHost_(AlarmHost),
00031   alarmPort_(atoi(AlarmPort.c_str())),
00032   alarmFilename_(AlarmFilename),
00033   lastAlarmFilename_(LastAlarmFilename)
00034 {
00035   if ((s_ = socket (AF_INET, SOCK_DGRAM, 0)) < 0) shit ("socket");
00036
00037   memset ((char *)&iTu_, 0, sizeof(struct sockaddr_in));
00038   iTu_.sin_family      = AF_INET;
00039   iTu_.sin_addr.s_addr = inet_addr(alarmHost_.c_str());
00040   iTu_.sin_port        = htons(alarmPort_);
00041 }
00042
00043
00044 bool AlarmFile::sendAlarm(std::string msg)
00045 {
00046   if ( "" != msg ){
00047     int cc =  sendto(s_, msg.c_str(), strlen(msg.c_str()), 0,(struct sockaddr *)&iTu_, sizeof(iTu_));
00048
00049     if (cc < 0){
00050       perror("Error is ");
00051       fprintf(stderr, "Value of errno: %d\n", errno);
00052     }
00053
00054     fprintf(stderr, "Alarma enviada (%d): %s a %s:%d\n", cc, msg.c_str(), inet_ntoa(iTu_.sin_addr), (int)
      ntohs(iTu_.sin_port));
00055
00056     return true;
00057   }
00058   return false;
00059 }
00060
```

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:

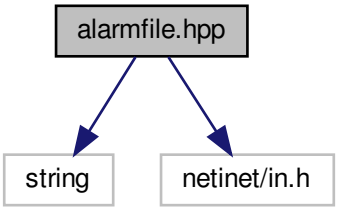
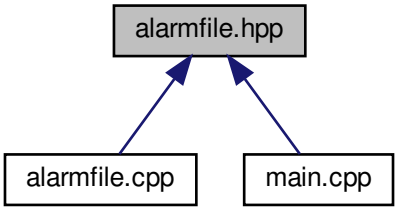


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



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
00038
00045   bool sendAlarm(std::string msg);
00046 private:
00047
00053   void shit (const char *mens);
00054   std::string alarmHost_;
00055   int alarmPort_;
00056   std::string alarmFilename_;
00057   std::string lastAlarmFilename_;
00058   int s_;
00059   struct sockaddr_in iTu_;
00060 };
00061
00062 #endif
```

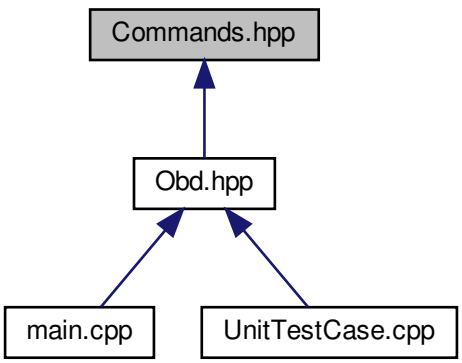
5.5. Referencia del Archivo Commands.hpp

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:

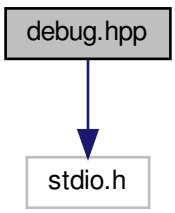
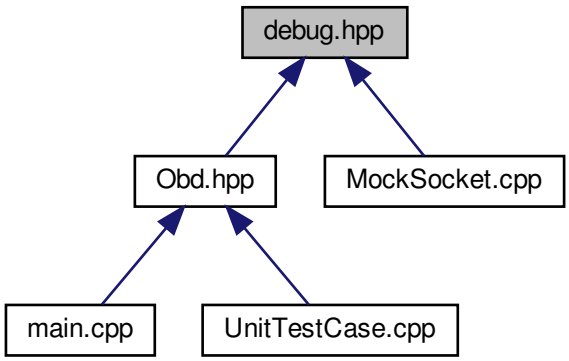


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



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.

