Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de un simulador electrónico de una ECU y su diagnóstico sobre CAN y OBD-II

Autor: José Beltrán Zambrano

Tutor: Federico Barrero García

Cotutor: Jesús Sánchez García

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

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

Desarrollo de un simulador electrónico de una ECU y su diagnóstico sobre CAN y OBD-II

Autor:
José Beltrán Zambrano

Tutor:
Federico Barrero García
Profesor titular

Cotutor:
Jesús Sánchez García

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

Trabajo Fin de Grado: Desarrollo de un simulador electrónico de una ECU y su diagnóstico sobre CAN y OBD-II

Autor: José Beltrán Zambrano

Tutor: Federico Barrero García

Cotutor Jesús Sánchez García

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2015

El Secretario del Tribunal

Sobre el autor

José Beltrán Zambrano

josebelzam@gmail.com

Grado en Ingeniería de las Tecnologías de Telecomunicación

Universidad de Sevilla

Agradecimientos

Realizar el proyecto y este documento, no ha sido para nada una tarea fácil, es por eso que quisiera agradecer su ayuda a las siguientes personas.

Para empezar agradecer a Federico Barrero la oportunidad que me ha dado ofreciéndome este proyecto, por la paciencia que ha tenido con las múltiples adversidades que han ido surgiendo durante la realización del mismo y por atenderme con amabilidad y cercanía cada vez que me ha sido necesario.

También mis más profundo agradecimiento a Jesús Sánchez, por su constante e incansable ayuda, por sacar tiempo para resolverme todas mis dudas e inquietudes y por darme la confianza que me faltaba para terminar este proyecto.

No me puedo olvidar de mis compañeros de carrera, agradecerles a ellos esos buenos momentos que hemos pasado juntos. A todos y cada uno de ellos les deseo lo mejor y que no perdamos el contacto después de esta etapa tan bonita que hemos vivido juntos.

Por último, quiero agradecer a toda mi familia el constante apoyo durante todos estos años y el estar siempre a mi lado tanto en los buenos como en los malos momentos. A mis padres, a mi hermana, a mi novia y a mi familia política, gracias.

José Beltrán Zambrano

Sevilla, 2015

Resumen

Los sistemas de diagnóstico de a bordo nos permiten conocer en tiempo real el estado de un vehículo. No solo nos permiten conocer los códigos de fallos almacenados, también nos permiten conocer en tiempo real un gran número de variables de especial relevancia como la velocidad, el nivel de combustible, el nivel de emisión de CO2, etc.

Este Trabajo Fin de Grado se centra en un sistema de diagnóstico específico, de los muchos existentes, basado en el protocolo OBD-II sobre CAN-BUS. En primer lugar se realiza un estudio teórico de los distintos protocolos de comunicación bus y de diagnosis usados actualmente en los vehículos. A continuación, se propone una implementación práctica de un sistema de diagnosis OBD-II centrándose en el extremo del BUS que corresponde al *Engine Control Unit* o Unidad de Control de Motor (ECU).

El sistema propuesto consiste en desarrollar un equipo simulador de una ECU. Estos equipos son de gran utilidad cuando se necesitan incluir nuevos parámetros a medir en las ECUs y ver cómo se detectan luego en los programas software de visualización de diagnóstico o programas escáner. Este tipo de equipos pueden ser de gran ayuda para los nuevos vehículos eléctricos, en los cuales la ECU necesitará trabajar cada vez más con distintos sensores y parámetros de los que se está acostumbrado a ver en los vehículos de combustión.

En el sistema desarrollado se ha implementado el estándar OBD-II en una placa Arduino UNO conectada un módulo transceiver CAN del fabricante Seeed Studio: CAN-BUS Shield. La ECU se ha conectado a un PC a través de un puerto USB haciendo uso de una interfaz OBD-II, y para visualizar los datos de forma gráfica se ha hecho uso de un software de diagnosis gratuito como es ScanMaster-ELM.

Abstract

The on-board diagnostic systems allow us to know in real time the vehicle status, not only lets us know the diagnostic trouble codes, also allows us to know in real time a large number of variables particularly important as speed, fuel level, CO2 emissions level, etc.

This BSc thesis studies focuses on a specific diagnosis system based on OBD-II protocol onto CAN-BUS. In the first place a theoretical research is done about various communication buses and diagnostic protocols in vehicles. Next, a practical implementation of an OBD-II diagnosis system, focusing on the part of the bus corresponding to the Engine Control Unit (ECU), is proposed.

The proposed system consists of developing a simulation equipment of an ECU. These equipments are very useful when you need to include new parameters to measure in an ECU and then see how diagnostic visualization programs or scanner programs detect these new parameters. This type of equipment may be helpful for new electric vehicles, in which the ECU needs to work with different sensors and parameters from those that we are used to in combustion vehicles.

The developed system has been implemented using OBD-II over Arduino UNO, connected to a CAN transceiver module manufactured by Seeed Studio: CAN-BUS Shield. The ECU simulator is connected to a PC through a USB port using an OBD-II interface and to display the data graphically, we havemade use of a free diagnosis software, the ScanMaster-ELM.

Índice

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Notación

CAN	Controller Area Network
CARB	California Air Resources Board
CSMA/CR	Carrier Sense Multiple Access / Collision Resolution
DTC	Diagnostic Trouble Codes
DLC	Data Link Connector
ECM	Engine Control Module
ECU	Engine Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
EGR	Exhaust Gas Recirculation
EPA	Environmental Protection Agency
IDE	Integrated Development Environment
ISO	International Organization for Standardization
ISO TP	ISO Transport Protocol
ITS	Intelligent Transport System
KWP2000	Keyword Protocol 2000
MAC	Medium Access Chanel
MIL	Malfunction Indicator Lamp
LSDU	Link Service Data Unit
LLC	Logical Link Control
OBD	On Board Diagnostics
OBD-I	On Board Diagnostics First Generation
OBD-II	On Board Diagnostics Second Generation
OSI	Open System Interconnection
PC	Personal Computer
PID	Parameter Identification
PWM	Pulse Width Modulation

RPM	Revoluciones por minute
RS-232	Recommended Standard 232
SAE	Society of Automotive Engineers
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
USB	Universal Serial Bus
VPW	Variable Pulse Width
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
WLAN	Wireless Local Area Network

1 Introducción y objetivos

En la actualidad y debido al constante avance tecnológico, los automóviles que anteriormente eran puramente mecánica y electricidad, hoy día son controlados por la electrónica y sistemas de procesamiento los cuales se encargan de controlar el automóvil informando al usuario sobre el estado del mismo, almacenando códigos de falla y transmitiendo información en tiempo real de variables que necesitan ser visualizadas. Estos avances han permitido simplificar la reparación del automóvil, detectando fallos potenciales que pudieran dañar más al automóvil, además de simplificar y centralizar la visualización de variables en tiempo real, que ahora pueden ser leídas directamente desde el ordenador central o ECU. El presente proyecto implementa un sistema de diagnóstico a bordo basado en el protocolo de diagnosis OBD-II sobre CAN-BUS que permitirá, tanto visualizar variables en tiempo real, como realizar un diagnóstico del estado del automóvil que muestre los códigos de falla almacenados y permita borrarlos una vez reparados. El sistema propuesto se centra en el desarrollo de un equipo electrónico simulador de una ECU. Estos equipos son de gran utilidad cuando se necesita incluir nuevos parámetros a medir en la ECU y ver cómo se detectan luego en los programas software de visualización de diagnóstico.

1.1 Introducción

El siguiente Trabajo Fin de Grado forma parte del proyecto PROMUEVE, un proyecto que se encuentra encuadrado dentro de un conjunto de proyectos de I+D+i financiados por la Junta de Andalucía y el Gobierno de España. El trabajo ha sido tutelado por el profesor Federico Barrero García, investigador principal del proyecto PROMUEVE, y co-tutelado por Jesús Sánchez García.

El proyecto PROMUEVE trata de desarrollar un prototipo de vehículo eléctrico, modelo CROSS RIDER, diseñando un nuevo sistema de propulsión, y creando en él un sistema de comunicaciones que comprenda una red interna (o "*In-Vehicle Network*"). Dicha red permitirá la conectividad con el exterior mediante comunicaciones *Vehicle-to-Vehicle* (V2V) y *Vehicle-to-Infrastructure* (V2I) para que el vehículo pueda ser integrado en un sistema ITS (*Intelligent Transport System*). También se dotará al vehículo de conectividad con dispositivos que permitan la visualización del estado del mismo tales como Smart-phones, Tablets, etc. Según lo dicho, se aprecia que este proyecto está dividido en dos grandes líneas de trabajo:

- Desarrollo de un nuevo sistema de propulsión, basado en accionamiento multifásico.
- Desarrollo de un sistema de comunicaciones para el vehículo eléctrico.

Este Trabajo Fin de Grado se encuadra dentro de la segunda línea de trabajo, es decir, en el desarrollo de un sistema de comunicaciones. Dicho desarrollo se ha dividido en tres trabajos fin de grado:

- Diseño y montaje de una red vehicular cableada basada en el bus CAN y OBD-II. En esta línea de trabajo se encuentra este Trabajo Fin de Grado con los objetivos señalados en la siguiente sección.
- Diseño y montaje de una red inalámbrica que permita comunicar el ordenador de a bordo con dispositivos externos compatibles con los estándares WAVE/CALM. Para ello se emplearán tarjetas ALIX-BOARD.
- Desarrollo de una interfaz HMI que permita visualizar en Smart-phones o Tablets el funcionamiento del vehículo eléctrico.

1.2 Objetivos

Los ordenadores de a bordo actuales están diseñados para controlar las funciones mecánicas y eléctricas del automóvil, llevando el control de sensores que informen del estado en tiempo

real del vehículo, por tanto es necesario un dispositivo que muestre toda esta información al usuario permitiéndole conocer de forma rápida y sencilla el estado de su vehículo.

Para una correcta comunicación entre el ordenador de a bordo y el sistema de diagnóstico es necesario iniciar el dialogo mediante los protocolos de comunicación definidos para ello. En este sentido resulta necesario realizar un estudio sobre los protocolos de comunicación que usan los distintos fabricantes de automóviles.

Por otro lado, un sistema de diagnóstico se compone principalmente de dos extremos, uno que corresponde al software de diagnóstico o escáner, empleado para leer y representar visualmente la información de diagnóstico (parámetros, fallos, etc.); y otro compuesto por las ECUs instaladas dentro del vehículo que, tras recopilar la información de los sensores, envían esta información hacia el software de visualización. Ambos extremos son importantes, aunque en la actualidad hay mucha más información disponible acerca de programas software de diagnóstico que de ECUs.

También cabe señalar, que las ECUs están ya evolucionando para adaptarse a las nuevas características de los vehículos eléctricos. En estos vehículos van a necesitar trabajar con otros tipos de sensores distintos a los que comúnmente se encuentran en los vehículos de combustión. En estos casos es muy útil tener un dispositivo que haga las veces de simulador de ECU, al que se le puedan incorporar nuevos sensores y parámetros a medir propios de vehículos eléctricos y poder ver la nueva información recopilada en los programas escáner disponibles.

Se marca pues como objetivo general de este Trabajo Fin de Grado el diseño de un equipo electrónico de simulación de una ECU, que sea capaz de comunicarse con programas software de visualización de la información de diagnóstico, de enviar variables en tiempo real, de responder cuando se le soliciten los códigos de fallos del vehículo, así como de borrar dichos códigos de falla supuesta reparación. Con el fin de cumplir el objetivo final se ha dividido el trabajo en las siguientes tareas:

- Estudio detallado de los protocolos de comunicación usados para realizar diagnósticos de a bordo, haciendo especial hincapié en el protocolo OBD-II sobre CAN-BUS.
    - Estado actual de las comunicaciones vehiculares.
    - Estudio de las distintas versiones del protocolo OBD-II, así como de las distintas capas físicas sobre las que puede estar implementado, entre ellas CAN-BUS.
    - Estudio de los distintos escáneres y soluciones para el diagnóstico OBD-II.

- Desarrollo e implementación de un equipo de simulación de una ECU que sea capaz de comunicarse con un escáner OBD-II, simulando el comportamiento en tiempo real de un vehículo en marcha.

  - Seleccionar el hardware adecuado para la implementación del simulador de dicha ECU.
  - Seleccionar el escáner OBD-II y la solución de comunicación propuesta.
  - Desarrollo de una aplicación sobre el hardware seleccionado capaz de enviar tramas OBD-II sobre CAN-BUS comunicándose así con el escáner propuesto.

- Simulación del sistema completo, lectura de variables en tiempo real, detección de códigos de falla y borrado de los mismos.
  - Comunicación bidireccional entre la ECU desarrollada y el escáner OBD-II.
  - Visualización de variables en tiempo real: velocidad, RPM, etc.
  - Estado del vehículo en tiempo real haciendo uso del indicador luminoso MIL.
  - Detección y borrado de códigos de falla.

2 Fundamentos Teóricos

En el siguiente capítulo se realiza un estudio de los protocolos de comunicación empleados para el diagnóstico de abordo en vehículos, haciendo especial hincapié en el estándar empleado en este Trabajo Fin de Grado, es decir en OBD-II sobre CAN-BUS. OBD-II se caracteriza por ser un sistema estandarizado, que permite, de manera fácil, ver qué errores se han producido en un vehículo cualquiera utilizando una única codificación y un conector estandarizado. La normativa más importante en esta materia es la establecida por la Organización Internacional para la Estandarización (ISO): OBD ISO 15031. En esta norma se recogen todos los aspectos a tener en cuenta a la hora de realizar una comunicación empleando OBD-II. A lo largo del siguiente capítulo ahondaremos más en esta normativa con el fin de presentar los conocimientos sobre el funcionamiento de dicho protocolo, en sus distintas versiones, que se han empleado en el desarrollo de este proyecto.

2.1 Evolución de la electrónica del automóvil

A finales del siglo XIX se introdujo en Europa el automóvil como medio de transporte, estos primeros vehículos llevaban un motor de combustión interna de cuatro tiempos bastante pesado y rudimentario. Más adelante, Gottlieb Daimler ideó una variante mucho más ligera que sería el precursor de todos los motores de combustión posteriores. Con los años, los automóviles fueron incorporando innovaciones que aumentaron su rendimiento y mejoraron sus prestaciones, estas mejoras incluían el uso del diferencial, correas, baterías, etc. Pero en su diseño, el motor de combustión interna no experimentó grandes cambios.

A inicios del siglo XX las innovaciones mecánicas siguieron sin afectar al diseño básico de los motores, suponiendo tan solo la adición de elementos orientados a la optimización de los mismos. Es a finales de los 70 cuando se empieza a incorporar la electrónica a los automóviles. En esta línea, se añadieron los primeros sensores a los motores para verificar su correcto funcionamiento, junto a estos se incorporaron unidades de control de motor encargadas de manejar dichos sensores. El objetivo inicial de estos elementos electrónicos era el control de las emisiones de gases contaminantes y facilitar el diagnóstico de averías.

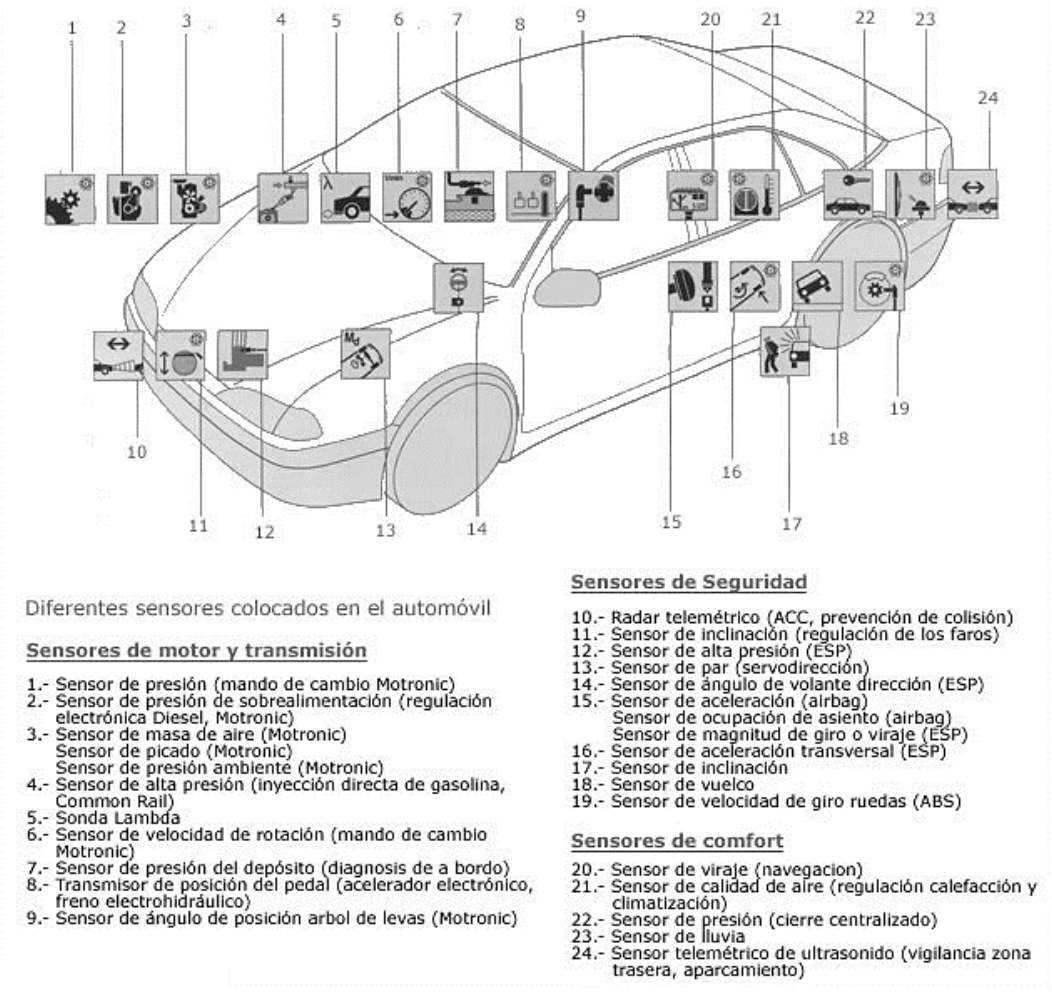


Figura 1: Sensores del automóvil a gran escala.

A partir de la década de los 80 la mayor parte de las innovaciones provienen principalmente de la incorporación de la electrónica, y no de la incorporación de mejoras mecánicas. Se añadieron un gran número de sensores al automóvil y se mejoraron las unidades de control de motor, tal fue el avance que a día de hoy podemos encontrar más de 200 sensores en un automóvil de gama media. Las primeras unidades de control eran conocidas como Módulos de Control de Motor o ECM, con el tiempo se hicieron más complejas y pasaron a convertirse en Unidades de Control Electrónico o ECU. En España las ECUs son conocidas coloquialmente como centralitas.

En la actualidad los sensores se encargan de medir temperaturas, presiones, rotaciones, volúmenes y una gran cantidad de parámetros relacionados con el funcionamiento del motor. La información captada por estos sensores es enviada y almacenada en las centralitas o ECUs, una vez allí esta información es comparada con los valores óptimos que están almacenados en las memorias. Cuando se encuentra un valor incorrecto, la centralita notifica un fallo avisando al conductor de alguna forma (indicadores luminosos, sonidos, mensajes de alerta, etc.), los fallos quedan almacenados para su posterior verificación por el personal cualificado.

Tal es la importancia de estos sistemas de captación de errores, que a día de hoy existe una ley en vigor que obliga a los nuevos automóviles a disponer de una interfaz por medio de la cual se pueda obtener información o un diagnóstico del automóvil mediante un equipo de prueba. Estos sistemas de diagnóstico se han adquirido especial importancia a la hora de garantizar la seguridad del conductor, además de facilitar el chequeo y las reparaciones a los profesionales del sector.

La gran mayoría de fabricantes incorporan en sus vehículos sistemas de autodiagnóstico o sistemas de diagnóstico a bordo

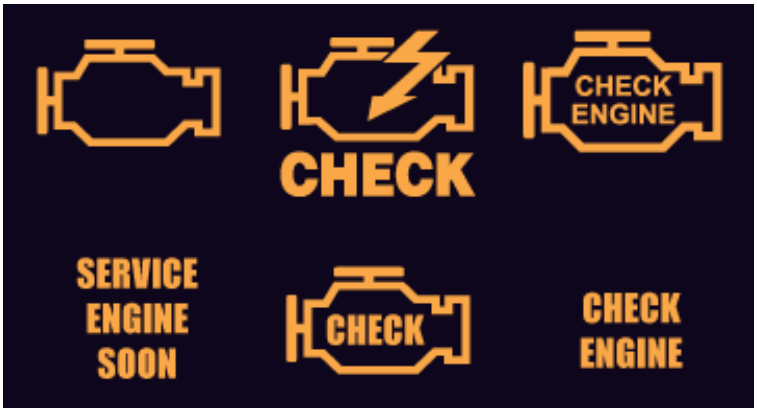


Figura 2: Distintos iconos del indicador luminoso.

2.2 Introducción a los sistemas OBD

Con el fin de combatir los problemas de contaminación en la ciudad de Los Ángeles (California, EEUU), el Estado de California exigió sistemas de control de emisiones de gases en los modelos de automóvil posteriores a 1966. El Gobierno Federal de los Estados Unidos extendió estos controles a toda la nación en 1968. El Congreso aprobó el *Clean Air Act* (Acta Anticontaminación) en 1970, ese mismo año se fundó la *Environmental Protection Agency* (Agencia de Protección Medioambiental) o EPA, esta agencia inició el desarrollo de una serie de estándares o normas con el fin de regular las emisiones de gases contaminantes, además de unos requerimientos para el mantenimiento de los vehículos con el fin de ampliar su vida útil.

Tras esta oleada de normas, los fabricantes implementaron sistemas de encendido y de alimentación controlada de combustible, con sensores que medían las prestaciones del motor y ajustaban los sistemas para conseguir minimizar la contaminación. Además estos sensores permitían una cierta ayuda en la reparación del automóvil y su mantenimiento preventivo.

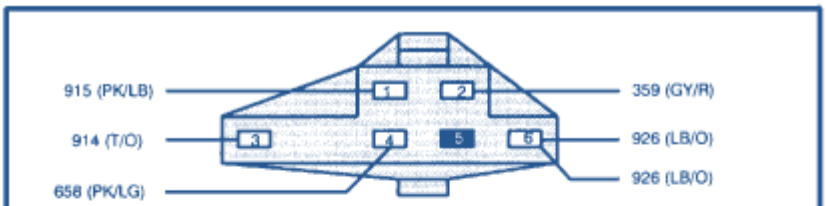
En abril de 1985 un organismo estatal de California, el CARB (*California Air Resources Board*), aprobó una regulación para un sistema de diagnóstico a bordo u OBD (*On-Board Diagnostic*). Esta regulación que se aplica a los automóviles vendidos en el estado de California a partir de 1988, específica que el Módulo de Control de Motor o ECM (*Engine Control Module*) debe monitorizar ciertos componentes del vehículo relacionados con las emisiones de gases para asegurar un correcto funcionamiento, además de encender una lámpara indicadora de fallo o MIL (*Malfunction Indicator Lamp*) en el cuadro de mandos cuando se detecta un problema. El nuevo sistema OBD también aporta un sistema de Códigos de Error de Diagnóstico o DTC (*Diagnostic Trouble Codes*) y unas tablas de errores en los manuales de reparación para ayudar a los técnicos a determinar las causas más probables de avería en el motor y problemas en las emisiones. Los objetivos básicos de esta nueva regulación eran fundamentalmente dos:

- Reforzar el cumplimiento de las normativas de la regulación de emisión de gases, alertando al conductor cuando se produzca algún fallo.
- Ayudar a los técnicos en la identificación y reparación de fallos en el sistema de control de emisiones.

Por lo tanto se implantó OBD en los sistemas que se consideraban causa principal del incremento en las emisiones de gases de escape en caso de fallo. Principalmente se implantó en los sensores principales del motor, en el sistema de medición de combustible y en el sistema de Recirculación de Gases de Escape o EGR (*Exhaust Gas Recirculation*).