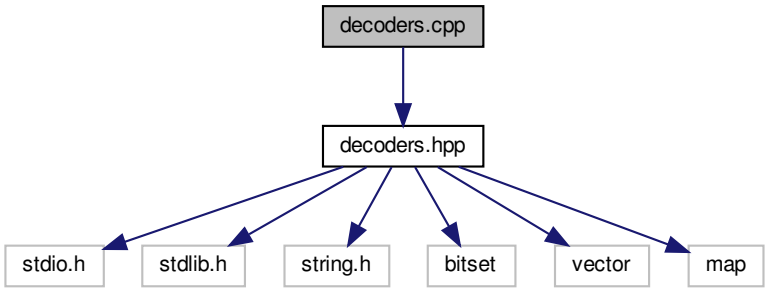
5.8. debug.hpp

```
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) para ISO15765-4 CAN.
- std::string convertDTCs (std::string dtc)

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

Función de decodificación de los DTC activos en el vehículo.

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

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

  Función de decodificación del PID STATUS.
- float decodeCargaPosicionEGR (char *response)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.9.1. 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 71 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 308 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 340 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 294 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 110 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 322 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 136 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 408 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 315 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 415 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 466 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 422 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 477 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 520 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 544 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 329 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 365 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 152 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 500 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 301 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 351 del archivo decoders.cpp.

5.9.2.23. decodeVIN()

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

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

Parámetros

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

Devuelve

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

Definición en la línea 21 del archivo decoders.cpp.

5.9.2.24. 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 509 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 /* decodeVIN para vehículos que no soportan ISO 15765-4 CAN
00047 std::string decodeVIN(char * response){
00048     std::string bytes_res(response);
00049     std::string vin;
00050
00051     //División de orden y datos
00052     std::string order = bytes_res.substr(0,8);
00053     std::string vin_bytes = bytes_res.substr(8,70);
00054
00055     std::size_t found = vin_bytes.find("\n");
00056     while(found!=std::string::npos){
00057         vin_bytes.erase(found,7);
00058         found = vin_bytes.find("\n");
00059     }
00060
00061
00062     for (uint32_t i = 0; i < vin_bytes.size(); i+=2){
00063         std::string vin_char = vin_bytes.substr(i,2);
00064         //Conversión de bytes en char
00065         vin.push_back((char) stoi(vin_char,nullptr,16));
00066     }
00067     return vin;
00068 }
00069 */
00070
00071 std::string convertDTCs(std::string dtc){
00072     if(dtc[0] == '0'){
00073         dtc.replace(0,1,"P0");
00074     } else if (dtc[0] == '1'){
00075         dtc.replace(0,1,"P1");
00076     } else if (dtc[0] == '2'){
00077         dtc.replace(0,1,"P2");
00078     } else if (dtc[0] == '3'){
```

```
00079         dtc.replace(0,1,"P3");
00080     } else if (dtc[0] == '4'){
00081         dtc.replace(0,1,"C0");
00082     } else if (dtc[0] == '5'){
00083         dtc.replace(0,1,"C1");
00084     } else if (dtc[0] == '6'){
00085         dtc.replace(0,1,"C2");
00086     } else if (dtc[0] == '7'){
00087         dtc.replace(0,1,"C3");
00088     } else if (dtc[0] == '8'){
00089         dtc.replace(0,1,"B0");
00090     } else if (dtc[0] == '9'){
00091         dtc.replace(0,1,"B1");
00092     } else if (dtc[0] == 'A'){
00093         dtc.replace(0,1,"B2");
00094     } else if (dtc[0] == 'B'){
00095         dtc.replace(0,1,"B3");
00096     } else if (dtc[0] == 'C'){
00097         dtc.replace(0,1,"U0");
00098     } else if (dtc[0] == 'D'){
00099         dtc.replace(0,1,"U1");
00100     } else if (dtc[0] == 'E'){
00101         dtc.replace(0,1,"U2");
00102     } else if (dtc[0] == 'F'){
00103         dtc.replace(0,1,"U3");
00104     }
00105 
00106     return dtc;
00107 }
00108 
00109 //Modo 03
00110 std::vector<std::string> decodeDTCs(char *response){
00111     std::vector<std::string> vec_dtcs;
00112     std::string bytes_res(response);
00113 
00114     std::string dtc_1 = bytes_res.substr(0,4);
00115     if (dtc_1.compare("0000")){
00116         dtc_1 = convertDTCs(dtc_1);
00117         vec_dtcs.push_back(dtc_1);
00118     }
00119     std::string dtc_2 = bytes_res.substr(4,4);
00120     if (dtc_2.compare("0000")){
00121         dtc_2 = convertDTCs(dtc_2);
00122         vec_dtcs.push_back(dtc_2);
00123     }
00124     std::string dtc_3 = bytes_res.substr(8,4);
00125     if (dtc_3.compare("0000")){
00126         dtc_3 = convertDTCs(dtc_3);
00127         vec_dtcs.push_back(dtc_3);
00128     }
00129 
00130     return vec_dtcs;
00131 }
00132 
00133     //Modo 01-> Descripcion - PID - Valor Mínimo - Valor Máximo - Unidad - Fórmula
00134 //00 - PIDs implementados [01 - 20] -Cada bit indica si los siguientes 32 PID están implementados (1) o no
     (0): [A7..D0] == [PID 01..20]
00135 
00136 std::vector<int> decodePIDS(char *response){
00137     //Conversión a long para poder convertirlo a bitset
00138     long value_rcv = std::stol(response, nullptr, 16);
00139     //Conversión a bitset
00140     std::bitset<PID_BITS> setBit (value_rcv);
00141     std::vector<int> vec_pids;
00142     //Comprobación de PIDs disponibles(bitset lectura al reves)
00143     for (int i = PID_BITS-1; i >= 0; i--){
00144         if(setBit[i]){
00145             vec_pids.push_back(PID_BITS-i);
00146         }
00147     }
00148     return vec_pids;
00149 }
00150 
00151 //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
00152 std::map<std::string, std::string> decodeStatus(char *response){
00153     std::map<std::string, std::string> status;
00154 
00155     std::string bytes_res(response);
00156     std::string responseA = bytes_res.substr(0,2);
00157     std::string responseB = bytes_res.substr(2,2);
00158     std::string responseC = bytes_res.substr(4,2);
00159     std::string responseD = bytes_res.substr(6,2);
00160 
00161     int intA = std::stoi(responseA, nullptr, 16);
00162     int intB = std::stoi(responseB, nullptr, 16);
00163     int intC = std::stoi(responseC, nullptr, 16);
```

```
00164     int intD = std::stoi(responseD, nullptr, 16);
00165 
00166     std::bitset<STATUS_BITS> byteA (intA);
00167     std::bitset<STATUS_BITS> byteB (intB);
00168     std::bitset<STATUS_BITS> byteC (intC);
00169     std::bitset<STATUS_BITS> byteD (intD);
00170 
00171     if (byteA[7]){
00172         status["MIL"] = "Encendida";
00173         status["DTC_CNT"] = std::to_string(intA-128);
00174     } else {
00175         status["MIL"] = "Apagada";
00176         status["DTC_CNT"] = std::to_string(intA);
00177     }
00178 
00179     if (byteB[0]){
00180         if (byteB[4])
00181             status["Sistema de detección de condiciones inadecuadas de ignición en cilindros"] = "Prueba
      Incorrecta";
00182         else
00183             status["Sistema de detección de condiciones inadecuadas de ignición en cilindros"] = "Prueba
      Correcta";
00184     }
00185     if (byteB[1]){
00186         if (byteB[5])
00187             status["Sistema de combustible"] = "Prueba Incorrecta";
00188         else
00189             status["Sistema de combustible"] = "Prueba Correcta";
00190     }
00191     if (byteB[2]){
00192         if (byteB[6])
00193             status["Sistema de componentes integrales"] = "Prueba Incorrecta";
00194         else
00195             status["Sistema de componentes integrales"] = "Prueba Correcta";
00196     }
00197 
00198     if (byteB[3]){
00199         status["IGNICION"] = "Chispa";
00200         if (byteC[0]){
00201             if (byteD[0])
00202                 status["Sistema de eficiencia del convertidor catalítico"] = "Prueba Incorrecta";
00203             else
00204                 status["Sistema de eficiencia del convertidor catalítico"] = "Prueba Correcta";
00205         }
00206         if (byteC[1]){
00207             if (byteD[1])
00208                 status["Sistema de calentamiento de convertidor catalítico"] = "Prueba Incorrecta";
00209             else
00210                 status["Sistema de calentamiento de convertidor catalítico"] = "Prueba Correcta";
00211         }
00212         if (byteC[2]){
00213             if (byteD[2])
00214                 status["Sistema evaporativo"] = "Prueba Incorrecta";
00215             else
00216                 status["Sistema evaporativo"] = "Prueba Correcta";
00217         }
00218         if (byteC[3]){
00219             if (byteD[3])
00220                 status["Sistema secundario de aire"] = "Prueba Incorrecta";
00221             else
00222                 status["Sistema secundario de aire"] = "Prueba Correcta";
00223         }
00224         if (byteC[4]){
00225             if (byteD[4])
00226                 status["Sistema de fugas de aire acondicionado"] = "Prueba Incorrecta";
00227             else
00228                 status["Sistema de fugas de aire acondicionado"] = "Prueba Correcta";
00229         }
00230         if (byteC[5]){
00231             if (byteD[5])
00232                 status["Sistema de sensores de oxígeno"] = "Prueba Incorrecta";
00233             else
00234                 status["Sistema de sensores de oxígeno"] = "Prueba Correcta";
00235         }
00236         if (byteC[6]){
00237             if (byteD[6])
00238                 status["Sistema de calentamiento del sensor de oxígeno"] = "Prueba Incorrecta";
00239             else
00240                 status["Sistema de calentamiento del sensor de oxígeno"] = "Prueba Correcta";
00241         }
00242         if (byteC[7]){
00243             if (byteD[7])
00244                 status["Sistema de recirculación de los gases de escape (Exhaust Gas Recicrulation, EGR)"]
      = "Prueba Incorrecta";
00245             else
00246                 status["Sistema de recirculación de los gases de escape (Exhaust Gas Recicrulation, EGR)"]
      = "Prueba Correcta";
```

```
00247         }
00248     } else {
00249         status["IGNICION"] = "Compresión";
00250         if (byteC[0]){
00251             if (byteD[0])
00252                 status["Sistema de catalizador NMHC"] = "Prueba Incorrecta";
00253             else
00254                 status["Sistema de catalizador NMHC"] = "Prueba Correcta";
00255         }
00256         if (byteC[1]){
00257             if (byteD[1])
00258                 status["Sistema monitor de NOx/SCR"] = "Prueba Incorrecta";
00259             else
00260                 status["Sistema monitor de NOx/SCR"] = "Prueba Correcta";
00261         }
00262         if (byteC[3]){
00263             if (byteD[3])
00264                 status["Sistema de presión de impulso"] = "Prueba Incorrecta";
00265             else
00266                 status["Sistema de presión de impulso"] = "Prueba Correcta";
00267         }
00268         if (byteC[5]){
00269             if (byteD[5])
00270                 status["Sistema del sensor de gases de escape"] = "Prueba Incorrecta";
00271             else
00272                 status["Sistema del sensor de gases de escape"] = "Prueba Correcta";
00273         }
00274         if (byteC[6]){
00275             if (byteD[6])
00276                 status["Sistema de monitor del filtro de partículas (Particular Matter, PM)"] = "Prueba
      Incorrecta";
00277             else
00278                 status["Sistema de monitor del filtro de partículas (Particular Matter, PM)"] = "Prueba
      Correcta";
00279         }
00280         if (byteC[7]){
00281             if (byteD[7])
00282                 status["Sistema de recirculación de gases de escape (Exhaust Gas Recirculation, EGR) y/o
      VVT"] = "Prueba Incorrecta";
00283             else
00284                 status["Sistema de recirculación de gases de escape (Exhaust Gas Recirculation, EGR) y/o
      VVT"] = "Prueba Correcta";
00285         }
00286     }
00287
00288     return status;
00289 }
00290 //02
00291 //03
00292 //04 - Carga calculada del motor , 0 , 100 , % , A/2.55
00293
00294 float decodeCargaPosicionEGR(char *response){
00295     int dec = (int)strtol(response, NULL, 16);
00296     return dec/2.55;
00297 }
00298
00299 //05 - Temperatura del líquido de enfriamiento del motor , -40 , 215 , °C , A-40
00300
00301 float decodeTempGeneral(char *response){
00302     int dec = (int)strtol(response, NULL, 16);
00303     return dec-40;
00304 }
00305
00306 //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
00307
00308 float decodeAjusteCombustibleEGR(char *response){
00309     int dec = (int)strtol(response, NULL, 16);
00310     return (dec/1.28)-100;
00311 }
00312
00313 //0a - Presión del combustible , 0 , 765 , kPa , 3A
00314
00315 float decodePresionCombustible(char *response){
00316     int dec = (int)strtol(response, NULL, 16);
00317     return 3*dec;
00318 }
00319
00320 //0b - Presión absoluta del colector de admisión , 0 , 255 , kPa , A
00321
00322 float decodeHexToDec(char *response){
00323     int dec = (int)strtol(response, NULL, 16);
00324     return dec;
00325 }
00326
00327 //0c - RPM del motor , 0 , 16,383.75 , rpm , (256A+B)/4
00328
```

```
00329 float decodeRPM(char *response){
00330     int dec = (int)strtol(response, NULL, 16);
00331     return dec/4.0;
00332 }
00333
00334 //0d - Velocidad del vehículo , 0 , 255 , km/h , A
00335
00336 //decodeHexToDec
00337
00338 //0e - Avance del tiempo , -64 , 63.5 , ° antes TDC , A/2-64
00339
00340 float decodeAvanceTiempo(char *response){
00341     int dec = (int)strtol(response, NULL, 16);
00342     return (dec/2.0)-64;
00343 }
00344
00345 //0f - Temperatura del aire del colector de admisión , -40 , 215 , °C, A-40
00346
00347 //decodeTempGeneral
00348
00349 //10 - Velocidad del flujo del aire MAF , 0 , 655.35  , gr/sec , (256A+B)/100
00350
00351 float decodeVelocidadMAF(char *response){
00352     int dec = (int)strtol(response, NULL, 16);
00353     return dec/100.0;
00354 }
00355
00356 //11 - Posición del acelerador , 0 , 100 , % , A/2.55
00357
00358 //decodeCargaPosicionEGR
00359
00360
00361 //12 - Estado del aire secundario controlado
00362 //13 - Presencia de sensores de oxígeno (en 2 bancos)
00363 //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)
00364
00365 struct OxigenoResponse decodeSensorOxigeno(char *response){
00366     char AResponse[3], BResponse[3];
00367     float A, B;
00368     struct OxigenoResponse datos;
00369     //Añade caracter \0 al final de la cadena AResponse
00370     memset(AResponse, '\0', sizeof(AResponse));
00371     //Divide los bytes de respuesta
00372     strncpy( AResponse, response, 2);
00373     strcpy( BResponse, response + 2 );
00374
00375     A = (int)strtol(AResponse, NULL, 16);
00376     datos.A = A/200;
00377     /*
00378     Según la documentación si B==FF, entonces el sensor no se usa en el cálculo del ajuste
00379     if (strcmp(BResponse, "FF") == 0){
00380         //Si B==FF, entonces el sensor no se usa en el cálculo del ajuste
00381         datos.B=0;
00382     } else {
00383         //Si B!=FF, se aplica fórmula
00384         B = (int)strtol(BResponse, NULL, 16);
00385         datos.B = (B/1.28)-100;
00386     }
00387     */
00388     B = (int)strtol(BResponse, NULL, 16);
00389     datos.B = (B/1.28)-100;
00390
00391     return datos;
00392 }
00393
00394 //1c - Estándar OBD implementado en este vehículo ,
00395 //1d - Sensores de oxígenos presentes en el banco 4
00396 //1e - Estado de las entradas auxiliares
00397 //1f - Tiempo desde que se puso en marcha el motor , 0 , 65,535 , sec , 256A+B
00398
00399 //decodeHexToDec
00400
00401 //20 - PID implementados [21 - 40]
00402 //21 - Distancia recorrida con la luz indicadora de falla (Malfunction Indicator Lamp, MIL) encendida , 0 ,
      65,535 , km
00403
00404 //decodeHexToDec
00405
00406 //22 - Presión del tren de combustible, relativa al colector de vacío , 0 , 5177.265 , kPa, 0.079(256A+B)
00407
00408 float decodePresionCombColector(char *response){
00409     int dec = (int)strtol(response, NULL, 16);
00410     return 0.079*dec;
00411 }
00412
```

```
00413 //23 - Presión del medidor del tren de combustible (Diesel o inyección directa de gasolina) , 0 , 655,350 ,
       kPa , 10(256A+B)
00414
00415 float decodePresionMedidorCombustible(char *response){
00416     int dec = (int)strtol(response, NULL, 16);
00417     return 10*dec;
00418 }
00419
00420 //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
00421
00422 struct OxigenoResponse decodeRelacionCombAire(char *response){
00423     char ABResponse[5], CDResponse[5];
00424     float AB, CD;
00425     struct OxigenoResponse datos;
00426     //Añade caracter \0 al final de la cadena AResponse
00427     memset(ABResponse, '\0', sizeof(ABResponse));
00428     //Divide los bytes de respuesta
00429     strncpy( ABResponse, response, 4);
00430     strcpy( CDResponse, response + 4 );
00431
00432     AB = (int)strtol(ABResponse, NULL, 16);
00433     datos.A = AB/32768;
00434     CD = (int)strtol(CDResponse, NULL, 16);
00435     datos.B = CD/8192;
00436
00437     return datos;
00438 }
00439
00440 //2c - EGR comandado , 0 , 100 , % , A/2.55
00441
00442 //decodeCargaPosicionEGR
00443
00444 //2d - falla EGR , -100 , 99.2 , % , A/1.28-100
00445
00446 //decodeAjusteCombustibleEGR(
00447
00448 //2e - Purga evaporativa comandada , 0 , 100 , % , A/2.55
00449
00450 //decodeCargaPosicionEGR
00451
00452 //2f - Nivel de entrada del tanque de combustible , 0 , 100 , % , A/2.55
00453
00454 //decodeCargaPosicionEGR
00455
00456 //30 - Cantidad de calentamientos desde que se borraron los fallas , 0 , 255 , cuenta, A
00457
00458 //decodeHexToDec
00459
00460 //31 - Distancia recorrida desde que se borraron los fallas , 0 , 65,535 , km , 256A+B
00461
00462 //decodeHexToDec
00463
00464 //32 - Presión de vapor del sistema evaporativo , -8,192 , 8191.75 , Pa , (256A + B) / 4 - 8192
00465
00466 float decodePresionVapor(char *response){
00467     int dec = (int)strtol(response, NULL, 16);
00468     return (dec/4.0)-8192.0;
00469 }
00470
00471 //33 - Presión barométrica absoluta , 0 , 255 , kPa , A
00472
00473 //decodeHexToDec
00474
00475 //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
00476
00477 struct OxigenoResponse decodeRelacionCombAireActual(char *
      response){
00478     char ABResponse[5], CResponse[3], DResponse[3];
00479     float AB, C, D;
00480     struct OxigenoResponse datos;
00481     //Añade caracter \0 al final de la cadena AResponse
00482     memset(ABResponse, '\0', sizeof(ABResponse));
00483     memset(CResponse, '\0', sizeof(CResponse));
00484     //Divide los bytes de respuesta
00485     strncpy( ABResponse, response, 4);
00486     strncpy( CResponse, response + 4, 2);
00487     strcpy( DResponse, response + 6);
00488
00489     AB = (int)strtol(ABResponse, NULL, 16);
00490     datos.A = AB/32768;
00491     C = (int)strtol(CResponse, NULL, 16);
00492     D = (int)strtol(DResponse, NULL, 16);
00493     datos.B = C+(D/256)-128;
00494
00495     return datos;
```

```
00496 }
00497
00498 //3c,3d,3e,3f - Temperatura del catalizador: Banco 1-4, Sensor , -40 , 6,513.5 , °C , (256A+B)/10-40
00499
00500 float decodeTempCatalizador(char *response){
00501     int dec = (int)strtol(response, NULL, 16);
00502     return (dec/10.0)-40;
00503 }
00504
00505 //40 - PID implementados [41 - 60]
00506 //41 - Estado de los monitores en este ciclo de manejo
00507 //42 - Voltaje del módulo de control , 0 , 65.535 , V , (256A+B)/1000
00508
00509 float decodeVoltajeControl(char *response){
00510     int dec = (int)strtol(response, NULL, 16);
00511     return dec/1000.0;
00512 }
00513
00514 //43 - Valor absoluta de carga , 0 , 25,700 , % , (256A+B)/2.55
00515
00516 //decodeCargaPosicionEGR
00517
00518 //44 - Relación equivaliente comandada de combustible - aire , 0 , <2 , prop. , (256A+B)/32768
00519
00520 float decodeRelacionCombAireBasica(char *response){
00521     int dec = (int)strtol(response, NULL, 16);
00522     return dec/32768.0;
00523 }
00524
00525 //45 - Posición relativa del acelerador , 0 , 100 , % , A/2.55
00526
00527 //decodeCargaPosicionEGR
00528
00529 //46 - Temperatura del aire ambiental , -40 , 215 , °C , A-40
00530
00531 //decodeTempGeneral
00532
00533 //47,48,49,4a,4b 4c - Posición absoluta del acelerador B,C,D,E,F  Actuador comandando del acelerador , 0 ,
       100 , % , A/2.55
00534
00535 //decodeCargaPosicionEGR
00536
00537 //4d, 4e - Tiempo transcurrido con MIL encendido Tiempo transcurrido desde que se borraron los códigops de
       fallas , 0 , 65,535 , min , 256A+B
00538
00539 /*4f Valor máximo de la relación de equivalencia de combustible - aire,
00540  voltaje del sensor de oxígenos, corriente del sensor de oxígenos
00541   y presión absoluta del colector de entrada , 0 0 0 0 , 255 255 255 2550, prop. V mA kPa, A B C D*10
00542 */
00543
00544 struct RelacionesResponse decodeRelaciones(char *response){
00545     char AResponse[3], BResponse[3], CResponse[3], DResponse[3];
00546     struct RelacionesResponse datos;
00547     //Añade caracter \0 al final de la cadena AResponse
00548     memset(AResponse, '\0', sizeof(AResponse));
00549     memset(BResponse, '\0', sizeof(BResponse));
00550     memset(CResponse, '\0', sizeof(CResponse));
00551     //Divide los bytes de respuesta
00552     strncpy( AResponse, response, 2);
00553     strncpy( BResponse, response + 2, 2);
00554     strncpy( CResponse, response + 4, 2);
00555     strcpy( DResponse, response + 6);
00556
00557     datos.A = (int)strtol(AResponse, NULL, 16);
00558     datos.B = (int)strtol(BResponse, NULL, 16);
00559     datos.C = (int)strtol(CResponse, NULL, 16);
00560     datos.D = 10*(int)strtol(DResponse, NULL, 16);
00561
00562     return datos;
00563 }
```