Más tarde surgiría OBD-I (*On Board Diagnostics First Generation*), esta primera regulación OBD obligaba a los fabricantes a instalar un sistema de monitorización de algunos de los componentes controladores de emisiones en automóviles. OBD-I se volvió obligatorio en todos los vehículos a partir de 1991, aunque con el tiempo se demostró que no era un sistema

efectivo ya que solamente monitoreaban algunos componentes relacionados con las emisiones, sin ser calibrados para un nivel específico de emisiones.



C198 (BLACK)
DATA LINK CONNECTOR (DLC)

PIN	CIRCUIT	CIRCUIT FUNCTION
1	915 (PK/LB)	Data (−)
2	359 (GY/R)	Sensor Signal Return
3	914 (T/O)	Data (+)
4	658 (PK/LG)	Malfunction Indicator lamp Output
5	—	NOT USED
6	926 (LB/O)	Fuel Pump Enable Output

Figura 3: Conector OBD-I de la marca FORD.

Inicialmente hubo varios estándares y cada fabricante tenía sus propios sistemas de códigos. En 1988 la Sociedad de Ingenieros de Automoción o SAE (*Society of Automotive Engineers*) definió un conector estándar y un conjunto de códigos de diagnóstico, con el fin de terminar con estas diferencias entre fabricantes que dificultaban mucho la tarea de los técnicos.

Más tarde, nuevas medidas más estrictas en los límites de emisiones llevaron a la creación, por parte de la SAE, de un nuevo estándar OBD-II (*On Board Diagnostics Second Generation*), este estándar fue adoptado por la EPA y el CARB aprobándose para su implementación en 1996. El nuevo estándar OBD-II no solo aportaría un control casi completo del motor, también monitorizaría partes del chasis y otros dispositivos del vehículo convirtiéndose así en el centro de control de diagnóstico del vehículo. Junto con este nuevo estándar se evolucionó de los antiguos Módulos de Control de Motor o ECMs a las actuales Unidades de Control Electrónico o ECU, verdaderas cajas negras de un vehículo.

Desde ese momento, OBD-II sería un requisito legal para todos los nuevos automóviles en Estados Unidos. Con base a esta regla americana se impuso en los noventa la inclusión de sistemas de diagnóstico también para los automóviles destinados al mercado europeo. En Europa, según la Directiva 98/69EG, los automóviles de gasolina desde el año 2000 en adelante, los diésel desde 2003 en adelante, y los camiones desde 2005 en adelante tienen que estar provistos de un sistema de diagnóstico OBD-II.

En la actualidad, la siguiente evolución planeada para sistemas de diagnóstico es el OBD-III, en el que los propios automóviles se comunican con las autoridades si se produce un

empeoramiento de las emisiones de gases nocivos mientras se está en marcha. Si esto sucede, se pedirá a través de una tarjeta indicativa, que se corrijan los defectos. Con OBD-III se pretende que los vehículos puedan ser monitorizados en cualquier momento y lugar, permitiendo vigilar cuidadosamente el cumplimiento de la política de emisiones contaminantes.

2.3 Sistemas OBD-II

La presente sección está dedicada a estudiar más a fondo el funcionamiento de los sistemas OBD-II, con el objetivo de aclarar las siguientes cuestiones: ¿para qué sirven los sistemas OBD-II? ¿cómo funciona OBD-II? ¿qué protocolos usa OBD-II? y ¿cómo obtener la información mediante OBD-II?

2.3.1 Funciones de los sistemas OBD-II

Todos los vehículos actuales, disponen de una o varias ECUs[1] (*Electronic Control Units*), que se encargan de gestionar los distintos sistemas eléctricos, electrónicos y mecánicos que contienen. En concreto, la ECU gestiona ciertos parámetros del motor del vehículo para asegurar su correcto funcionamiento. Las relaciones entre estos parámetros deben mantenerse acotadas, dependiendo de las condiciones externas estos parámetros tendrán un rango de variación determinado, en caso contrario es que se está produciendo algún mal funcionamiento en el vehículo.

Figura 4: ECU del Citroën C4.

[1] ECU puede hacer referencia tanto a Engine Control Unit como a Electronic Control Unit. En este documento se utilizará principalmente para hablar de Engine Control Units.

Los parámetros principales que dictan cómo debe estar funcionando el motor, y que verifican si todo está funcionando correctamente son:

- Velocidad.
- Carga.
- Temperatura del motor.
- Consumo de combustible.
- Temperatura ambiente.
- Caudal de aire.
- Emisiones.

Para conocer dichos parámetros, los automóviles actuales, incorporan una gran cantidad de sensores, que permiten a la ECU conocer cuáles son las condiciones externas, y decidir cómo actuar sobre el motor. En caso de que alguno de los parámetros se salga de los rangos marcados, el sistema OBD-II es el encargado de almacenar esta información y de avisar al conductor de que existe un mal funcionamiento en el motor, señalizando con un indicador luminoso que es recomendable ir al taller a revisar qué error se ha producido.

Una vez el vehículo llega al taller, el equipo de mecánicos puede acceder a la información almacenada mediante OBD-II, ver que error era el que se ha producido, y arreglarlo en caso de necesidad sin tener que hacer múltiples pruebas para descubrir la procedencia del error.

Algunas de las funciones específicas de control que puede desempeñar OBD-II según el tipo de motor son las siguientes:

- Control en los motores de gasolina:
    - Vigilancia del rendimiento del catalizador.
    - Diagnóstico de envejecimiento de sondas lambda.
    - Prueba de tensión de sondas lambda.
    - Sistema de aire secundario (si el vehículo lo incorpora).
    - Sistema de recuperación de vapores de combustible.
    - Prueba de diagnóstico de fugas.
    - Sistema de alimentación de combustible.
    - Fallos de la combustión.
    - Control del sistema de gestión electrónica.
    - Sensores y actuadores del sistema electrónico que intervienen en la gestión del motor o están relacionados con las emisiones de escape.

- Control en los motores diésel
    - Fallos de la combustión.

- Regulación del comienzo de la inyección.
- Regulación de la presión de sobrealimentación.
- Recirculación de gases de escape.
- Funcionamiento del sistema de comunicación entre unidades de mando, por ejemplo el bus CAN.
- Control del sistema de gestión electrónica.
- Sensores y actuadores del sistema electrónico que intervienen en la gestión del motor o están relacionados con las emisiones de escape.
- Control de la contaminación.

Uno de los aspectos más importantes que permite controlar OBD-II es la contaminación que produce el vehículo. El estado actual de la técnica no permite, o sería muy caro, realizar la medida directa de los gases CO (monóxido de carbono), HC (hidrocarburos) y NOx (óxidos nítricos), por lo que este control lo realiza la ECU de manera indirecta, detectando los niveles de contaminación a partir del análisis del funcionamiento de los componentes adecuados y del correcto desarrollo de las diversas funciones del equipo de inyección que intervengan en la combustión.

2.3.2 **Modos de medición OBD-II**

Para realizar todas sus funciones OBD-II puede trabajar en nueve modos de medición. Estos modos son comunes a todos los vehículos y permiten desde registrar datos para su verificación, extraer códigos de averías, borrarlos y realizar pruebas dinámicas de actuadores. Los modos en que se presentan la información se hallan estandarizados, al igual que el tamaño de la trama OBD-II. A continuación se estudia cada uno de ellos de forma detallada.

Byte #0	Byte #1	Byte #2	Byte #3	Byte #4	Byte #5	Byte #6	Byte #7

Figura 5: Trama estándar OBD-II.

El contenido de cada “Byte” dependerá de si la trama es de petición o de respuesta y del modo en el que se esté trabajando. La excepción es el Byte #0, que indica el número de bytes que contienen información OBD-II de la trama en cuestión. La existencia de este byte dependerá del protocolo sobre el que se envía el mensaje, ya que de los cinco protocolos que define el estándar OBD-II solo el protocolo CAN soporta ocho bytes de datos, el resto en cambio, solo soportan siete bytes de datos.

2.3.2.1 **Modo 01: Solicitud de datos de diagnóstico actuales**

El Modo 01 consiste en acceder a datos en tiempo real de valores analógicos o digitales de salidas y entradas a la ECU. Este modo también es llamado "flujo de datos". Trabajando en este modo es posible ver, por ejemplo, la temperatura de motor o el voltaje generado por una sonda lambda en tiempo real.

La información en este modo incluye un parámetro de identificación (PID o *Parameter ID*), este parámetro indica al sistema de diagnóstico de a bordo la información específica requerida. Existe un listado amplio de PIDs para este modo, no obstante no todos los PIDs son soportados por todos los sistemas. Por este motivo el PID $00 incluye un código que indica para cada ECU qué PIDs soporta. El PID $00 indica los PIDs soportados desde el PID $01 al $20, a su vez el PID $20 indica los PIDs soportados desde el PID $21 hasta el $40, y así sucesivamente. Todas las ECUs deben soportar por defecto el PID $00.

En la Tabla 1 se muestra el formato que debe tener la trama OBD-II para realizar peticiones en el Modo 01, sin tener en cuenta el byte cero que indica la longitud efectiva como ya comentamos anteriormente.

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una petición en el Modo 01	01
#2	Valor hexagesimal del PID requerido	XX
#3	(Opcional) Valor hexagesimal del segundo PID requerido	XX
#4	(Opcional) Valor hexagesimal del tercer PID requerido	XX
#5	(Opcional) Valor hexagesimal del cuarto PID requerido	XX
#6	(Opcional) Valor hexagesimal del quinto PID requerido	XX
#7	(Opcional) Valor hexagesimal del sexto PID requerido	XX

Tabla 1: Formato de petición OBD-II en Modo 01.

Como se puede observar, es posible solicitar hasta seis PIDs diferentes en una misma trama, aunque no es lo más común. El formato de la respuesta en Modo 01 es el que se muestra en la Tabla 2, en el caso de haberse requerido más de un PID habría que enviar una respuesta por cada PID solicitado. Para indicar que estamos respondiendo a una solicitud de un modo en concreto el valor hexagesimal será el valor del modo en cuestión, 1 en este caso, más 40 en hexagesimal.

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una respuesta en el Modo 01	41
#2	Valor hexagesimal del PID al que se está respondiendo	XX
#3	Dato A	XX

#4	Dato B	XX
#5	Dato C	XX
#6	Dato D	XX
#7	Sin uso	00

Tabla 2: Formato de respuesta OBD-II en Modo 01.

Donde Dato A, Dato B, Dato C y Dato D contienen los datos de diagnóstico referentes al PID solicitado. No siempre tienen que usarse los cuatro bytes de datos, por ejemplo para responder al PID $0D (velocidad del vehículo) solo se emplea el Dato A quedando el resto sin uso.

La Tabla 3 recoge los principales PIDs del estándar OBD-II según la definición de SAE J1979. Junto con su descripción se da la respuesta esperada para cada PID, junto con información sobre cómo traducir la respuesta en datos significativos. Además establece que no puede haber PIDs personalizados definidos por el fabricante que no están definidos en la norma OBD-II.

PID	Bytes	Descripción	Min.	Max.	Uds.	Fórmula
00	4	PIDs soportados [01-20]	-	-	-	32 Bits: BitX=0 > PIDX Soportado BitX=1 > PIDX No soportado
01	4	Estado de MIL y número de DTCs almacenados.	-	-	-	Bits codificados
03	2	Estado del sistema de combustible	-	-	-	Bits codificados
04	1	Valor de carga del motor	0	100	%	A×100/255
05	1	Temperatura del refrigerante	-40	215	ºC	A-40
0A	1	Presión del combustible	0	765	kPa	A×3
0C	2	Revoluciones por minuto del motor	0	16.383	rpm	((A×256)+B)/4
0D	1	Velocidad del vehículo	0	255	Km/h	A
0F	1	Temperatura del aire de entrada	-40	215	ºC	A-40
11	1	Posición del acelerador	0	100	%	A×100/255
13 … 1B	2	Voltaje en los distintos sensores de oxígeno	0	1.275	V	A/200 (B-128)×100/128

1C	1	Estándar OBD conforme al vehículo	-	-	-	Bits Codificados
1F	2	Tiempo transcurrido desde el arranque	0	65.535	seg.	(A×256)+B
20	4	PIDs soportados [21-40]	-	-	-	32 Bits: BitX=0 > PIDX Soportado BitX=1 > PIDX No soportado
21	2	Distancia recorrida con el indicador MIL encendido	0	65.535	Km	(A×256)+B
24 … 2B	4	Voltaje equivalente de la sonda lambda	0	8	V	((A×256)+B)×R ((C×256)+D)×R Con R=2/65535
2C	1	Recirculación de gases de escape o EGR	0	100	%	A×100/255
2E	1	Depuración por evaporación	0	100	%	A×100/255
2F	1	Nivel de combustible	0	100	%	A×100/255
31	2	Distancia recorrida desde el último limpiado de errores	0	65535	km	(A×256)+B
34 … 3B	4	Corriente equivalente de la sonda lambda	-128	128	mA	((A×256)+B) /32.768 ((C×256)+D)/128
3C … 3F	2	Temperatura del catalizador	-40	6.513	ºC	((A×256)+B)/10-40
40	4	PIDs soportados [41-60]	-	-	-	32 Bits: BitX=0 > PIDX Soportado BitX=1 > PIDX No soportado
42	2	Módulo de control de voltaje	0	65.535	V	(A×256)+B
46	1	Temperatura ambiente	-40	215	ºC	A-40
4D	2	Tiempo de marcha mientras el indicador MIL esta encendido	0	65.535	min.	(A×256)+B
51	1	Tipo de combustible	-	-	-	Bits codificados
5B	1	Nivel de batería híbrida	0	100	%	A×100/255

Tabla 3: Principales PIDs OBD-II según la SAE.

Donde A, B, C y D representan los bytes de datos Byte A, Byte B, Byte D y Byte C, el número de bytes de información usados en cada PID viene representado en la columna Bytes.

2.3.2.2 **Modo 02: Acceso a Cuadro de Datos Congelados**

Esta es una función muy útil de OBD-II porque la ECU toma una muestra de todos los valores relacionados con las emisiones, en el momento exacto de ocurrir un fallo. De esta manera, al recuperar estos datos, se pueden conocer las condiciones exactas en las que ocurrió dicho fallo. Normalmente solo existe un cuadro de datos que corresponde al primer fallo detectado aunque esto depende de la ECU.

El Modo 02 acepta los mismos PIDs que el Modo 01, con el mismo significado. La única diferencia entre estos dos modos es que en el primer caso leemos datos actuales y en el segundo datos almacenados del momento en el que ocurrió algún tipo de fallo.

El formato de las peticiones y respuestas en este modo es muy similar a las del Modo 01, con la única diferencia del indicador de cuadro de datos. Este indicador identifica el cuadro de datos congelados para evitar confusiones en caso de existir más de un cuadro. Dicho formato queda recogido en las siguientes tablas:

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una petición en el Modo 02	02
#2	Valor hexagesimal del PID requerido	XX
#3	Número de Cuadro	XX
#4	(Opcional) Valor hexagesimal del segundo PID requerido	XX
#5	(Opcional) Número de Cuadro	XX
#6	(Opcional) Valor hexagesimal del tercero PID requerido	XX
#7	(Opcional) Número de Cuadro	XX

Tabla 4: Formato de petición OBD-II en Modo 02.

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una respuesta en el Modo 02	42
#2	Valor hexagesimal del PID al que se está respondiendo	XX
#3	Número de Cuadro	XX
#4	Dato A	XX
#5	Dato B	XX
#6	Dato C	XX
#7	Dato D	XX

Tabla 5: Formato de respuesta OBD-II en Modo 02.

2.3.2.3 **Modo 03: Códigos de diagnóstico almacenados**

Este modo permite extraer de la memoria de la ECU todos los Códigos de Diagnóstico de Error o DTCs (*Data Trouble Code*) almacenados. Los Códigos de Diagnósticos de Error (DTCs) consisten en códigos de tres dígitos precedidos por un identificador alfanumérico. Cuando la ECU reconoce e identifica un problema, se almacena en la memoria un DTC que corresponde a ese fallo. Estos códigos tienen por objetivo ayudar al usuario a determinar la causa fundamental de un problema. En la siguiente figura se ilustra el formato DTC recomendado por la normativa SAE.

Figura 6: Formato DTCs.

Aunque existe una tabla con todos los códigos estandarizados, no impide que cada fabricante añada sus propios códigos para el control de parámetros o errores que no están tabulados en los códigos estándares.

El procedimiento para acceder a dichos DTCs es el siguiente, primero se envía una petición solicitando a la ECU todos los DTCs almacenados (Tabla 6). Acto seguido, esta responderá con un mensaje por cada DTC almacenado (Tabla 7), o con ceros en caso de no existir ningún código de error.

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una petición de los DTCs almacenados en Modo 03	03

Tabla 6: Formato de petición OBD-II en Modo 03.

El valor del resto de bytes no tiene importancia, si bien es cierto que la norma recomienda rellenar con todos los bytes a “$00” o “$55” en hexagesimal.

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una respuesta en el Modo 03	43
#2	Número de DTCs almacenados	XX
#3	DTC codificado: High Byte (HB)	XX
#4	DTC codificado: Low Byte (LB)	XX
#5	Sin uso	00
#6	Sin uso	00
#7	Sin uso	00

Tabla 7: Formato de respuesta OBD-II en Modo03.

Como se muestra en la tabla anterior los DTCs se encuentran codificados en dos bytes que llamaremos HB (*High Byte*) y LB (*Low Byte*). En el byte HB se encuentran codificados los tres primeros caracteres del DTC y en el byte LB los dos restantes, dicha codificación se ilustra en la Tabla 8:

Carácter	1	2	3	4	5
Bits	HB7-HB6	HB5-HB4	HB3-HB0	LB7-LB4	HB3-HB0
Código	00 = P 01 = C 10 = B 11 = U	00 = 0 01 = 1 10 = 2 11 = 3	xxxx su valor en Hexagesimal	xxxx su valor en Hexagesimal	xxxx su valor en Hexagesimal

Tabla 8: Codificación de los DTCs.

Por ejemplo, si el DTC es P0217, su codificación sería la siguiente:

- High Byte = “00000010”
  - P = “00”
  - 0 = “00”
  - 2 = “0010”
- Low Byte = “00010111”
  - 1 = “0001”
  - 7 = “0111”

2.3.2.4 **Modo 04: Borrado de DTCs y Cuadro de Congelados**

Con este modo se pueden borrar todos los códigos almacenados en la Unidad de Control Electrónico, incluyendo los DTCs y el cuadro de datos congelados. Este modo también apaga el

indicador luminoso MIL, aunque si se borraron los códigos sin haberse corregido el error, estos volverán a aparecer.

El funcionamiento en este modo se resume en una petición y una respuesta confirmando que el borrado o reseteo se ha llevado a cabo con éxito. En las Tablas 9 y 10 se ilustran estos mensajes de petición y respuesta en caso afirmativo.

Byte	Parámetro	Valor Hex.
#1	Solicita el borrado o reseteo de todos los DTCs almacenados y del cuadro de congelados	04

Tabla 9: Formato de petición OBD-II en Modo 04.

Byte	Parámetro	Valor Hex.
#1	Indica que el borrado se ha realizado correctamente	44

Tabla 10: Formato de respuesta afirmativa en Modo 04.

En el caso de que no haya sido posible el borrado de estos códigos el mensaje que devuelve la ECU es el siguiente:

Byte	Parámetro	Valor Hex.
#1	Identificador de respuesta negativa	7F
#2	Modo en el que se ha solicitado la acción	04
#3	Código de respuesta negativa: Condiciones Incorrectas	22

Tabla 11: Formato de respuesta negativa en Modo 05.

2.3.2.5 **Modo 05: Resultados del test de los sensores de oxígeno**

Este modo devuelve los resultados de las pruebas realizadas a los sensores de oxígeno para determinar el funcionamiento de los mismos y la eficiencia del convertidor catalítico, vital para el control de las emisiones del vehículo y para el correcto funcionamiento del mismo.

El test devuelve unos voltajes dependiendo de la mezcla aire-combustible, los valores obtenidos de la prueba son los siguientes:

- Umbral de voltaje en el sensor de oxígeno de rico a pobre.
- Umbral de voltaje en el sensor de oxígeno de pobre a rico.
- Voltaje alto/bajo en el sensor para el cambio.
- Tiempo de cambio de rico a pobre y de pobre a rico.
- Voltajes máximos y mínimos.
- Tiempos de transición entre los distintos sensores.

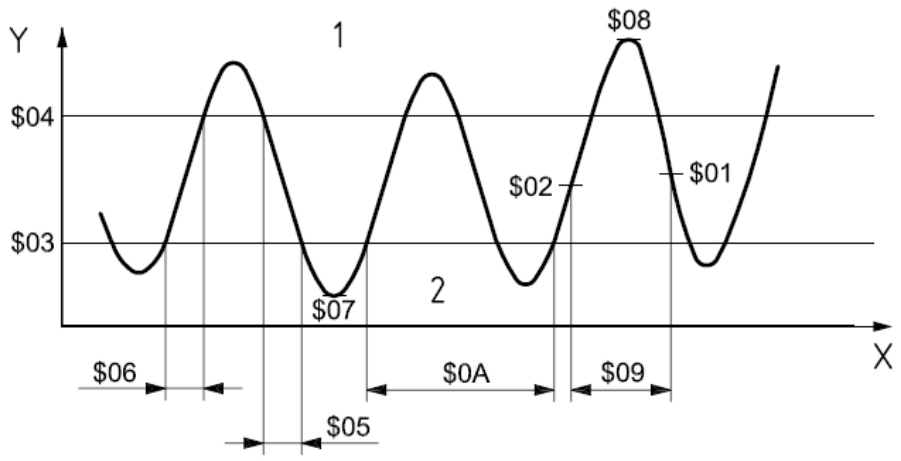


Figura 7: Resultados del test de los sensores de oxígeno.

2.3.2.6 **Modo 06: Resultado de las pruebas de control no permanente**

Este modo permite obtener los resultados de todas las pruebas OBD-II, las pruebas más comunes son las siguientes:

- Errores de combustión: bujías, arranque, inyectores, etc.
- Sistema de combustible: sensores de oxígeno, sensores de ciclo cerrado, retroalimentación y sistema de encendido.
- Catalizador.
- Calentamiento del catalizador.
- Sistema evaporativo.
- Sistema segundario de aire.
- Sensores de oxígeno.
- Calentamiento del sensor de oxígeno.
- Recirculación de gases de escape o EGR.

2.3.2.7 **Modo 07: Mostrar códigos de diagnóstico pendientes**

Este modo permite leer de la memoria de la ECU todos los códigos de diagnóstico pendientes que no hayan sido reparados o borrados previamente.

2.3.2.8 **Modo 08: Modo de control de a bordo**

Este modo permite realizar la prueba de actuadores. Con esta función, el personal autorizado puede activar y desactivar actuadores como bombas de combustible, válvula de relentí, entre otros actuadores del sistema automotriz.

2.3.2.9 **Modo 09: Información del vehículo**

Este modo es opcional, no todos los vehículos lo soportan. Básicamente este modo permite obtener información del vehículo como el número de serie, número de calibrado y verificación, información de la ECU, etc.

2.3.3 **Acceso a la información OBD-II**

A principios de los 80, cuando se extendió el uso de este sistema de diagnosis, cada fabricante era libre de incorporar su propio conector y utilizar los códigos de error que quisiera. Esto dificultaba mucho la utilización de este sistema para las reparaciones, ya que la inversión que requería en los talleres mecánicos era altísima y poco práctica (debían disponer de muchos lectores y de muchas tablas de códigos). Para que el uso de este sistema fuera practico y viable, en 1996, se llegó a un consenso entre los fabricantes y se estandarizaron los códigos y el conector.

2.3.3.1 **Detección de averías en el cuadro de instrumentos**

Como ya se ha comentado, el cuadro de instrumentos se dispone de un testigo luminoso de color amarillo con el ideograma de un motor conocido como MIL. Este testigo se enciende al accionar la llave de contacto y debe permanecer encendido unos 2 segundos después del arranque del motor. Esta es la forma en que se verifica el correcto funcionamiento del testigo, por parte del técnico o del usuario. Podemos conocer la existencia de averías mediante el siguiente código de luces:

- Destellos ocasionales indican averías de tipo esporádico.
- Cuando el testigo permanece encendido constantemente existe una avería de naturaleza seria que puede afectar a la emisión de gases o a la seguridad del vehículo.
- En el supuesto que se detecte una avería muy grave susceptible de dañar el motor o afectar a la seguridad, el testigo de averías luce de manera intermitente. En este caso se deberá parar el motor.