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:

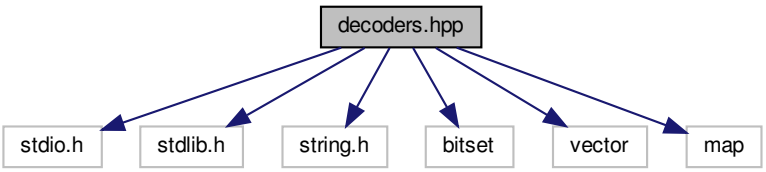
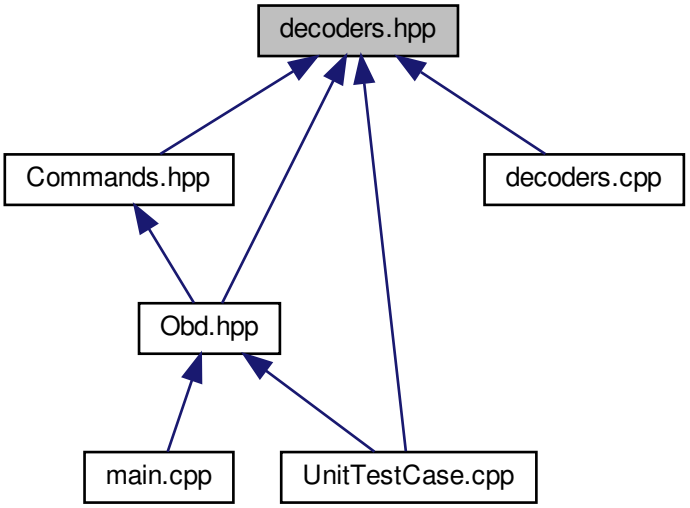


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



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) para ISO15765-4 CAN.
- std::string decodeDescribeProtocol (char *response)

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

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

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

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

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

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

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

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

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

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

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

  Función de decodificación de la presión de vapor del sistema evaporativo .
- float decodeTempCatalizador (char *response)

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

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

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

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

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

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

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

5.11.1. 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 71 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 308 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 340 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 294 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 110 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 322 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 136 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 408 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 315 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 415 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 466 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 422 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 477 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 520 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 544 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 329 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 365 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 152 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 500 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 301 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 351 del archivo decoders.cpp.

5.11.3.23. decodeVIN()

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

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

Parámetros

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

Devuelve

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

Definición en la línea 21 del archivo decoders.cpp.

5.11.3.24. 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 509 del archivo decoders.cpp.

5.12. decoders.hpp

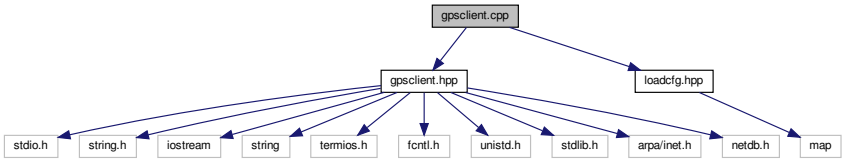
```
00001
00008 #ifndef DECODERS_HPP
00009 #define DECODERS_HPP
00010
00011 #include <stdio.h>
00012 #include <stdlib.h>
00013 #include <string.h>
00014 #include <bitset>
00015 #include <vector>
00016 #include <map>
00017
00018 #define PID_BITS 32
00019 #define STATUS_BITS 8
00025 struct OxigenoResponse {
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 gpsclient.cpp

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

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

Dependencia gráfica adjunta para gpsclient.cpp:



defines

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

5.13.1. Descripción detallada

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

Autor

Juan Manuel Vozmediano Torres

Fecha

09/08/2019

Definición en el archivo gpsclient.cpp.

5.13.2. Documentación de los 'defines'

5.13.2.1. NOGPSDATA

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

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

Definición en la línea 11 del archivo gpsclient.cpp.

5.14. gpsclient.cpp

```
00001
00008 #include "gpsclient.hpp"
00009 #include "loadcfg.hpp"
00010
00011 #define NOGPSDATA  "000000,010170,0000.000,N,00000.000,E"
00013 GpsClient::GpsClient (std::string GpsPort,
00014                       std::string validity):
00015   validity_(atoi(validity.c_str()))
00016 {
00017
00018   fprintf(stderr, "Init GpsClient %s %s\n", GpsPort.c_str(), validity.c_str());
00019   if ((s_= socket (AF_INET, SOCK_DGRAM, 0)) < 0)
00020     shit ("socket");
00021
00022   memset ((char *)&gpsin_, 0, sizeof(struct sockaddr_in));
00023   gpsin_.sin_family      = AF_INET;
00024   gpsin_.sin_addr.s_addr = inet_addr("127.0.0.1");
00025   gpsin_.sin_port        = htons(atoi(GpsPort.c_str()));
00026
00027   memset ((char *)&iyo_, 0, sizeof(struct sockaddr_in));
00028   iyo_.sin_family      = AF_INET;
00029   iyo_.sin_addr.s_addr = INADDR_ANY;
00030   iyo_.sin_port        = 0;
00031
00032   if (bind(s_, (struct sockaddr *)&iyo_, sizeof(iyo_)) == -1)
00033     shit("bind");
00034 }
00035
00036
00037 std::string GpsClient::getGPS()
00038 {
00039
00040   const int maxbuf = 512;
00041   char buffer [maxbuf];
00042   char msg[]    = "hhmmss,ddmmyy,llll.lll,N,yyyyy.yyy,E\n";
00043   struct timeval tv;
00044   int len = strlen(msg);
00045
00046   memset(msg, 0, len);
00047   // Asks
00048   /*
00049   int cc =  sendto(s_,
00050                    buffer,
00051                    maxbuf,
00052                    0,
00053                    (struct sockaddr *)&gpsin_,
00054                    sizeof(gpsin_));
00055                    */
00056   sendto(s_, buffer, maxbuf, 0, (struct sockaddr *)&gpsin_, sizeof(gpsin_));
00057
00058   fd_set rfds;
00059   tv.tv_sec  = 2;
00060   tv.tv_usec = 0;
00061   FD_ZERO(&rfds);
00062   FD_SET(s_, &rfds);
00063   int ret = select(s_+1, &rfds, NULL, NULL, &tv);
00064
00065   if ((ret == 0)||(!FD_ISSET(s_, &rfds))) { /* timeout, return invalid data */
00066         time_t lt;
00067         //struct tm *p;
00068         time(&lt);
00069         strftime(msg, len, "%H%M%S,%d%m%y,,,,", localtime(&lt));
00070   }
00071   else {
00072         struct sockaddr_in rv;
00073         socklen_t          addrlen = sizeof(struct sockaddr_in);
00074         (void) recvfrom(s_, msg, len, 0, (struct sockaddr *)&rv, &addrlen);
00075   }
00076   return std::string(msg);
00077 }
00078
```

5.15. Referencia del Archivo gpsclient.hpp

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

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

Dependencia gráfica adjunta para gpsclient.hpp:

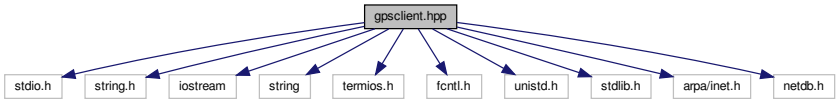
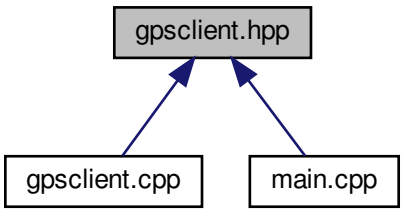


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- class GpsClient

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

5.15.1. Descripción detallada

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

Autor

Juan Manuel Vozmediano Torres

Fecha

09/08/2019

Definición en el archivo gpsclient.hpp.

5.16. gpsclient.hpp

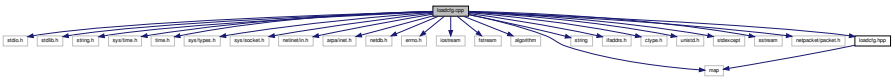
```
00001
00008 #ifndef GPSCLIENT_HPP
00009 #define GPSCLIENT_HPP
00010
00011 #include <stdio.h>
00012 #include <string.h>
00013 #include <iostream>
00014 #include <string>
00015 #include <termios.h>
00016 #include <fcntl.h>
00017 #include <unistd.h>
00018 #include <stdlib.h>
00019 #include <arpa/inet.h>
00020 #include <netdb.h>
00021
00028 class GpsClient {
00029
00030  private:
00031   int validity_;
00032   struct sockaddr_in gpsin_;
00033   struct sockaddr_in iyo_;
00034   int s_;
00036  public:
00037
00045   GpsClient (std::string GpsPort, std::string validity);
00046
00052   std::string getGPS ();
00053 };
00054
00055 #endif
```

5.17. 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.

5.17.1. 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.

5.17.2. Documentación de las funciones

5.17.2.1. getmac()

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

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

Parámetros

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

Devuelve

String de la MAC de la interfaz de red indicada.

Definición en la línea 60 del archivo loadcfg.cpp.

5.17.2.2. 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.

5.17.2.3. 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.

5.18. 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
00060 std::string getmac (const char* name)
00061 {
00062    int i;
00063    struct ifaddrs *addrs,*tmp;
00064    std::stringstream macaddress;
00065    char mymac[25];
00066    getifaddrs(&addrs);
00067    tmp = addrs;
00068    memset (mymac, 0, 25);
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 }
```

5.19. Referencia del Archivo loadcfg.hpp

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

`#include <map>`

Dependencia gráfica adjunta para loadcfg.hpp:

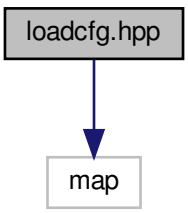
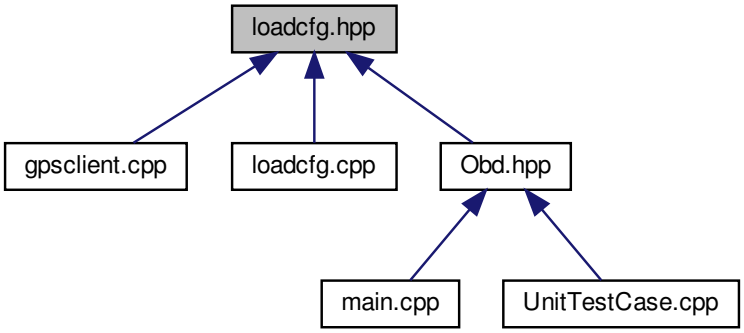


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



typedefs

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

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

Funciones

- void shit (const char *mens)

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

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

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

5.19.1. 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.