En caso de detectar algunas de estas señales luminosas es aconsejable que se acuda a un taller para que revisen el automóvil. Una vez en el taller, el equipo de mecánicos le facilitará la información sobre el error almacenado. Para ello será necesario un conector adecuado, conocer los modos de medición, conocer los códigos de avería y disponer de un sistema de lectura.

2.3.3.2 **Conector de diagnóstico estándar**

El conector de diagnóstico o DLC (*Data Link Connector*) permite la conexión con el sistema de lectura. Dicho conector deberá cumplir las especificaciones establecidas en la normativa ISO 15031-3. El conector DLC consta de 16 contactos como se observa en la Figura 8.

Figura 8: Conector OBD-II estándar.

Como se puede apreciar en la figura OBD-II puede estar implementado sobre varios protocolos, en futuros apartados se estudiaran cada uno de ellos, en especial el protocolo CAN.

2.3.3.3 **Sistemas de lectura**

Existen dos posibilidades a la hora de leer los códigos de averías:

- Conectar el cable OBD-II a un PC mediante una interfaz, para ello será necesario que dicho PC disponga del software necesario para realizar las lecturas. Entre estos cabe destacar: OBD-II ScanTool, ScanMaster o EasyOBDII. En la Figura 9 podemos ver el entorno gráfico de ScanMaster.
- Haciendo uso de una herramienta portátil de diagnóstico, más conocido como Scanner. Estos sistemas realizan el tratamiento de la información del OBDII del vehículo y muestran en su pantalla los códigos de error. En la Figura 10 se plasman algunas de estas herramientas de diagnóstico.

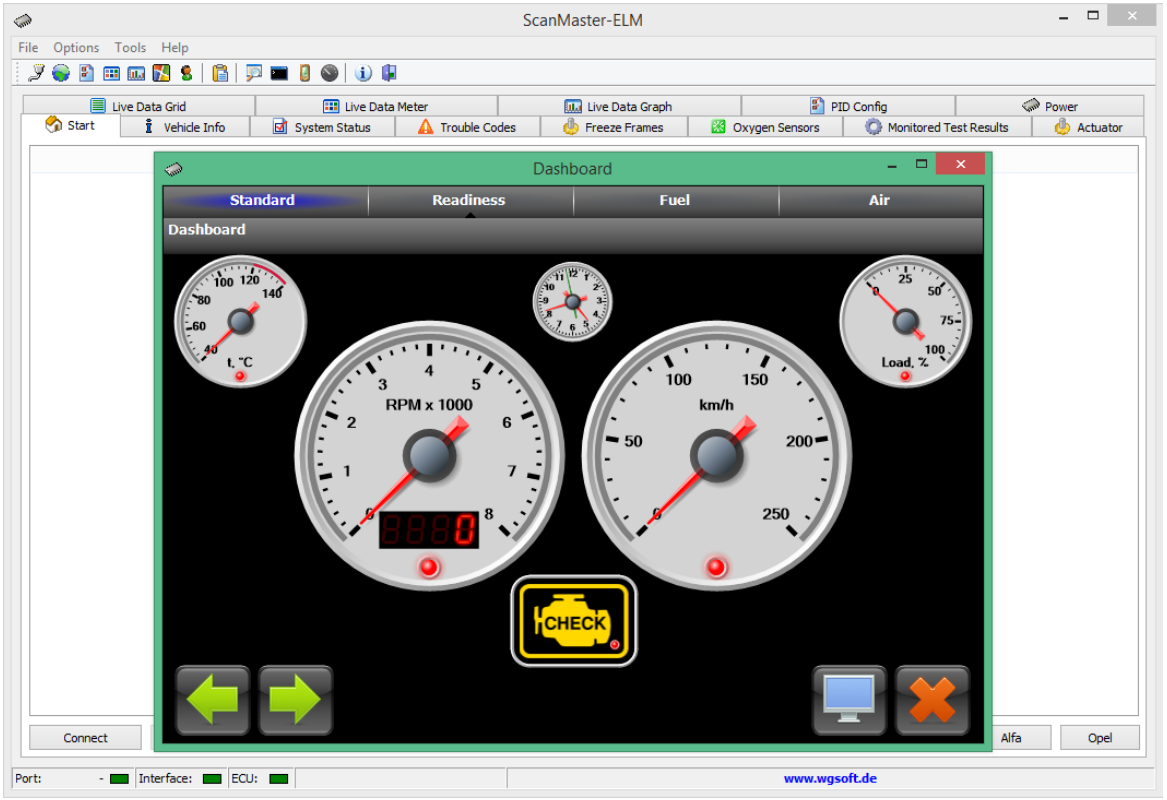


Figura 9: Entorno gráfico de ScanMaster.

Figura 10: Herramienta portátil OBD-II.

La normativa establece unos requisitos mínimos que deben cumplir todos los sistemas de lectura:

- Determinación automática de la interfaz de comunicación empleada.
- Determinación automática y exhibición de la disponibilidad de información sobre inspección y mantenimiento.
- Exhibición de códigos de diagnóstico relacionados con la emisión, datos en curso, congelado de datos e información del sensor de oxígeno.
- Borrado de los DTC, del congelado de datos y del estado de las pruebas de diagnóstico.

2.4 **Protocolos OBD-II**

Según el estándar, los sistemas OBD-II pueden emplear los siguientes protocolos para el intercambio de mensajes OBD-II:

- SAE J1850 PWM: protocolo de diagnóstico usado mayormente en vehículos Ford. Usa señales diferenciales y tiene una velocidad de transferencia de 41,6 Kbps.
- SAE J1850 VPW: protocolo de diagnóstico usado en los vehículos de General Motors. Tiene una velocidad de comunicación de 10,4 Kbps.
- ISO 9141/14230: antiguo protocolo usado en vehículos europeos entre 2000 y 2004.
- ISO 14230-4 (KWP2000): protocolo muy común en vehículos a partir de 2003. Básicamente se utiliza para la diagnosis *off-board* de centralitas electrónicas de vehículos y para cargar en las mismas versiones nuevas de software, lo que se conoce como reprogramar o "flashear". También funciona a una velocidad de 10,4 Kbps.
- ISO 11898/15765 (CAN): el protocolo CAN para el bus de diagnóstico comenzó a estar presente en vehículos desde 2003. A partir de 2008 es obligatorio en los vehículos estadounidenses.

La Tabla 11 ilustra la organización según el modelo OSI (*Open System Interconnection*) de los protocolos anteriormente nombrados.

Niveles OSI	**OBD-II**			
Físico (Nivel 1)	SAE J1850	ISO 9141-2	ISO 14230-1	ISO 11898 ISO 15765-4
Enlace (Nivel 2)	SAE J1850	ISO 9141-2	ISO 14230-1	ISO 11898 ISO 15765-4
Red (Nivel 3)	---	---	---	ISO 15765-2 ISO 15765-4
Transporte (Nivel 4)	---	---	---	---
Sesión (Nivel 5)	---	---	---	---
Presentación (Nivel 6)	---	---	---	ISO 15765-4
Aplicación (Nivel 7)	ISO 15031-5	ISO 15031-5	ISO 15031-5	ISO 15031-5

Tabla 12: Protocolos según el modelo OSI para el estándar OBD-II.

En los siguientes subapartados se realiza un estudio de los protocolos anteriormente nombrados, haciendo especial hincapié en el protocolo CAN.

2.4.1 **SAE J1850 (PWM y VPW)**

El estándar SAE para las clases A y B (velocidad de transmisión baja y media), es una combinación del SCP de Ford y del protocolo de Clase 2 de General Motors. Fue aprobado por la SAE en 1988 y revisado finalmente en 1994.

Existen dos versiones cuya diferencia radica en la codificación de bit y la velocidad de transmisión:

- La versión más lenta emplea una codificación modulada por ancho de pulso variable o VPW (*Variable Pulse Width*), alcanzando 10,4 Kbps y transmite sobre un único cable referido a masa.
- La versión más rápida emplea una codificación modulada por ancho de pulso o PWM (*Pulse Width Modulation*), consiguiendo 41,6 Kbps transmitiendo en modo diferencial sobre dos cables.

Como modo de acceso al medio emplea CSMA/CR (*Carrier Sense Multiple Access / Collision Resolution*), lo que significa que cualquier módulo puede transmitir si detecta que el bus está desocupado. Si más de un módulo intenta transmitir al mismo tiempo, un proceso de arbitraje determinará cuál de ellos continuará transmitiendo y quién deberá reintentarlo después.

La principal aportación de este protocolo fue la inclusión de las respuestas de los nodos destinatarios dentro de la propia trama emitida desde el nodo origen. En concreto permite: respuesta de un byte desde un simple destinatario, respuestas concatenadas de un byte desde múltiples destinatarios y respuesta de múltiples bytes desde un simple destinatario.

Este protocolo fue el primero en ser usado de forma masiva, actualmente aun esta en uso aunque tendiendo a desaparecer.

2.4.1.1 **Nivel Físico**

Los símbolos empleados para trasmitir información a través del bus dependen del tipo de modulación.

Si se transmite usando VPW, tendremos símbolos como los ilustrados en la Figura 11 con las siguientes características:

- El bus alterna entre pasivo y activo para cada bit.
- Cuando el bus se encuentra en estado activo, los pulsos largos dominan a los pulsos cortos. En caso de encontrarse en estado pasivo, los pulsos cortos dominan a los pulsos largos.

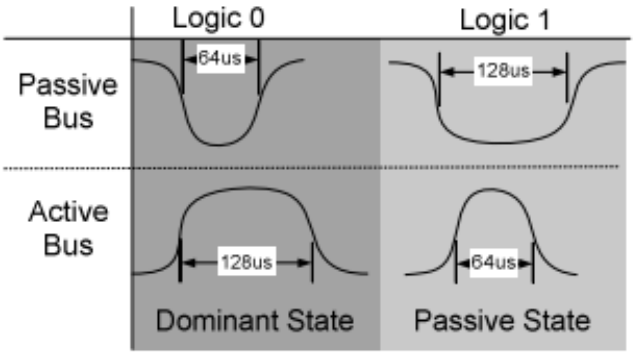


Figura 11: Símbolos SAE J1850 VPW.

En caso de transmitir usando una modulación PWM, se tendrán símbolos como los ilustrados en la Figura 12 con las siguientes características:

- Tiempo de bit fijo. Al principio de cada tiempo de bit se comenzará a transmitir el pulso corto o el pulso largo.
- El pulso largo domina al pulso corto.

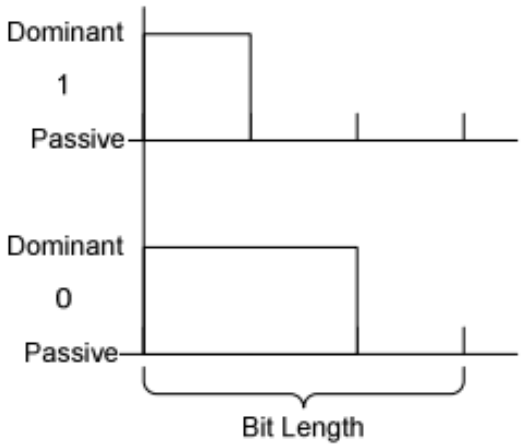


Figura 12: Símbolos SAE J1850 PWM.

2.4.1.2 **Nivel de enlace**

La estructura del mensaje es común a los dos métodos de modulación VPW y PWM. En la Figura 13 se muestra el formato de trama para el protocolo SAE J1850, a continuación se describe brevemente la utilidad de cada uno de sus campos:

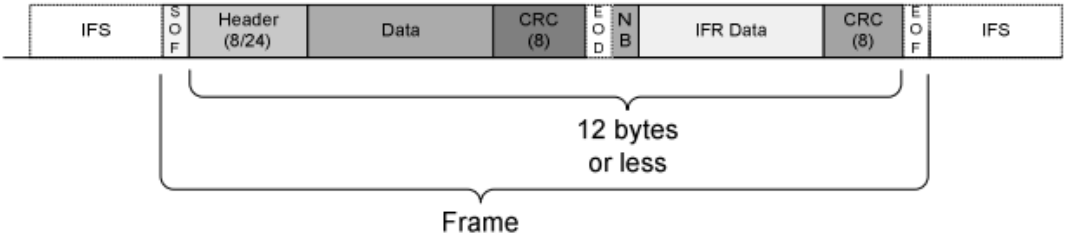


Figura 13: Formato de trama SAE J1850.

- **SOF (*Start of Frame*):** Indica el comienzo de la trama.
- **Cabecera (*Header*):** de uno a tres bytes que contienen:
  - Prioridad (3 bits).
  - Longitud de cabecera (1 bit).
  - IFR (*In-Frame Response*) Respuesta en la trama (1 bit).
  - Modo de dirección (1 bit).
  - Tipo de mensaje (2 bits).
  - Los dos bytes adicionales son para la dirección destino (8 bits) y la dirección origen (8 bits).
- **Bytes de datos:** máximo 7 bytes.
- **CRC (*Cyclical Redundancy Check*):** información redundante para el control de errores.
- **NB (*Normalization Bit*):** Bit de normalización.
- **Datos IFR.**
- **IFR CRC:** control de errores de los datos IFR.

2.4.2 ISO 9141-2

El protocolo 9141-2 está basado en la comunicación serie similar a RS-232 (*Recommended Standart 232*), sin embargo, los niveles de señal son diferentes y sobre una sola línea de comunicación bidireccional. Este protocolo tiene una tasa de datos serie asíncrona de 10,4 Kbps.

Las comunicaciones en serie usan niveles de tensión altos y bajos para transmitir bits, un cero binario está representado por un nivel de tensión de cero voltios y un uno binario está representado por un nivel de tensión 12 voltios.

Las tramas ISO 9141-2 son exactamente iguales que las utilizadas en el protocolo SAE J1850 (véase la Figura 13), la única diferencia entre ambos protocolos se encuentra en la capa física.

2.4.3 ISO 14230

El estándar ISO 14230, más conocido como KWP2000 (*Keyword Protocol 2000*), es un protocolo de comunicación serie muy similar al ISO 9141. Al igual que el protocolo ISO 9141

basado en un bus bidireccional sobre una única línea (llamada *K-line*), opcionalmente puede haber una segunda línea (llamada *L-line*) para las señales de llamada o *wakeup*. La tasa de datos puede oscilar entre los 1,2 y los 10,4 kbps.

Al igual que el protocolo ISO 9141, sus tramas están basadas en las de la normativa SAE J1850, la única diferencia está en el campo de datos que puede llegar a contener hasta 255 bytes.

2.4.4 CAN

El sistema de bus serie CAN, acrónimo del inglés *Controller Area Network*, nació en febrero de 1986, cuando la firma alemana Robert Bosch GmbH lo presentó en el congreso de la Sociedad de Ingeniería de la Automoción. Inicialmente se pensó en él como bus de campo, pero donde realmente encontró utilidad fue en el sector del automóvil. El Mercedes Clase E fue el primer coche en incorporar el bus CAN, 10 años después (1992). El sistema estaba compuesto por dos redes CAN, una red de alta velocidad, en la que se comunicaban las ECUs del motor, la unidad de control de la caja de cambios y el tablero de instrumentos; y una red de baja velocidad, para el control del aire acondicionado y de los dispositivos electrónicos internos, conectando ambas redes CAN.

La implementación realizada por Mercedes-Benz propició que otros fabricantes de automóviles comenzaran a utilizar redes CAN en sus modelos de lujo, por ejemplo BMW, Jaguar, Volvo, Saab y Volskwagen, más tarde se agregaron a la lista Fiat y Renault. Desde entonces, CAN se ha convertido en uno de los protocolos líderes en la utilización del bus serie.

2.4.4.1 Características básicas

CAN se basa en el modelo productor/consumidor (conocido en inglés como *publish/subscribe*), el cual es un concepto, o paradigma de comunicaciones de datos, que describe una relación entre un productor y uno o más consumidores. CAN es un protocolo orientado a mensajes, es decir la información que se va a intercambiar se descompone en mensajes, a los cuales se les asigna un identificador y se encapsulan en tramas para su transmisión. Cada mensaje tiene un identificador único dentro de la red, con el cual los nodos deciden aceptar o no dicho mensaje.

Dentro de sus principales características se encuentran:

- **Económico y sencillo:** dos de las razones que motivaron su desarrollo fueron precisamente la necesidad de economizar el coste monetario y el de minimizar la complejidad del cableado, por parte del sector automovilístico.
- **Estandarizado:** se trata de un estándar definido en las normas ISO.
- **Medio de transmisión adaptable:** el cableado es muy reducido en comparación con otros sistemas. Además, a pesar de que por diversas razones el estándar de hardware de

transmisión sea un par trenzado de cables, el sistema de bus CAN también es capaz de trabajar con un solo cable. Esta particularidad es empleada en diversos tipos de enlaces, como los enlaces ópticos o los enlaces de radio.

- **Estructura definida:** la información que circula entre las unidades a través de los dos cables son paquetes de bits (0's y 1's) con una longitud limitada y con una estructura definida de campos que conforman el mensaje.
- **Programación sencilla:** basada en la escritura de registros de los dispositivos CAN.
- **Número de nodos:** el número máximo de módulos no está limitado por la especificación básica y depende de las características de los controladores CAN. Las especificaciones de buses de campo lo limitan a un máximo de 64 nodos.
- **Garantía de tiempos de latencia:** CAN aporta la seguridad de que se transmitirá cierta cantidad de datos en un tiempo concreto, es decir, que la latencia de extremo a extremo no excederá una cantidad específica de tiempo. Además, la transmisión siempre será en tiempo real.
- **Optimización del ancho de banda:** los métodos utilizados para distribuir los mensajes en la red, como el envío de estos según su prioridad, contribuyen a un mejor empleo del ancho de banda disponible.
- **Desconexión autónoma de nodos defectuosos:** si un nodo de red cae, sea cual sea la causa, la red puede seguir funcionado, ya que es capaz de desconectarlo o aislarlo del resto. De forma contraria, también se pueden añadir nodos al bus sin afectar al resto del sistema, y sin necesidad de reprogramación.
- **Velocidad flexible:** ISO define dos tipos de redes CAN, una red de alta velocidad (de hasta 1 Mbps) definida por la ISO 11898-2, y una red de baja velocidad tolerante a fallos (menor o igual a 125 Kbps) definida por la ISO 11898-3.
- **Relación velocidad-distancia:** al punto anterior habría que añadir que la velocidad también depende de la distancia hasta un máximo de 1000 metros (aunque se puede aumentar la distancia con bridges o repetidores), como podemos ver en la Figura 14.

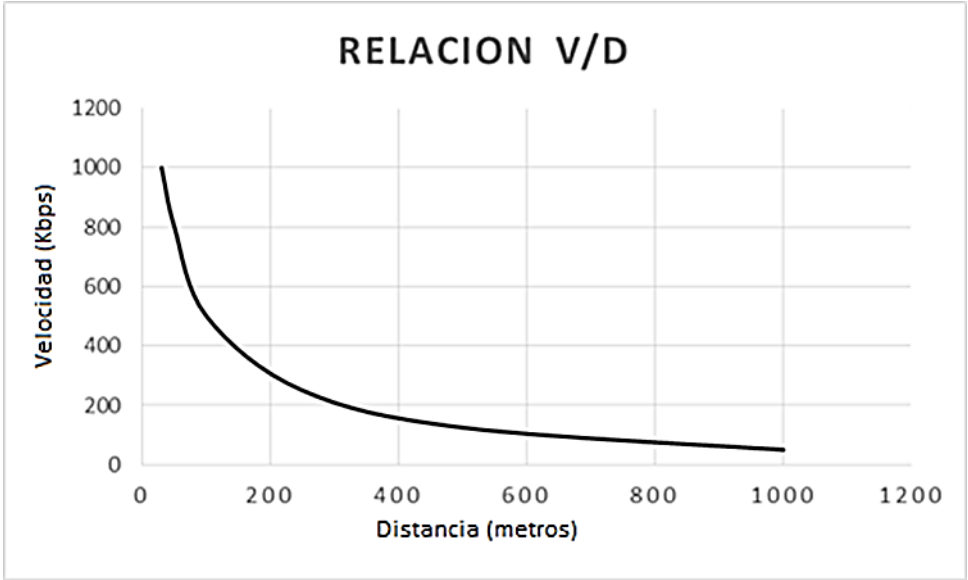


Figura 14: Relación velocidad-distancia.

- **Orientado a mensajes:** se trata de un protocolo orientado a mensajes, y no a direcciones, es decir, la información que se va a intercambiar se descompone en mensajes, a los cuales se les asigna un identificador y son encapsulados en tramas para su transmisión. Cada mensaje tiene un identificador único dentro de la red, a partir del cual los nodos deciden aceptar o no dicho mensaje. Además están priorizados.
- **Multidifusión (*Multicast*):** permite que todos los nodos puedan acceder al bus de forma simultánea con sincronización de tiempos.
- **Medio compartido (*Broadcasting*):** la información es enviada en la red a todos los destinos de forma simultánea. Así que los destinos habrán de saber si la información les concierne o deben rechazarla.
- **Detección y señalización de errores:** CAN posee una gran capacidad de detección de errores, tanto temporales, como permanentes, lograda a través de cinco mecanismos de detección, 3 de ellos a nivel de mensaje y 2 a nivel de bit. Los errores además pueden ser señalizados.
- **Retransmisión automática de tramas erróneas:** junto a la detección y señalización de errores la retransmisión automática de tramas erróneas aporta la integridad de los datos. Además ambos procesos son transparentes al usuario.
- **Jerarquía multimaestro:** CAN es un sistema multimaestro en el cual puede haber más de un maestro (*master*) al mismo tiempo y sobre la misma red, es decir, todos los nodos son capaces de transmitir, hecho que permite construir sistemas inteligentes y redundantes.

2.4.4.2 **Topología de red**

La topología básica de una red CAN es la mostrada en la Figura 15, al tratarse de un bus multimaestro, todos los nodos comparten las mismas características, distinguiéndose entre ellos a través del protocolo.

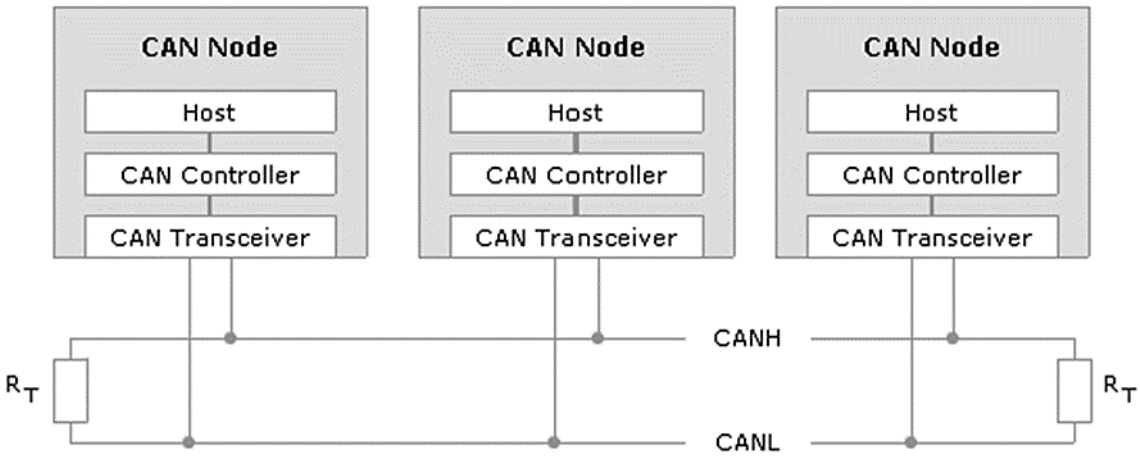


Figura 15: Topología de red del bus CAN.

Un nodo CAN se caracteriza por tres elementos:

- **Host:** Suele tratarse de un microcontrolador que interpreta los mensajes recibidos y decide qué mensajes se han de transmitir.
- **CAN Controller:** Se encarga de traducir y manipular los mensajes recibidos y enviados por el host según el protocolo. Gestiona todas las señales del bus y la velocidad de transmisión.
- **CAN Transceiver:** Se encarga de adaptar el nivel de las señales bidireccionalmente, es decir, convierte la señal lógica en modo diferencial (como se verá más adelante) para acceder al bus y transforma las señales diferenciales del bus a señales lógicas para que el CAN Controller las interprete.