5.19.2. Documentación de las funciones

5.19.2.1. getmac()

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

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

Parámetros

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

Devuelve

String de la MAC de la interfaz de red indicada.

Definición en la línea 60 del archivo loadcfg.cpp.

5.19.2.2. 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.

5.19.2.3. 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.

5.20. loadcfg.hpp

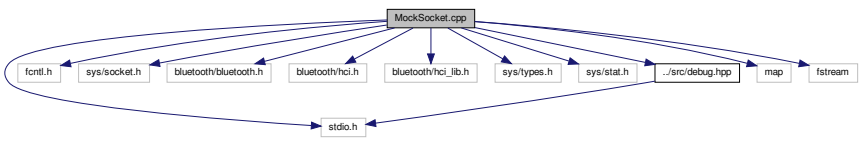
```
00001
00008 #ifndef LOADCFG_HPP
00009 #define LOADCFG_HPP
00010
00011 #include <map>
00012
00017 typedef  std::map <std::string, std::string> cfgType;
00018
00024 void shit (const char* mens);
00025
00032 void loadCfg (const char* filename, cfgType* pcfg);
00033
00040 std::string getmac (const char* name);
00041
00042 #endif
```

5.21. Referencia del Archivo MockSocket.cpp

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

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "../src/debug.hpp"
#include <map>
#include <fstream>
```

Dependencia gráfica adjunta para MockSocket.cpp:



Funciones

- std::string findDevPTS ()

  Función de detección del dispositivo para la simulación con OBDSIM.
- int **mock_socket** (int domain, int type, int protocol)
- int **hci_get_route** (bdaddr_t ∗bdaddr)
- int **hci_open_dev** (int dev_id)
- int **hci_inquiry** (int dev_id, int len, int max_rsp, const uint8_t ∗lap, inquiry_info ∗∗ii, long flags)
- int **hci_read_remote_name** (int sock, const bdaddr_t ∗ba, int len, char ∗name, int timeout)
- int **connect** (int sockfd, const struct sockaddr ∗addr, socklen_t addrlen)
- void **writeSocket** ()

5.21.1. 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.

5.21.2. Documentación de las funciones

5.21.2.1. findDevPTS()

```
std::string findDevPTS ( )
```

Función de detección del dispositivo para la simulación con OBDSIM.

Devuelve

String con la ruta del dispositivo al que conectarse para la simulación OBDSIM.

Definición en la línea 27 del archivo MockSocket.cpp.

5.22. 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 #include <map>
00020 #include <fstream>
00021
00027 std::string findDevPTS(){
00028     if(system("ls /dev/pts | tail -2 | head -1 > tmpPTSfile.txt") == -1){
00029         perror("Error ejecutando comando ");
00030     }
00031
00032     int ultPts;
00033     int tempVar;
00034     std::ifstream input_file("tmpPTSfile.txt");
00035     while ( input_file » tempVar )
00036     {
00037         ultPts = tempVar;
00038     }
```

```
00039
00040     std::string devFile = "/dev/pts/" + std::to_string(ultPts);
00041     remove("tmpPTSfile.txt");
00042
00043     return devFile;
00044 }
00045
00046 int mock_socket(int domain, int type, int protocol){
00047
00048     std::string devFile = findDevPTS();
00049
00050     int filedesc = open(devFile.c_str(), O_RDWR);
00051     if (filedesc < 0) {
00052         debugError("Error al abrir socket %d.", filedesc);
00053     }
00054
00055     return filedesc;
00056 }
00057
00058 int hci_get_route( bdaddr_t *bdaddr ){
00059     int value = 0;
00060     return value;
00061 }
00062
00063 int hci_open_dev( int dev_id ){
00064     int filedesc = open("/dev/null", O_RDONLY);
00065     if(filedesc < 0)
00066         filedesc = -1;
00067     return filedesc;
00068 }
00069
00070 int hci_inquiry(int dev_id, int len, int max_rsp, const uint8_t *lap, inquiry_info **ii, long flags){
00071     int value = 1;
00072     return value;
00073 }
00074
00075
00076 int hci_read_remote_name(int sock, const bdaddr_t *ba, int len, char *name, int timeout){
00077     int value = 1;
00078     strcpy(name, "OBDII");
00079     return value;
00080 }
00081
00082 int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen){
00083     int value = 0;
00084     return value;
00085 }
00086
00087 void writeSocket(){
00088
00089 }
```

5.23. Referencia del Archivo Obd.hpp

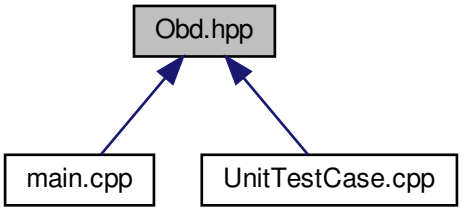
Archivo que contiene la clase con la implementación de la conexión y envío de mensajes OBD con el dispositivo ELM327.

```
#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.

5.23.1. 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.

5.23.2. Documentación de los 'defines'

5.23.2.1. 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.

5.24. 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                 perror("error");
00191                 throw std::string("No se ha podido conectar");
00192             }
00193
00194             debugLog("Conectado!");
00195             this->m_status = true;
00196
00197             //Creamos instancia epoll para la recepción de datos en el socket
00198             this->epoll_fd = epoll_create(1);
00199             if (this->epoll_fd < 0) {
00200                 perror("No se ha podido crear epoll");
00201                 close(this->m_cli_s);
00202             }
00203
00204             this->ev.events = EPOLLIN;
00205             this->ev.data.fd = this->m_cli_s;
00206
00207             //Añadimos el socket de conexión a la instancia de epoll creada
00208             int err = epoll_ctl(this->epoll_fd, EPOLL_CTL_ADD, this->m_cli_s, &ev);
00209
00210             if (err) {
00211                 perror("No se ha podido añadir el socket cliente a la instancia epoll");
00212                 close(this->m_cli_s);
00213                 close(this->epoll_fd);
00214             }
00215
00216
00217         } catch(std::string e) {
00218             std::cerr « e « std::endl;
00219         }
00220     }
00221
00229     void readFileData(){
00230         std::ifstream ifs("data/PIDS.json");
00231         auto j = json::parse(ifs);
00232
00233         //Convertimos todos los PIDS en objetos del tipo Commands y los añadimos a Obd
00234         for (int i = 0; i < (int)j.size(); ++i)
00235         {
00236             this->map_commands.insert(tupla(j[i]["name"], Commands(j[i])));
00237         }
00238     }
00239
00249     void send(Commands command){
00250         //Iniciamos en un hilo de ejecución la función polling de recepción de datos
00251         std::thread t1(&Obd::polling, this, command);
00252
00253         char *p;
00254         char buf[1024];
00255         int len;
00256
00257         //Comando a enviar
00258         std::string message = command.getCMD();
00259         strcpy(buf, message.c_str());
00260
00261         len = strlen(buf);
00262         buf[len] = '\n';
00263         buf[len+1] = '\0';
00264
00265         // Todo los mensajes a ELM327  deben terminar con el caracter retorno de carro (hex  '0D', \r).
00266         p = buf;
00267         while (*p) {
00268             if (*p == '\n')
```

```
00269                 *p = '\r';
00270             p++;
00271         }
00272 
00273         debugLog("Mensaje a enviar: %s", buf);
00274         debugLog("Enviando mensaje...");
00275         if(write(this->m_cli_s, buf, strlen(buf)) != (ssize_t) strlen(buf)){
00276             debugError("Error enviando mensaje.");
00277         }
00278 
00279         //Queda a la espera de finalización de ejecución del hilo de recepción del mensaje OBD
00280         t1.join();
00281     }
00282 
00295     void polling(Commands command){
00296         struct epoll_event events[MAX_EP_EVTS];
00297         int nfds;
00298         bool continuar = true;
00299 
00300         //debugLog("Polling function");
00301 
00302         // Bucle infinito para el envío de datos por bluetooth al conector OBD
00303         while(continuar) {
00304         // Buffer para enviar y recibir
00305             char message_rcv[1024], buf[1024], *p;
00306             ssize_t len;
00307             int i;
00308             //Quedamos a la espera de recepción de eventos en la instancia epoll (socket)
00309             nfds = epoll_wait(this->epoll_fd, events, MAX_EP_EVTS, -1);
00310             if (nfds < 0) {
00311                 perror("epoll error");
00312                 break;
00313             }
00314             for (i = 0; i < nfds; i++) {
00315                 if ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP)) {
00316                     debugError("epoll error");
00317                 }
00318                 //Si los eventos detectados corresponden al socket de conexión con el vehículo, tratamos el
      mensaje
00319                 if (events[i].data.fd == this->m_cli_s) {
00320                     len = read(this->m_cli_s, &buf, sizeof(buf) - 1);
00321                     if (len < 0) {
00322                         perror("socket read error");
00323                         continue;
00324                     }
00325                     //debugLog("Evento leído: %s", buf);
00326                     strcat(message_rcv, buf);
00327                     //Si se detecta el caracter ">" se ha finalizado el mensaje
00328                     if(strstr(buf, ">") != NULL) {
00329                         len = strlen(message_rcv);
00330                         message_rcv[len] = '\0';
00331 
00332                         p = message_rcv;
00333                         //Conversión inversa del mensaje ELM327 enviado en el último carácter
00334                         while(*p) {
00335                             if (*p == '\r')
00336                                 *p = '\n';
00337                             p++;
00338                         }
00339                         //Transformar respuesta
00340                         debugLog("Mensaje recibido:\n%s", message_rcv);
00341 
00342                         char * ocurrencia = message_rcv;
00343                         if((ocurrencia=strstr(ocurrencia, command.getCMDResponse().c_str()))
      != NULL){
00344                             while((ocurrencia=strstr(ocurrencia, command.
      getCMDResponse().c_str())) != NULL){
00345                                 //debugLog("Ocurrencia encontrada");
00346                                 char info[1024];
00347                                 memset(info, '\0', sizeof(info));
00348                                 strncpy(info, ocurrencia + command.getCMD().size() , command.
      getBytesResponse());
00349                                 debugLog("Información: %s", info);
00350                                 std::string type_data = command.getTypeData();
00351                                 //Dependiendo del tipo de dato de la respuesta se busca el decodificador
      correspondiente
00352                                 if (!type_data.compare("float")){
00353                                     auto varResultado = this->decoderFunctionsFloat[command.
      getDecoder().c_str()](info);
00354                                     std::cout « command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00355                                     std::cout « "-> " « varResultado « " "« command.
      getUnits() « std::endl;
00356                                     this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00357                                 } else if(!type_data.compare("OxigenoResponse")){
```

```
00358                                   auto varResultado = this->decoderFunctionsStructOx[command.
      getDecoder().c_str()](info);
00359                                   std::cout « command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00360                                   std::cout « "-> " « varResultado.A « "/" « varResultado.B « " "«
      command.getUnits() « std::endl;
00361                                   this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00362
00363                               } else if (!type_data.compare("RelacionesResponse")) {
00364                                   auto varResultado = this->decoderFunctionsStructRel[command.
      getDecoder().c_str()](info);
00365                                   std::cout « command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00366                                   std::cout « "-> " « varResultado.A « "/" « varResultado.B « "/" «
       varResultado.C « "/" « varResultado.D « " "« command.getUnits() « std::endl;
00367                                   this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00368
00369                               } else if(!type_data.compare("vectorInt")){
00370                                   auto varResultado = this->decoderFunctionsVectorInt[command.
      getDecoder().c_str()](info);
00371                                   this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00372
00373                                   std::cout « command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00374                                   //Tratamiento para los PIDs disponibles
00375                                   for (uint32_t i = 0; i < varResultado.size(); ++i){
00376                                       std::string substr_cmd = command.getCMD().substr(2,2);
00377                                       int sum_pid = stoi(substr_cmd,nullptr,16);
00378                                       std::stringstream stream;
00379                                       stream « std::hex « sum_pid+varResultado[i];
00380                                       std::string result(stream.str());
00381                                       if(result.size() == 1)
00382                                       //Si el resultado solo tiene un caracter se añade un 0 al principio
00383                                           result.insert(0,"0");
00384                                       result.insert(0,"01");
00385                                       std::transform(result.begin(), result.end(),result.begin(),
      ::toupper);
00386
00387                                       //Almacenamos el resultado de los PIDs disponibles
00388                                       this->vecPIDs.push_back(result);
00389                                   }
00390                               } else if (!type_data.compare("vectorStr")) {
00391                                   auto varResultado = this->decoderFunctionsVectorStr[command.
      getDecoder().c_str()](info);
00392                                   this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00393                                   //Decodificador para DTC
00394                                   if (varResultado.empty()){
00395                                       debugLog("No hay DTC en el vehículo");
00396                                   } else {
00397                                       this->vecDTCs = varResultado;
00398                                       for (uint32_t i = 0; i < varResultado.size(); ++i)
00399                                       {
00400                                           debugLog("Enviar DTC: %s", varResultado[i].c_str());
00401                                       }
00402                                   }
00403                               } else if (!type_data.compare("string")) {
00404                                   auto varResultado = this->decoderFunctionsStr[command.
      getDecoder().c_str()](info);
00405                                   this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00406                                   //Decodificador para el número de identificación del vehículo
00407                                   if(!command.getDecoder().compare("decodeVIN"))
00408                                       this->vin.append(varResultado);
00409                                   std::cout « command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00410                                   std::cout « "-> " « varResultado « std::endl;
00411                               } else if (!type_data.compare("map")) {
00412                                   auto varResultado = this->decoderFunctionsMap[command.
      getDecoder().c_str()](info);
00413                                   this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00414                                   this->mapStatus = varResultado;
00415                               } else {
00416                                   debugLog("Tipo de dato no reconocido");
00417                               }
00418                               ocurrencia++;
00419                           }
00420                           std::cout « "------------------------------------------------------------" <
      < std::endl;
```

```
00421                             memset(message_rcv, '\0', sizeof(message_rcv));
00422                             continuar = false;
00423                             //Respuestas de mensajes de AT de configuración
00424                         } else if((strstr(message_rcv, "OK")) != NULL){
00425                             debugLog("%s = OK.", command.getDescription().c_str());
00426                             memset(message_rcv, '\0', sizeof(message_rcv));
00427                             continuar = false;
00428                             //Vehículo sin el dato solicitado
00429                         } else if((strstr(message_rcv, "NO DATA")) != NULL){
00430                             debugLog("%s = No disponible.", command.
      getDescription().c_str());
00431                             memset(message_rcv, '\0', sizeof(message_rcv));
00432                             continuar = false;
00433                         } else {
00434                             //Para conocer el protocolo actual
00435                             if(!command.getName().compare("DESCRIBE_PROTOCOL")){
00436                                 char info[1024];
00437                                 char* token = strtok(message_rcv, "\n");
00438                                 strcpy(info, token);
00439                                 auto varResultado = this->decoderFunctionsStr[command.
      getDecoder().c_str()](info);
00440                                 this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00441                                 this->currentProtocol = varResultado;
00442                             }else if(!command.getName().compare("DESCRIBE_PROTOCOL_NUMBER")){
00443                                 char info[1024];
00444                                 char* token = strtok(message_rcv, "\n");
00445                                 strcpy(info, token);
00446                                 auto varResultado = this->decoderFunctionsStr[command.
      getDecoder().c_str()](info);
00447                                 this->map_commands.find(command.getName())->second.setResValue(
      varResultado);
00448                                 this->currentProtocolNumber = varResultado;
00449                             } else {
00450                                 debugLog("Mensaje recibido no entendido!");
00451                             }
00452                             memset(message_rcv, '\0', sizeof(message_rcv));
00453                             continuar = false;
00454                         }
00455                     }
00456 
00457                     memset(buf, '\0', sizeof(buf));
00458                 } else {
00459                     debugError("Evento desconocido");
00460                 }
00461             }
00462         }
00463     }
00464 
00476     void initMessages(){
00477         //Inicialización de la conexión con ELM327
00478         std::map<std::string, std::string> listPIDs = {
00479             {"PIDS_B", "0120"},
00480             {"PIDS_C", "0140"},
00481             {"PIDS_D", "0160"},
00482             {"PIDS_E", "0180"},
00483             {"PIDS_F", "01A0"},
00484             {"PIDS_G", "01C0"}
00485         };
00486         this->send(this->map_commands.find("RESET")->second);
00487         this->send(this->map_commands.find("DEFAULT_VALUES")->second);
00488         this->send(this->map_commands.find("RESP_SIN_ESPACIOS")->second);
00489         this->send(this->map_commands.find("SIN_ECO")->second);
00490         this->send(this->map_commands.find("SIN_HEADER")->second);
00491         this->send(this->map_commands.find("AUTO_PROTO")->second);
00492         this->send(this->map_commands.find("STATUS")->second);
00493         this->send(this->map_commands.find("GET_VIN")->second);
00494         this->send(this->map_commands.find("PIDS_A")->second);
00495         //Bucle para detectar PIDs disponibles
00496         for (std::map<std::string, std::string>::iterator it=listPIDs.begin(); it!=listPIDs.end(); ++it){
00497             if(this->existPID(it->second)){
00498                 this->send(this->map_commands.find(it->first)->second);
00499             }
00500         }
00501         debugLog("N° de comandos disponibles = %zu", vecPIDs.size());
00502     }
00503 
00511     void initDecoderFunctions(){
00512     //Inicia las funciones dependiendo del tipo de dato de respuesta
00513     this->decoderFunctionsFloat = {
00514             { "decodeCargaPosicionEGR", decodeCargaPosicionEGR},
00515             { "decodeTempGeneral", decodeTempGeneral},
00516             { "decodeAjusteCombustibleEGR", decodeAjusteCombustibleEGR},
00517             { "decodePresionCombustible", decodePresionCombustible},
00518             { "decodeHexToDec", decodeHexToDec},
00519             { "decodeRPM", decodeRPM},
00520             { "decodeAvanceTiempo", decodeAvanceTiempo},
```

```
00521             { "decodeVelocidadMAF", decodeVelocidadMAF},
00522             { "decodePresionCombColector", decodePresionCombColector},
00523             { "decodePresionMedidorCombustible", decodePresionMedidorCombustible
      },
00524             { "decodePresionVapor", decodePresionVapor},
00525             { "decodeTempCatalizador", decodeTempCatalizador},
00526             { "decodeVoltajeControl", decodeVoltajeControl},
00527             { "decodeRelacionCombAireBasica", decodeRelacionCombAireBasica}
00528         };
00529         this->decoderFunctionsStructOx = {
00530             { "decodeSensorOxigeno", decodeSensorOxigeno},
00531             { "decodeRelacionCombAire", decodeRelacionCombAire},
00532             { "decodeRelacionCombAireActual", decodeRelacionCombAireActual}
00533         };
00534 
00535         this->decoderFunctionsStructRel["decodeRelaciones"] = decodeRelaciones;
00536         this->decoderFunctionsVectorInt["decodePIDS"] = decodePIDS;
00537         this->decoderFunctionsVectorStr["decodeDTCs"] = decodeDTCs;
00538         this->decoderFunctionsStr = {
00539             {"decodeVIN", decodeVIN},
00540             {"decodeDescribeProtocol", decodeDescribeProtocol}
00541         };
00542         this->decoderFunctionsMap["decodeStatus"] = decodeStatus;
00543         this->noDecodeFunctionAT["noDecodeAT"] = noDecodeAT;
00544     }
00545 
00552     void disconnectBluetooth(){
00553         debugLog("Desconectando dispositivo Bluetooth");
00554         close(this->m_cli_s);
00555         close(this->epoll_fd);
00556     }
00557 
00564     bool existPID(std::string command){
00565         bool exists = false;
00566         //Comprueba en la lista de PIDs que el comando este implementado
00567         for (uint32_t i = 0; i < this->vecPIDs.size(); ++i){
00568             if(!this->vecPIDs[i].compare(command)){
00569                 exists = true;
00570                 break;
00571             }
00572         }
00573         return exists;
00574     }
00575 
00583     void printPIDs(){
00584         //Iteración para imprimir por consola los PIDs disponibles
00585         for (std::map<std::string, Commands>::iterator it=this->map_commands.begin(); it!=this->
      map_commands.end(); ++it){
00586             Commands command = it->second;
00587             std::string str_cmd = command.getCMD();
00588             for (uint32_t i = 0; i < this->vecPIDs.size(); ++i){
00589                 if(!str_cmd.compare(this->vecPIDs[i])){
00590                     std::cout « str_cmd « " - "« command.getName() « " - " « command.
      getDescription() « " - Min=" « command.getMIN() « " Max=" « command.
      getMAX() « std::endl;
00591                     break;
00592                 }
00593             }
00594         }
00595     }
00596 
00603     void printStatus(){
00604         //Imprime por consola los resultados de las pruebas del comando STATUS
00605         for (std::map<std::string, std::string>::iterator it=this->mapStatus.begin(); it!=this->mapStatus.
      end(); ++it){
00606             std::cout « it->first « " -> " « it->second « std::endl;
00607         }
00608     }
00609 
00615     std::string getVIN(){
00616         //Devuelve el número de identificación del vehículo
00617         return this->vin;
00618     }
00619 
00628     std::vector<std::string> getDTCs(){
00629         time_t curr_time;
00630         tm *curr_tm;
00631         char date_string[100];
00632         char time_string[100];
00633         time(&curr_time);
00634 
00635         curr_tm = localtime(&curr_time);
00636         strftime(date_string, 50, "%d/%m/%Y", curr_tm);
00637         strftime(time_string, 50, "%T", curr_tm);
00638         std::cout « date_string « " " « time_string « std::endl;
00639 
00640         //Consulta de DTC del vehículo
```

```
00641         this->send(this->map_commands.find("STATUS")->second);
00642         if (this->mapStatus["DTC_CNT"].compare("0")){
00643             this->send(this->map_commands.find("GET_DTC")->second);
00644         } else {
00645             std::cout « "No hay DTC disponibles" « std::endl;
00646         }
00647         return this->vecDTCs;
00648     }
00649 
00655     bool isValid(){
00656         //Bool del estado de la conexión bluetooth
00657         return this->m_status;
00658     }
00659 private:
00660     // Atributos privados de la clase "Obd"
00661     std::vector<std::string> vecPIDs;
00662     std::vector<std::string> vecDTCs;
00663     std::string vin;
00664     std::string currentProtocol;
00665     std::string currentProtocolNumber;
00666     std::map<std::string, std::string> mapStatus;
00668     std::map<std::string, std::function<void()» noDecodeFunctionAT;
00669     std::map<std::string, std::function<float(char *)» decoderFunctionsFloat;
00670     std::map<std::string, std::function<struct OxigenoResponse(char *)» decoderFunctionsStructOx;
00671     std::map<std::string, std::function<struct RelacionesResponse(char *)» decoderFunctionsStructRel;
00672     std::map<std::string, std::function<std::vector<int>(char *)» decoderFunctionsVectorInt;
00673     std::map<std::string, std::function<std::vector<std::string>(char *)» decoderFunctionsVectorStr;
00674     std::map<std::string, std::function<std::string(char *)» decoderFunctionsStr;
00675     std::map<std::string, std::function<std::map<std::string, std::string>(char *)» decoderFunctionsMap;
00677     char dest[19] = { 0 };
00678     int m_cli_s;
00679     bool m_deviceFound = false;
00680     bool m_status = false;
00681     int epoll_fd;
00682     struct epoll_event ev;
00683 };
00684 
00685 
00686 #endif
```

5.25. 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 (std::string tipoSimulador, int tiempoEspera)

Función de inicialización del simulador OBDSIM.