2.4.4.3 **Nivel Físico**

La capa física de CAN, es responsable de la transferencia de bits entre los distintos nodos que componen la red. Define aspectos como niveles de señal, codificación, sincronización y tiempos en que los bits se transfieren al bus.

En la especificación original de CAN, la capa física no fue definida, permitiendo diferentes opciones para la elección del medio y niveles eléctricos de transmisión. Las características de las señales eléctricas en el bus, fueron establecidas más tarde por el estándar ISO 11898-2 para

las aplicaciones de alta velocidad y por el estándar ISO 11898-3 para las aplicaciones de baja velocidad.

En ambos casos, la codificación de bits se realiza por el método NRZ (*Non-Return-to Zero*) que se caracteriza por que el nivel de señal puede permanecer constante durante largos periodos de tiempo y habrá que tomar medidas para asegurarse de que el intervalo máximo permitido entre dos señales no es superado. Esto es importante para la sincronización (*Bit Timing*).

Este tipo de codificación requiere poco ancho de banda para transmitir, pero en cambio, no puede garantizar la sincronización de la trama transmitida. Para resolver esta falta de sincronismo se emplea la técnica del *bit stuffing*, esto es, cada 5 bits consecutivos con el mismo estado lógico en una trama (excepto del delimitador de final de trama y el espacio entre tramas), se inserta un bit de diferente polaridad, no perdiéndose así la sincronización. Por otro lado este bit extra debe ser eliminado por el receptor de la trama, que sólo lo utilizará para sincronizar la transmisión.

No hay flanco de subida ni de bajada para cada bit, durante el tiempo de bit hay bits dominantes (“0”) y recesivos (“1”) y disminuye la frecuencia de señal respecto a otras codificaciones.

2.4.4.3.1 Estándar ISO 11898-3 (baja velocidad)

Los nodos conectados en este bus interpretan dos niveles lógicos denominados dominante y recesivo, como se plasma en la Figura 16:

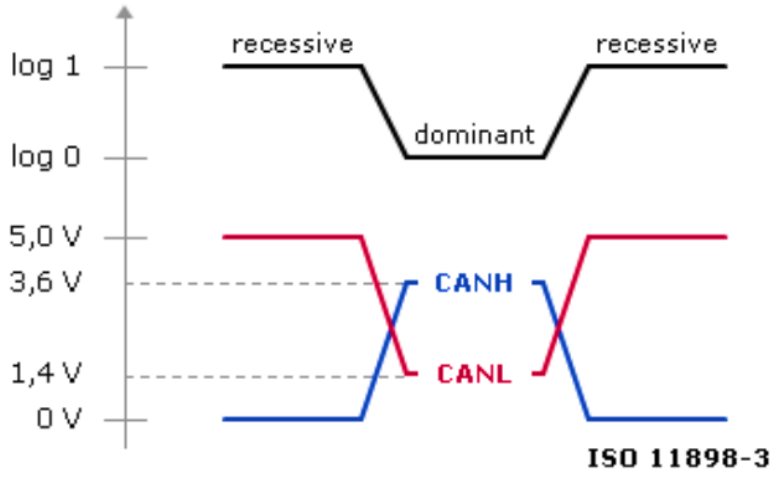


Figura 16: Niveles CAN de baja velocidad.

- **Dominante:** la tensión diferencial (CAN_H - CAN_L) es del orden de 2V, con CAN_H = 3.6V y CAN_L = 1.4V (nominales).
- **Recesivo:** la tensión diferencial (CAN_H - CAN_L) es del orden de 5V, con CAN_H = 0V y CAN_L = 5V (nominales).

A diferencia del bus de alta velocidad, el bus de baja velocidad requiere dos resistencias en cada transceptor: RTH para la señal CAN_H y RTL para la señal CAN_L.

Esta configuración permite al transceptor de bus de baja velocidad (*fault-tolerant*) detectar fallos en la red. La suma de todas las resistencias en paralelo, debe estar en el rango de 100-500Ω.

2.4.4.3.2 Estándar ISO 11898-2 (alta velocidad)

Los nodos conectados en este bus interpretan los siguientes niveles lógicos dominante y recesivo, como se plasma en la Figura 17:

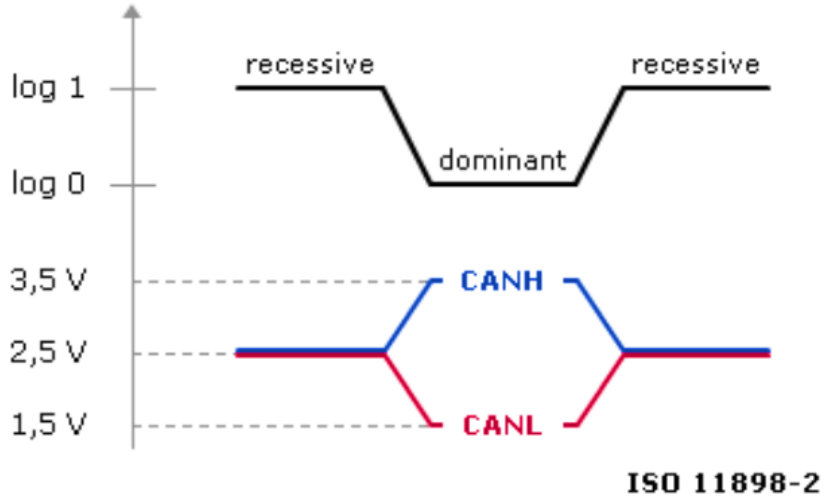


Figura 17: Niveles CAN de alta velocidad.

- **Dominante:** la tensión diferencial (CAN_H - CAN_L) es del orden de 2V, con CAN_H = 3.5V y CAN_L = 1.5V (nominales).
- **Recesivo:** la tensión diferencial (CAN_H - CAN_L) es del orden de 0V, con CAN_H = CAN_L = 2.5V (nominales).

El par de cables trenzados (CAN_H y CAN_L) constituyen una transmisión de línea. Si dicha transmisión de línea no está configurada con los valores correctos, cada trama transferida causa una reflexión que puede originar fallos de comunicación. Como la comunicación en el bus CAN fluye en ambos sentidos, ambos extremos de red deben de estar cerrados mediante una resistencia de 120Ω. Ambas resistencias deberían poder disipar 0.25W de potencia.

2.4.4.4 **Nivel de Enlace**

Esta capa es la responsable de controlar el flujo de información entre los nodos de la red. Es decir, se encarga de la transmisión de los bits en *frames* o tramas de información, se ocupa de que los mensajes lleguen al destino sin errores, controla las secuencias de transmisión, los acuses de recibo y si en determinado caso no se recibe un mensaje correctamente se encarga de retransmitirlo. Se puede dividir esta capa en dos subcapas que se ocupan de diferentes tareas:

- Subcapa MAC (*Medium Access Chanel*)
- Subcapa LLC (*Logical Link Control*)

2.4.4.4.1 Subcapa MAC

Esta subcapa representa el núcleo del protocolo CAN. Por un lado presenta los mensajes recibidos a la subcapa LLC y acepta los mensajes para ser transmitidos a dicha subcapa y por otro lado es responsable del mecanismo de arbitraje de acceso al medio.

Unas de las características que distingue a CAN con respecto a otras normas, es su técnica de acceso al medio denominada como CSMA/CD+CR o *Carrier Sense Multiple Access/Collision Detection + Collision Resolution* (Acceso Múltiple con detección de portadora, detección de colisión más resolución de colisión). Cada nodo debe vigilar el bus en un periodo sin actividad antes de enviar un mensaje (*Carrier Sense*) y además, una vez que ocurre el periodo sin actividad cada nodo tiene la misma oportunidad de enviar un mensaje (*Multiple Access*). En caso de que dos nodos comiencen a transmitir al unísono se detectará la colisión.

El método de acceso al medio utilizado en bus CAN añade una característica adicional: la resolución de colisión. En la técnica CSMA/CD utilizada en redes Ethernet ante colisión de varias tramas, todas se pierden. CAN resuelve la colisión con la supervivencia de una de las tramas que chocan en el bus. Además la trama superviviente es aquella a la que se ha identificado como de mayor prioridad. La resolución de colisión se basa en una topología eléctrica que aplica una función lógica determinista a cada bit, que se resuelve con la prioridad del nivel definido como bit de tipo dominante. Definiendo el bit dominante como equivalente al valor lógico '0' y bit recesivo al nivel lógico '1' se trata de una función AND de todos los bits transmitidos simultáneamente. Cada transmisor escucha continuamente el valor presente en el bus, y se retira cuando ese valor no coincide con el que dicho transmisor ha forzado. Mientras hay coincidencia la transmisión continúa, finalmente el mensaje con identificador de máxima prioridad sobrevive. Los demás nodos reintentarán la transmisión lo antes posible.

Se ha de tener en cuenta que la especificación CAN de Bosch no establece cómo se ha de traducir cada nivel de bit (dominante o recesivo) a variable física. Cuando se utiliza el par trenzado según ISO 11898 el nivel dominante es una tensión diferencial positiva en el bus, el

nivel recesivo es ausencia de tensión, o cierto valor negativo, (los transceptores no generan corriente sobre las resistencias de carga del bus). Esta técnica aporta la combinación de dos factores muy deseados en aplicaciones industriales distribuidas: la posibilidad de fijar con determinismo la latencia en la transmisión de mensajes entre nodos y el funcionamiento en modo multimaestro sin necesidad de gestión del arbitraje, es decir control de acceso al medio, desde las capas de software de protocolo. La prioridad queda así determinada por el contenido del mensaje que en CAN es un campo determinado, el identificador de mensaje, el que determina la prioridad.

En un bus único, un identificador de mensaje ha de ser asignado a un solo nodo concreto, es decir, se ha de evitar que dos nodos puedan iniciar la transmisión simultánea de mensajes con el mismo identificador y datos diferentes. El protocolo CAN establece que cada mensaje es único en el sistema, de manera que por ejemplo, si en un automóvil existe la variable "presión de aceite", esta variable ha de ser transmitida por un nodo concreto, con un identificador concreto, con una longitud fija concreta y coherente con la codificación de la información en el campo de datos.

En la Figura 18 se muestra un claro ejemplo de arbitraje, en este caso el nodo 3 transmite su dato y el resto se quedan a la escucha.

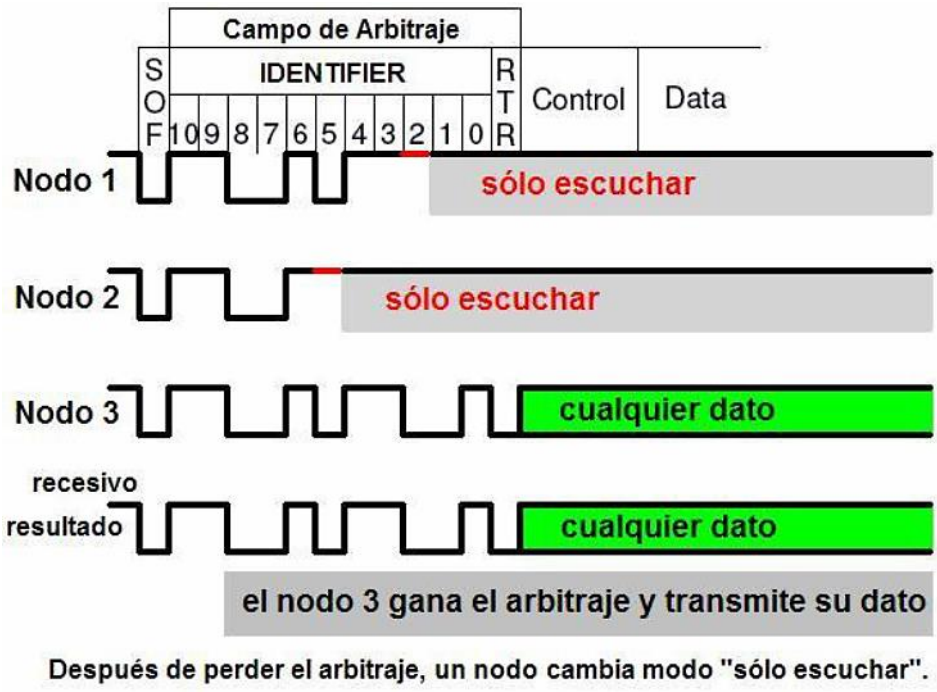


Figura 18: Ejemplo de arbitraje.