- void closeOBDSIM ()

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

  Prueba de integración para el funcionamiento de la obtención de DTC y PIDS disponibles.
- TEST_CASE ("Test OBD class data SPEED", "[OBD]")

  Prueba de integración para el funcionamiento de solicitud de un dato continúo (velocidad).
- TEST_CASE ("Test Revoluciones Por Minuto", "[decoders]")

  Prueba unitaria del decodificador RPM.
- TEST_CASE ("Test Posición EGR", "[decoders]")

  Prueba unitaria del decodificador de posición EGR.
- TEST_CASE ("Test Temperatura General", "[decoders]")

  Prueba unitaria del decodificador de la temperatura general.
- TEST_CASE ("Test Ajuste Combustible EGR", "[decoders]")

  Prueba unitaria del decodificador de la temperatura general.
- TEST_CASE ("Test Presión Combustible", "[decoders]")

  Prueba unitaria del decodificador de la presión del combustible.
- TEST_CASE ("Test Hexadecimal a Decimal", "[decoders]")

  Prueba unitaria del decodificador hexadecimal a decimal.
- TEST_CASE ("Test Avance Tiempo", "[decoders]")

  Prueba unitaria del decodificador del avance del tiempo.
- TEST_CASE ("Test Velocidad Flujo Aire MAF", "[decoders]")

  Prueba unitaria del decodificador de la tasa de flujo del aire (MAF).
- TEST_CASE ("Test Presión del Combustible, relativa al colector de vacío", "[decoders]")

  Prueba unitaria del decodificador de presión de combustible relativa al colector de vacío.
- TEST_CASE ("Test Presión del Combustible (Diesel o inyección directa de gasolina)", "[decoders]")

  Prueba unitaria del decodificador de presión de combusible (Diese o inyección directa de gasolina).
- TEST_CASE ("Test Presión de Vapor del Sistema Evaporativo", "[decoders]")

  Prueba unitaria del decodificador de Vapor del Sistema Evaporativo.
- TEST_CASE ("Test Temperatura del Catalizador", "[decoders]")

  Prueba unitaria del decodificador de Temperatura del Catalizador.
- TEST_CASE ("Test Voltaje del Módulo de Control", "[decoders]")

  Prueba unitaria del decodificador de Voltaje del Módulo de Control.
- TEST_CASE ("Test Relación Equivaliente Comandada de Combustible", "[decoders]")

  Prueba unitaria del decodificador de Relación Equivaliente Comandada de Combustible.
- TEST_CASE ("Comprobación Diagnostic Trouble Codes", "[DTC]")

  Prueba unitaria del conversor del primer byte DTC.
- TEST_CASE ("Test VIN (Vehicle Identification Number) ISO15765-4 CAN", "[decoders]")

  Prueba unitaria del Número de Identificación del vehículo.
- TEST_CASE ("Test Describir el Protocolo Actual", "[decoders]")

  Prueba unitaria del decodificador de descriptor del protocolo actual.
- SCENARIO ("Test de Sensores de Oxígeno", "[decoders]")

  Escenario de pruebas con distintos test de los sensores de oxígeno.
- SCENARIO ("Test de Relación Equivalente Combustible-Aire", "[decoders]")

  Escenario de pruebas con distintos test de la relación equivalente combustible-aire.
- SCENARIO ("Test de Relación Equivalente Combustible-Aire Actual", "[decoders]")

  Escenario de pruebas con distintos test de la relación equivalente combustible-aire actual.
- SCENARIO ("Test de Valores máximos relación de combustible-aire, voltaje, corriente y presión absoluta", "[decoders]")

  Escenario de pruebas con distintos test de valores máximo relación de combustible-aire, voltaje, corriente y presión absoluta.

- SCENARIO ("Test de decodificación de Data Trouble Codes (DTC)", "[decoders]")

  Escenario de pruebas con distintos test de DTC activos.
- SCENARIO ("Test de decodificación PIDs disponibles", "[decoders]")

  Escenario de pruebas con distintos test los PIDS implementados en el vehículo.
- SCENARIO ("Test de decodificación del estado del coche", "[decoders]")

  Escenario de pruebas con distintos valores de los monitores de diagnóstico tras las pruebas del vehículo.

5.25.1. 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.

5.25.2. Documentación de los 'defines'

5.25.2.1. 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.

5.25.2.2. 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.

5.25.2.3. WAIT_OBDSIM

```
#define WAIT_OBDSIM 15
```

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

Definición en la línea 16 del archivo UnitTestCase.cpp.

5.25.3. Documentación de las funciones

5.25.3.1. getMinicomCMD()

```
void getMinicomCMD (
            char * cmd )
```

Función que obtiene el comando Minicom para la primera conexión con el simulador OBDSIM.

Parámetros

cmd	Puntero a cadena de caracteres para almacenar el comando de minicom a ejecutar.

Definición en la línea 27 del archivo UnitTestCase.cpp.

5.25.3.2. initOBDSIM()

```
void initOBDSIM (
            std::string tipoSimulador,
            int tiempoEspera )
```

Función de inicialización del simulador OBDSIM.

Parámetros

tipoSimulador	String para indicar el tipo de simulador a iniciar.
tiempoEspera	Entero con el número de segundos de espera del simulador.

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

Definición en la línea 47 del archivo UnitTestCase.cpp.

5.26. 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
00021
00027 void getMinicomCMD(char * cmd){
00028
00029     debugLog("Principio función getMinicomCMD.");
00030
00031     std::string devFile = findDevPTS();
00032
00033     debugLog("Dispositivo a conectarnos %s", devFile.c_str());
00034
00035     snprintf(cmd, BUFSIZE, "minicom -p %s &", devFile.c_str());
00036 }
00037
00047 void initOBDSIM(std::string tipoSimulador, int tiempoEspera){
00048
00049     char cmd[BUFSIZE];
00050
00051     debugLog("Ejecutamos OBDSIM.");
00052
00053     char cmdTipoSimulador[BUFSIZE];
00054
00055     snprintf(cmdTipoSimulador, BUFSIZE, "obdsim -g %s &", tipoSimulador.c_str());
00056
00057     if(system(cmdTipoSimulador) == -1){
00058         perror("Error ejecutando comando ");
00059     }
00060     // 5 segundos para configurar parámetros para el test
00061     //debugLog("Esperamos la introducción de valores.");
00062
00063     sleep(tiempoEspera);
00064
00065     debugLog("Llamamos a la función de Minicom.");
00066
00067     getMinicomCMD(cmd);
00068
00069     // Se abre terminal en el test con minicom, porque el primer mensaje obdsim no envía >
00070     // a nivel de código, pero si con un terminal con minicom
00071     //system("minicom -p /dev/pts/3 &");
00072     debugLog("Ejecutamos minicom.");
00073
00074     if(system(cmd) == -1){
00075         perror("Error ejecutando comando ");
00076     }
00077
00078     sleep(1);
00079
00080     debugLog("Matamos minicom.");
00081     // Se mata el proceso minicom, ya que, no es necesario
00082     if(system("pkill minicom") == -1){
00083         perror("Error ejecutando comando ");
00084     }
00085
00086     sleep(1);
00087 }
00088
00093 void closeOBDSIM(){
00094     if(system("pkill obdsim") == -1){
00095         perror("Error ejecutando comando ");
00096     }
00097 }
00098
00103 TEST_CASE( "Test OBD class DTC", "[OBD]" ) {
00104
00105     debugLog("Comenzando test OBD class DTC");
00106
00107     debugLog("Iniciamos el simulador OBDSIM");
00108     // Iniciamos el simulador OBDSIM para las pruebas
00109     initOBDSIM("gui_fltk", WAIT_OBDSIM);
00110
00111     debugLog("Iniciamos conexión OBD");
00112
00113     Obd connection = Obd("OBDII");
00114
00115     debugLog("Finalizado el proceso de inicio de conexión OBD");
00116
00117     REQUIRE (connection.isValid() == true);
00118
00119     connection.printStatus();
```

```
00120
00121     connection.getDTCs();
00122
00123     std::cout « connection.map_commands.find("GET_DTC")->second.getJson().dump(4) « std::endl
    ;
00124
00125     closeOBDSIM();
00126 }
00127
00132 TEST_CASE( "Test OBD class data SPEED", "[OBD]" ) {
00133
00134     debugLog("Comenzando test OBD class data SPEED");
00135
00136     debugLog("Iniciamos el simulador OBDSIM");
00137     // Iniciamos el simulador OBDSIM para las pruebas
00138     initOBDSIM("Cycle", 1);
00139
00140     debugLog("Iniciamos conexión OBD");
00141
00142     Obd connection = Obd("OBDII");
00143
00144     debugLog("Finalizado el proceso de inicio de conexión OBD");
00145
00146     REQUIRE (connection.isValid() == true);
00147
00148     connection.printPIDs();
00149
00150     for (int i = 0; i < 15; ++i){
00151         connection.send(connection.map_commands.find("SPEED")->second);
00152
00153         sleep(1);
00154
00155         std::cout « connection.map_commands.find("SPEED")->second.getJson().dump(4) «
    std::endl;
00156     }
00157
00158     closeOBDSIM();
00159 }
00160
00165 TEST_CASE( "Test Revoluciones Por Minuto", "[decoders]" ) {
00166     REQUIRE( decodeRPM((char *)"0000") == 0);
00167     REQUIRE( decodeRPM((char *)"FFFF") == Approx(16383.75).epsilon(0.01));
00168     REQUIRE( decodeRPM((char *)"7FFF") == Approx(8191.75).epsilon(0.01));
00169     REQUIRE( decodeRPM((char *)"12F2") == Approx(1212.5).epsilon(0.01));
00170 }
00171
00176 TEST_CASE( "Test Posición EGR", "[decoders]" ) {
00177     REQUIRE( decodeCargaPosicionEGR((char *)"00") == Approx(0).epsilon(0.01));
00178     REQUIRE( decodeCargaPosicionEGR((char *)"FF") == Approx(100).epsilon(0.01));
00179     REQUIRE( decodeCargaPosicionEGR((char *)"7F") == Approx(50).epsilon(0.01));
00180 }
00181
00186 TEST_CASE( "Test Temperatura General", "[decoders]" ) {
00187     REQUIRE( decodeTempGeneral((char *)"00") == Approx(-40).epsilon(0.01));
00188     REQUIRE( decodeTempGeneral((char *)"FF") == Approx(215).epsilon(0.01));
00189     REQUIRE( decodeTempGeneral((char *)"7F") == Approx(87).epsilon(0.01));
00190 }
00191
00196 TEST_CASE( "Test Ajuste Combustible EGR", "[decoders]" ) {
00197     REQUIRE( decodeAjusteCombustibleEGR((char *)"00") == Approx(-100).epsilon(0.0
    1));
00198     REQUIRE( decodeAjusteCombustibleEGR((char *)"FF") == Approx(99.2).epsilon(0.0
    1));
00199     REQUIRE( decodeAjusteCombustibleEGR((char *)"7F") == Approx(-0.78125).epsilon
    (0.01));
00200 }
00201
00206 TEST_CASE( "Test Presión Combustible", "[decoders]" ) {
00207     REQUIRE( decodePresionCombustible((char *)"00") == Approx(0).epsilon(0.01));
00208     REQUIRE( decodePresionCombustible((char *)"FF") == Approx(765).epsilon(0.01));
00209     REQUIRE( decodePresionCombustible((char *)"7F") == Approx(381).epsilon(0.01));
00210 }
00211
00216 TEST_CASE( "Test Hexadecimal a Decimal", "[decoders]" ) {
00217     REQUIRE( decodeHexToDec((char *)"00") == Approx(0).epsilon(0.01));
00218     REQUIRE( decodeHexToDec((char *)"FF") == Approx(255).epsilon(0.01));
00219     REQUIRE( decodeHexToDec((char *)"7F") == Approx(127).epsilon(0.01));
00220 }
00221
00226 TEST_CASE( "Test Avance Tiempo", "[decoders]" ) {
00227     REQUIRE( decodeAvanceTiempo((char *)"00") == Approx(-64).epsilon(0.01));
00228     REQUIRE( decodeAvanceTiempo((char *)"FF") == Approx(63.5).epsilon(0.01));
00229     REQUIRE( decodeAvanceTiempo((char *)"7F") == Approx(-0.5).epsilon(0.01));
00230 }
00231
00236 TEST_CASE( "Test Velocidad Flujo Aire MAF", "[decoders]" ) {
00237     REQUIRE( decodeVelocidadMAF((char *)"0000") == Approx(0).epsilon(0.01));
```

```
00238     REQUIRE( decodeVelocidadMAF((char *)"FFFF") == Approx(655.35).epsilon(0.01));
00239     REQUIRE( decodeVelocidadMAF((char *)"7FFF") == Approx(327.67).epsilon(0.01));
00240 }
00241
00246 TEST_CASE( "Test Presión del Combustible, relativa al colector de vacío", "[decoders]" ) {
00247     REQUIRE( decodePresionCombColector((char *)"0000") == Approx(0).epsilon(0.01))
      ;
00248     REQUIRE( decodePresionCombColector((char *)"FFFF") == Approx(5177.265).epsilon
      (0.01));
00249     REQUIRE( decodePresionCombColector((char *)"7FFF") == Approx(2588.593).epsilon
      (0.01));
00250 }
00251
00256 TEST_CASE( "Test Presión del Combustible (Diesel o inyección directa de gasolina)", "[decoders]" )
       {
00257     REQUIRE( decodePresionMedidorCombustible((char *)"0000") == Approx(0).
      epsilon(0.01));
00258     REQUIRE( decodePresionMedidorCombustible((char *)"FFFF") == Approx(65535
      0).epsilon(0.01));
00259     REQUIRE( decodePresionMedidorCombustible((char *)"7FFF") == Approx(32767
      0).epsilon(0.01));
00260 }
00261
00266 TEST_CASE( "Test Presión de Vapor del Sistema Evaporativo", "[decoders]" ) {
00267     REQUIRE( decodePresionVapor((char *)"0000") == Approx(-8192).epsilon(0.01));
00268     REQUIRE( decodePresionVapor((char *)"FFFF") == Approx(8191.75).epsilon(0.01));
00269     REQUIRE( decodePresionVapor((char *)"7FFF") == Approx(-0.25).epsilon(0.01));
00270 }
00271
00276 TEST_CASE( "Test Temperatura del Catalizador", "[decoders]" ) {
00277     REQUIRE( decodeTempCatalizador((char *)"0000") == Approx(-40).epsilon(0.01));
00278     REQUIRE( decodeTempCatalizador((char *)"FFFF") == Approx(6513.5).epsilon(0.01));
00279     REQUIRE( decodeTempCatalizador((char *)"7FFF") == Approx(3236.7).epsilon(0.01));
00280 }
00281
00286 TEST_CASE( "Test Voltaje del Módulo de Control", "[decoders]" ) {
00287     REQUIRE( decodeVoltajeControl((char *)"0000") == Approx(0).epsilon(0.01));
00288     REQUIRE( decodeVoltajeControl((char *)"FFFF") == Approx(65.535).epsilon(0.01));
00289     REQUIRE( decodeVoltajeControl((char *)"7FFF") == Approx(32.767).epsilon(0.01));
00290 }
00291
00296 TEST_CASE( "Test Relación Equivaliente Comandada de Combustible", "[decoders]" ) {
00297     REQUIRE( decodeRelacionCombAireBasica((char *)"0000") == Approx(0).epsilon(
      0.01));
00298     REQUIRE( decodeRelacionCombAireBasica((char *)"FFFF") == Approx(2).epsilon(
      0.01));
00299     REQUIRE( decodeRelacionCombAireBasica((char *)"7FFF") == Approx(1).epsilon(
      0.01));
00300 }
00301
00306 TEST_CASE( "Comprobación Diagnostic Trouble Codes", "[DTC]" ) {
00307     REQUIRE( convertDTCs("0123") == "P0123");
00308     REQUIRE( convertDTCs("1234") == "P1234");
00309     REQUIRE( convertDTCs("2345") == "P2345");
00310     REQUIRE( convertDTCs("3456") == "P3456");
00311     REQUIRE( convertDTCs("4567") == "C0567");
00312     REQUIRE( convertDTCs("5678") == "C1678");
00313     REQUIRE( convertDTCs("6789") == "C2789");
00314     REQUIRE( convertDTCs("789A") == "C389A");
00315     REQUIRE( convertDTCs("89AB") == "B09AB");
00316     REQUIRE( convertDTCs("9ABC") == "B1ABC");
00317     REQUIRE( convertDTCs("ABCD") == "B2BCD");
00318     REQUIRE( convertDTCs("BCDE") == "B3CDE");
00319     REQUIRE( convertDTCs("CDEF") == "U0DEF");
00320     REQUIRE( convertDTCs("DEF0") == "U1EF0");
00321     REQUIRE( convertDTCs("EF01") == "U2F01");
00322     REQUIRE( convertDTCs("F012") == "U3012");
00323 }
00324
00329 TEST_CASE( "Test VIN (Vehicle Identification Number) ISO15765-4 CAN", "[decoders]" ) {
00330     REQUIRE( decodeVIN((char *)"01573056\n1:3058455036384A\n2:34313430303530") == "
      W0V0XEP68J4140050");
00331     REQUIRE( decodeVIN((char *)"01314434\n1:47503030523535\n2:42313233343536") == "
      1D4GP00R55B123456");
00332 }
00333
00338 TEST_CASE( "Test Describir el Protocolo Actual", "[decoders]" ) {
00339     REQUIRE( decodeDescribeProtocol((char *)"AUTO, ISO 15765-4 (CAN 11/500)") == "
      AUTO, ISO 15765-4 (CAN 11/500)");
00340     REQUIRE( decodeDescribeProtocol((char *)"ISO 9141-2") == "ISO 9141-2");
00341 }
00342
00347 SCENARIO( "Test de Sensores de Oxígeno", "[decoders]" ) {
00348     GIVEN("La estructura OxigenoResponse con valores 0") {
00349         struct OxigenoResponse datosOX = {0,0};
00350
00351         REQUIRE( datosOX.A == 0 );
```

```
00352         REQUIRE( datosOX.B == 0 );
00353
00354         WHEN( "Valores de entrada mínimos 0x0000" ) {
00355             datosOX = decodeSensorOxigeno((char *)"0000");
00356
00357             THEN( "Resultados tienen que ser los mínimos" ) {
00358                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00359                 REQUIRE( datosOX.B == Approx(-100).epsilon(0.01));
00360             }
00361         }
00362         WHEN( "Valores de entrada máximos 0xFFFF" ) {
00363             datosOX = decodeSensorOxigeno((char *)"FFFF");
00364
00365             THEN( "Resultados tienen que ser los máximos" ) {
00366                 REQUIRE( datosOX.A == Approx(1.275).epsilon(0.01));
00367                 REQUIRE( datosOX.B == Approx(99.2).epsilon(0.01));
00368             }
00369         }
00370         WHEN( "Valores de entrada intermedio 0x7F7F" ) {
00371             datosOX = decodeSensorOxigeno((char *)"7F7F");
00372
00373             THEN( "Resultados tienen que ser intermedios" ) {
00374                 REQUIRE( datosOX.A == Approx(0.635).epsilon(0.01));
00375                 REQUIRE( datosOX.B == Approx(-0.78125).epsilon(0.01));
00376             }
00377         }
00378         WHEN( "Valores Voltaje Mínimo y Ajuste Combustible Máximo 0x00FF" ) {
00379             datosOX = decodeSensorOxigeno((char *)"00FF");
00380
00381             THEN( "Resultados tienen que ser Voltaje Mínimo y Ajuste Combustible Máximo" ) {
00382                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00383                 REQUIRE( datosOX.B == Approx(99.2).epsilon(0.01));
00384             }
00385         }
00386         WHEN( "Valores Ajuste Combustible Mínimo y Voltaje Máximo 0xFF00" ) {
00387             datosOX = decodeSensorOxigeno((char *)"FF00");
00388
00389             THEN( "Resultados tienen que ser Ajuste Combustible Mínimo y Voltaje Máximo" ) {
00390                 REQUIRE( datosOX.A == Approx(1.275).epsilon(0.01));
00391                 REQUIRE( datosOX.B == Approx(-100).epsilon(0.01));
00392             }
00393         }
00394     }
00395 }
00396
00401 SCENARIO( "Test de Relación Equivalente Combustible-Aire", "[decoders]" ) {
00402     GIVEN("La estructura OxigenoResponse con valores 0") {
00403         struct OxigenoResponse datosOX = {0,0};
00404
00405         REQUIRE( datosOX.A == 0 );
00406         REQUIRE( datosOX.B == 0 );
00407
00408         WHEN( "Valores de entrada mínimos 0x00000000" ) {
00409             datosOX = decodeRelacionCombAire((char *)"00000000");
00410
00411             THEN( "Resultados tienen que ser los mínimos" ) {
00412                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00413                 REQUIRE( datosOX.B == Approx(0).epsilon(0.01));
00414             }
00415         }
00416         WHEN( "Valores de entrada máximos 0xFFFFFFFF" ) {
00417             datosOX = decodeRelacionCombAire((char *)"FFFFFFFF");
00418
00419             THEN( "Resultados tienen que ser los máximos" ) {
00420                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00421                 REQUIRE( datosOX.B == Approx(8).epsilon(0.01));
00422             }
00423         }
00424         WHEN( "Valores de entrada intermedio 0x7FFF7FFF" ) {
00425             datosOX = decodeRelacionCombAire((char *)"7FFF7FFF");
00426
00427             THEN( "Resultados tienen que ser intermedios" ) {
00428                 REQUIRE( datosOX.A == Approx(1).epsilon(0.01));
00429                 REQUIRE( datosOX.B == Approx(4).epsilon(0.01));
00430             }
00431         }
00432         WHEN( "Valores Voltaje Mínimo y Relación Equivalente Máxima 0xFFFF0000" ) {
00433             datosOX = decodeRelacionCombAire((char *)"FFFF0000");
00434
00435             THEN( "Resultados tienen que ser Voltaje Mínimo y Relación Equivalente Máxima" ) {
00436                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00437                 REQUIRE( datosOX.B == Approx(0).epsilon(0.01));
00438             }
00439         }
00440         WHEN( "Valores Relación Equivalente Mínima y Voltaje Máximo 0x0000FFFF" ) {
00441             datosOX = decodeRelacionCombAire((char *)"0000FFFF");
00442
```

```
00443             THEN( "Resultados tienen que ser Relación Equivalente MínimA y Voltaje Máximo" ) {
00444                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00445                 REQUIRE( datosOX.B == Approx(8).epsilon(0.01));
00446             }
00447         }
00448     }
00449 }
00450
00455 SCENARIO( "Test de Relación Equivalente Combustible-Aire Actual", "[decoders]" ) {
00456     GIVEN("La estructura OxigenoResponse con valores 0") {
00457         struct OxigenoResponse datosOX = {0,0};
00458
00459         REQUIRE( datosOX.A == 0 );
00460         REQUIRE( datosOX.B == 0 );
00461
00462         WHEN( "Valores de entrada mínimos 0x00000000" ) {
00463             datosOX = decodeRelacionCombAireActual((char *)"00000000");
00464
00465             THEN( "Resultados tienen que ser los mínimos" ) {
00466                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00467                 REQUIRE( datosOX.B == Approx(-128).epsilon(0.01));
00468             }
00469         }
00470         WHEN( "Valores de entrada máximos 0xFFFFFFFF" ) {
00471             datosOX = decodeRelacionCombAireActual((char *)"FFFFFFFF");
00472
00473             THEN( "Resultados tienen que ser los máximos" ) {
00474                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00475                 REQUIRE( datosOX.B == Approx(128).epsilon(0.01));
00476             }
00477         }
00478         WHEN( "Valores de entrada intermedio 0x7FFF7FFF" ) {
00479             datosOX = decodeRelacionCombAireActual((char *)"7FFF7FFF");
00480
00481             THEN( "Resultados tienen que ser intermedios" ) {
00482                 REQUIRE( datosOX.A == Approx(1).epsilon(0.01));
00483                 REQUIRE( datosOX.B == Approx(-0.00390625).epsilon(0.01));
00484             }
00485         }
00486         WHEN( "Valores Corriente Mínimo y Relación Equivalente Máxima 0xFFFF0000" ) {
00487             datosOX = decodeRelacionCombAireActual((char *)"FFFF0000");
00488
00489             THEN( "Resultados tienen que ser Corriente Mínimo y Relación Equivalente Máxima" ) {
00490                 REQUIRE( datosOX.A == Approx(2).epsilon(0.01));
00491                 REQUIRE( datosOX.B == Approx(-128).epsilon(0.01));
00492             }
00493         }
00494         WHEN( "Valores Relación Equivalente Mínima y Corriente Máximo 0x0000FFFF" ) {
00495             datosOX = decodeRelacionCombAireActual((char *)"0000FFFF");
00496
00497             THEN( "Resultados tienen que ser Relación Equivalente MínimA y Corriente Máximo" ) {
00498                 REQUIRE( datosOX.A == Approx(0).epsilon(0.01));
00499                 REQUIRE( datosOX.B == Approx(128).epsilon(0.01));
00500             }
00501         }
00502     }
00503 }
00504
00509 SCENARIO( "Test de Valores máximos relación de combustible-aire, voltaje, corriente y presión
      absoluta", "[decoders]" ) {
00510     GIVEN("La estructura RelacionesResponse con valores 0") {
00511         struct RelacionesResponse datosREL = {0,0,0,0};
00512
00513         REQUIRE( datosREL.A == 0 );
00514         REQUIRE( datosREL.B == 0 );
00515         REQUIRE( datosREL.C == 0 );
00516         REQUIRE( datosREL.D == 0 );
00517
00518         WHEN( "Valores de entrada mínimos 0x00000000" ) {
00519             datosREL = decodeRelaciones((char *)"00000000");
00520
00521             THEN( "Resultados tienen que ser los mínimos" ) {
00522                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00523                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00524                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00525                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00526             }
00527         }
00528         WHEN( "Valores de entrada máximos 0xFFFFFFFF" ) {
00529             datosREL = decodeRelaciones((char *)"FFFFFFFF");
00530
00531             THEN( "Resultados tienen que ser los máximos" ) {
00532                 REQUIRE( datosREL.A == Approx(255).epsilon(0.01));
00533                 REQUIRE( datosREL.B == Approx(255).epsilon(0.01));
00534                 REQUIRE( datosREL.C == Approx(255).epsilon(0.01));
00535                 REQUIRE( datosREL.D == Approx(2550).epsilon(0.01));
00536             }
```

```
00537         }
00538         WHEN( "Valores de entrada intermedio 0x7F7F7F7F" ) {
00539             datosREL = decodeRelaciones((char *)"7F7F7F7F");
00540 
00541             THEN( "Resultados tienen que ser intermedios" ) {
00542                 REQUIRE( datosREL.A == Approx(127).epsilon(0.01));
00543                 REQUIRE( datosREL.B == Approx(127).epsilon(0.01));
00544                 REQUIRE( datosREL.C == Approx(127).epsilon(0.01));
00545                 REQUIRE( datosREL.D == Approx(1270).epsilon(0.01));
00546             }
00547         }
00548         WHEN( "Valores de entrada Relación Máxima, resto mínimo 0xFF000000" ) {
00549             datosREL = decodeRelaciones((char *)"FF000000");
00550 
00551             THEN( "Resultados tienen que ser Relación Máxima, resto mínimo" ) {
00552                 REQUIRE( datosREL.A == Approx(255).epsilon(0.01));
00553                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00554                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00555                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00556             }
00557         }
00558         WHEN( "Valores Voltaje Máximo, resto mínimo 0x00FF0000" ) {
00559             datosREL = decodeRelaciones((char *)"00FF0000");
00560 
00561             THEN( "Resultados tienen que ser Voltaje Máximo, resto mínimo" ) {
00562                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00563                 REQUIRE( datosREL.B == Approx(255).epsilon(0.01));
00564                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00565                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00566             }
00567         }
00568         WHEN( "Valores Corriente Máxima, resto mínimo 0x0000FF00" ) {
00569             datosREL = decodeRelaciones((char *)"0000FF00");
00570 
00571             THEN( "Resultados tienen que ser Corriente Máxima, resto mínimo" ) {
00572                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00573                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00574                 REQUIRE( datosREL.C == Approx(255).epsilon(0.01));
00575                 REQUIRE( datosREL.D == Approx(0).epsilon(0.01));
00576             }
00577         }
00578         WHEN( "Valores Presión Máxima, resto mínimo 0x000000FF" ) {
00579             datosREL = decodeRelaciones((char *)"000000FF");
00580 
00581             THEN( "Resultados tienen que ser Presión Máxima, resto mínimo" ) {
00582                 REQUIRE( datosREL.A == Approx(0).epsilon(0.01));
00583                 REQUIRE( datosREL.B == Approx(0).epsilon(0.01));
00584                 REQUIRE( datosREL.C == Approx(0).epsilon(0.01));
00585                 REQUIRE( datosREL.D == Approx(2550).epsilon(0.01));
00586             }
00587         }
00588     }
00589 }
00590 
00595 SCENARIO( "Test de decodificación de Data Trouble Codes (DTC)", "[decoders]" ) {
00596     GIVEN("Vector vacío cuyos componentes son strings (DTC's)") {
00597         std::vector<std::string> vec_dtcs;
00598 
00599         REQUIRE( vec_dtcs.empty() == 1 );
00600 
00601         WHEN( "Sólo un DTC" ) {
00602             vec_dtcs = decodeDTCs((char *)"013300000000");
00603 
00604             THEN( "Resultado de sólo un DTC" ) {
00605                 REQUIRE( vec_dtcs.size() == 1);
00606                 REQUIRE( vec_dtcs[0] == "P0133");
00607             }
00608         }
00609         WHEN( "Dos DTC's" ) {
00610             vec_dtcs = decodeDTCs((char *)"0133D0160000");
00611 
00612             THEN( "Resultado de dos DTC's" ) {
00613                 REQUIRE( vec_dtcs.size() == 2);
00614                 REQUIRE( vec_dtcs[0] == "P0133");
00615                 REQUIRE( vec_dtcs[1] == "U1016");
00616             }
00617         }
00618         WHEN( "Tres DTC's" ) {
00619             vec_dtcs = decodeDTCs((char *)"0133D0161131");
00620 
00621             THEN( "Resultado de tres DTC's" ) {
00622                 REQUIRE( vec_dtcs.size() == 3);
00623                 REQUIRE( vec_dtcs[0] == "P0133");
00624                 REQUIRE( vec_dtcs[1] == "U1016");
00625                 REQUIRE( vec_dtcs[2] == "P1131");
00626             }
00627         }
```

```
00628     }
00629 }
00630
00635 SCENARIO( "Test de decodificación PIDs disponibles", "[decoders]" ) {
00636     GIVEN("Vector vacío cuyos componentes son int (DTC's)") {
00637         std::vector<int> vec_pids;
00638
00639         REQUIRE( vec_pids.empty() == 1 );
00640
00641         WHEN( "PIDs impares disponibles" ) {
00642             vec_pids = decodePIDS((char *)"AAAAAAAA");
00643
00644             THEN( "Resultado con números impares del 1-20 (Hexadecimal, decimal 1-32)" ) {
00645                 REQUIRE( vec_pids.size() == 16);
00646                 int pids_impares = 1;
00647                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00648                     REQUIRE( vec_pids[i] == pids_impares);
00649                     pids_impares+=2;
00650
00651                 }
00652             }
00653         }
00654         WHEN( "PIDs pares disponibles" ) {
00655             vec_pids = decodePIDS((char *)"55555555");
00656
00657             THEN( "Resultado con números pares del 1-20 (Hexadecimal, decimal 1-32)" ) {
00658                 REQUIRE( vec_pids.size() == 16);
00659                 int pids_pares = 2;
00660                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00661                     REQUIRE( vec_pids[i] == pids_pares);
00662                     pids_pares+=2;
00663
00664                 }
00665             }
00666         }
00667         WHEN( "PIDs primeros disponibles" ) {
00668             vec_pids = decodePIDS((char *)"FFFF0000");
00669
00670             THEN( "Resultado con los primeros números del 1-20 (Hexadecimal, decimal 1-32)" ) {
00671                 REQUIRE( vec_pids.size() == 16);
00672                 int pids_primeros = 1;
00673                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00674                     REQUIRE( vec_pids[i] == pids_primeros);
00675                     pids_primeros++;
00676
00677                 }
00678             }
00679         }
00680         WHEN( "PIDs ultimos disponibles" ) {
00681             vec_pids = decodePIDS((char *)"0000FFFF");
00682
00683             THEN( "Resultado con los ultimos números del 1-20 (Hexadecimal, decimal 1-32)" ) {
00684                 REQUIRE( vec_pids.size() == 16);
00685                 int pids_ultimos = 17;
00686                 for (uint32_t i = 0; i < vec_pids.size(); ++i){
00687                     REQUIRE( vec_pids[i] == pids_ultimos);
00688                     pids_ultimos++;
00689
00690                 }
00691             }
00692         }
00693     }
00694 }
00695
00700 SCENARIO( "Test de decodificación del estado del coche", "[decoders]" ) {
00701     GIVEN("Map de <string, string> de estado vacío") {
00702         std::map<std::string, std::string> status;
00703
00704         REQUIRE( status.empty() == 1 );
00705
00706         WHEN( "La entrada es 0x81076504" ) {
00707             status = decodeStatus((char *)"81076504");
00708
00709             THEN( "Resultado 1 DTC, Compresión, MIL Encendida y todas pruebas correctas" ) {
00710                 REQUIRE( status.size() == 9);
00711                 REQUIRE( status["MIL"] == "Encendida");
00712                 REQUIRE( status["DTC_CNT"] == "1");
00713                 REQUIRE( status["IGNICION"] == "Compresión");
00714                 REQUIRE( status["Sistema de catalizador NMHC"] == "Prueba Correcta");
00715                 REQUIRE( status["Sistema de detección de condiciones inadecuadas de ignición en cilindros"]
      == "Prueba Correcta");
00716                 REQUIRE( status["Sistema de componentes integrales"] == "Prueba Correcta");
00717                 REQUIRE( status["Sistema de combustible"] == "Prueba Correcta");
00718                 REQUIRE( status["Sistema del sensor de gases de escape"] == "Prueba Correcta");
00719                 REQUIRE( status["Sistema de monitor del filtro de partículas (Particular Matter, PM)"] == "
     Prueba Correcta");
00720             }
```

```
00721         }
00722     }
00723 }
```

Índice alfabético