2.4.4.4.2 Subcapa LLC

La subcapa LLC describe la parte alta de la capa de enlace de datos y define las tareas independientes del método de acceso al medio. Asimismo, proporciona dos tipos de servicios de transmisión sin conexión al usuario de la capa LLC (*LLC user*):

- **Servicio de transmisión de datos sin reconocimiento:** proporciona, al usuario LLC, los medios para intercambiar unidades de datos de servicio de enlace LSDU (*Link Service Data Unit*) sin establecer una conexión de enlace de datos. La transmisión de datos puede ser punto a punto, multidifusión o difusión.
- **Servicio de petición de datos remota sin reconocimiento:** proporciona, al usuario LLC, los medios para solicitar que un nodo remoto transmita sus LDSUs sin establecer una conexión de enlace de datos.

De acuerdo con los tipos de servicios, se definen dos formatos de tramas, de datos LLC y remota LLC. Ambos formatos definen identificadores de 11 bits (estándar) y de 29 bits (extendida).

Las funciones de la subcapa LLC son las siguientes:

- **Filtrar mensajes (*frame acceptance filtering*):** el identificador de una trama no indica la dirección destino pero define el contenido del mensaje, y mediante esta función todo receptor activo en la red determina si el mensaje es relevante o no para sus propósitos.
- **Notificar sobrecarga (*overload notification*):** si las condiciones internas de un receptor requieren un retraso en la transmisión de la siguiente trama de datos o remota, la subcapa LLC transmite una trama de sobrecarga. Una trama de sobrecarga puede ser generada por cualquier módulo que debido a sus condiciones internas no está en condiciones de iniciar la recepción de un nuevo mensaje. De esta forma retrasa el inicio de transmisión de un nuevo mensaje. Un módulo puede generar como máximo 2 tramas de sobrecarga consecutivas para retrasar el mensaje.
- **Proceso de recuperación *(recovery management*):** la subcapa LLC proporciona la capacidad de retransmisión automática de tramas cuando una trama pierde el arbitraje o presenta errores durante su transmisión, dicho servicio se confirma al usuario hasta que la transmisión se completa con éxito.

2.4.4.4.3 Tipos de tramas

Existen distintos tipos de tramas predefinidas por CAN para la gestión de la transferencia de mensajes:

- **Trama de datos:** se utiliza normalmente para poner información en el bus y que la puedan recibir algunos o todos los nodos.
- **Trama de información remota:** puede ser utilizada por un nodo para solicitar la transmisión de una trama de datos con la información asociada a un identificador dado.

El nodo que disponga de la información definida por el identificador la transmitirá en una trama de datos.

- **Trama de error:** se generan cuando algún nodo detecta algún error definido.
- **Trama de sobrecarga:** se generan cuando algún nodo necesita más tiempo para procesar los mensajes recibidos.
- **Espaciado entre tramas:** las tramas de datos (y de interrogación remota) se separan entre sí por una secuencia predefinida que se denomina espaciado inter-trama.
- **Bus en reposo:** En los intervalos de inactividad se mantiene constantemente el nivel recesivo del bus.

2.4.4.4.3.1 Trama de datos

Es la utilizada por un nodo normalmente para poner información en el bus. Puede incluir entre 0 y 8 bytes de información útil. En la Figura 19 se muestra el formato de la trama y a continuación se detallan cada uno de sus campos:

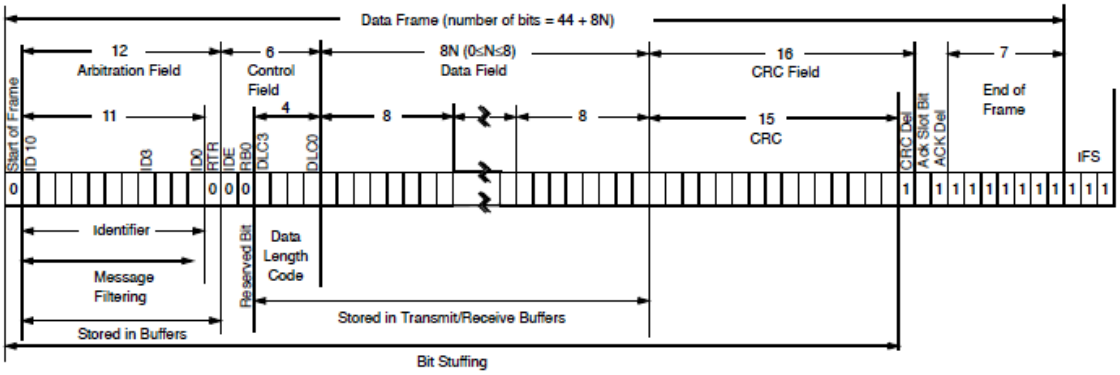


Figura 19: Trama de datos CAN.

- **SOF (*Start Of Frame*)** (1 bit): Determina el inicio de la trama.
- **Campo Arbitraje** (12 a 32 bits):
  - *Identifier* (11 bits): Identificador (único) de los datos que también determina la prioridad del mensaje.
  - *Substitute remote request* (SRR) (1bit): En el caso de tratarse de una trama de CAN 2.0A, este campo actúa como RTR que se explica más adelante. En caso de tratarse de una trama CAN 2.0B, ha de tener un valor recesivo y no realiza ninguna función.
  - *Identifier extension bit* (IDE) (1bit): Determina si el identificador del mensaje es de 11 o de 29 bits. Las tramas de CAN 2.0A contienen mensajes con identificadores de 11 bits, mientras que las de CAN 2.0B usan identificadores de 29 bits.

  - *Identifier extension* (18 bits): Contiene el resto de identificador (único) en el caso de tramas de CAN 2.0B.
  - *Remote transmission request* (RTR) (1 bit): Determina si el mensaje contiene datos o no. Las tramas con datos configuran este campo como dominante, y las que no, como recesivo. Las tramas que no contienen datos se conocen como remotas, tienen menos prioridad que las tramas de datos y sirven para pedir datos a otros nodos del bus.

- **Campo Control** (6 bits):
  - *Reserved bits* (r0, r1) (2 bits): Bits reservados para otras funcionalidades.
  - *Data length code* (DLC) (4 bits): Indica el número de datos que se envía en el campo de datos (en bytes).

- **Campo Datos** (0-64 bits): Contiene entre 0 y 8 bytes (determinado por el campo DLC y RTR) con los datos a transmitir.
- **Campo CRC** (16 bits):
  - *Cyclic redundancy check* (CRC) (15 bits): Código para el control de error de la trama enviada. El receptor utiliza este campo para comprobar que no ha habido errores durante la transmisión.
  - *CRC delimiter* (1 bit): Debe ser un bit recesivo y delimita el final del CRC.

- **Campo ACK** (2 bits):
  - *Slot ACK* (1 bit): El transmisor emite un bit recesivo mientras que los receptores emiten un bit dominante tras leer el CRC para advertir al transmisor que el mensaje ha sido recibido. En caso contrario, el transmisor considerará ha habido un error en su mensaje.
  - *ACK Delimiter* (1bit): Debe ser un bit recesivo y delimita el final del ACK.

- **EOF (*End Of Frame*):** Esta etapa indica el final del mensaje mediante 7 bits recesivos.
- ***Intermission*:** El espacio entre trama y trama ha de ser como mínimo de 3 bits recesivos.

2.4.4.4.3.2 Trama remota

Un nodo envía una trama remota para solicitar información a otro nodo. Para ello configura el campo RTR del paquete CAN como recesivo. Estas tramas no contienen datos como se puede observar en la Figura 20.

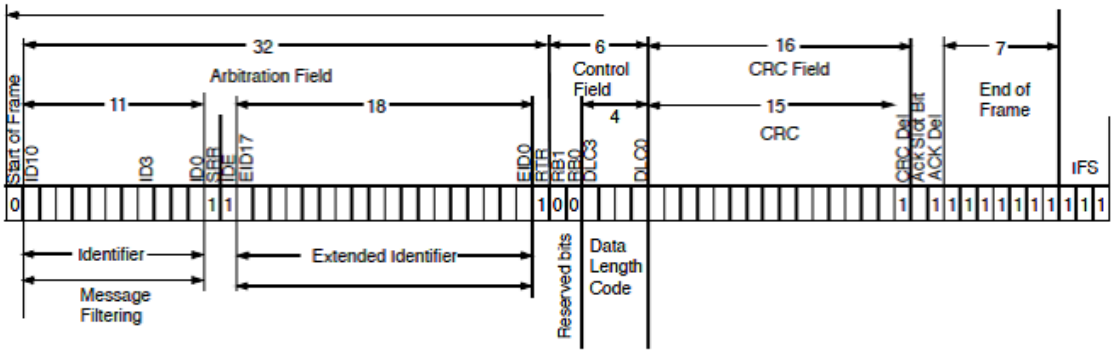


Figura 20: Trama remota CAN.

2.4.4.4.3.3 Trama de error

Se utiliza para advertir de un error en el bus. Los nodos CAN detectan los siguientes tipos de errores:

- ***CRC Error:*** Se calcula un *Cyclic Redundancy Check* (CRC) de 15 bits por el nodo de transmisión y este valor se transmite al bus. Todos los nodos de la red reciben este mensaje, calculan un CRC y verifican que el valor de CRC coincide. Si no coincide, hay un error de CRC y se genera un error de estructura.
- ***Acknowledge Error:*** Indica si al menos un nodo recibió correctamente el mensaje. Si el bit es recesivo entonces ningún nodo lo recibió. Después de que se genera el error de estructura se repite el mensaje original después de un tiempo de intermedio adecuado.
- ***Form Error:*** Si el nodo detecta un error en *End of Frame*, o *Interframe*, Ack, CRC, entonces salta el error de forma. El mensaje se reenvía después de un tiempo prudencial.
- ***Bit Error:*** Sucede cuando un transmisor envía un bit dominante y detecta uno recesivo, o viceversa. En este caso no se genera el *Bit Error* si es algo normal del proceso de arbitraje, pero si se genera el mensaje es reenviado.
- ***Stuff Error:*** Esto significa que el nivel de bit es colocado en el bus en el periodo de tiempo del bit. CAN es además asíncrono, y el *bit stuffing* es usado para permitir al nodo que recibe sincronizar por recuperación del reloj desde la tormenta de datos. Los nodos que reciben se sincronizan a partir de transiciones de recesivas a dominantes. Si hay más de 5 bits de la misma polaridad en una fila, CAN automáticamente produce una polaridad diferente en el bit en la tormenta de datos. El nodo que la recibe la usará para sincronizarse, pero ignorará el bit de *stuff* como dato. Si entre el comienzo del *Start of Frame* y *CRC Delimiter* hay 6 bits consecutivos con la misma polaridad, entonces las reglas del *bit stuffing* se han violado, y ocurre un *Stuff Error*.

Tras detectar un error el nodo actuará de una forma u otra según se encuentre en estado activo o pasivo:

- Si un nodo en estado de error "Activo" detecta un error en el bus interrumpe la comunicación del mensaje en proceso generando un "Indicador de error activo" que consiste en una secuencia de 6 bits dominantes sucesivos. Esta secuencia rompe la regla de relleno de bits y provocará la generación de tramas de error en otros nodos. Por tanto el indicador error puede extenderse entre 6 y 12 bits dominantes sucesivos. Finalmente se recibe el campo de delimitación de error formado por los 8 bits recesivos. Entonces la comunicación se reinicia y el nodo que había sido interrumpido reintenta la transmisión del mensaje.
- Si un nodo en estado de error "Pasivo" detecta un error, el nodo transmite un "Indicador de error pasivo" seguido, de nuevo, por el campo delimitador de error. El indicador de error de tipo pasivo consiste en 6 bits recesivos seguidos y, por tanto, la trama de error para un nodo pasivo es una secuencia de 14 bits recesivos. De aquí se deduce que la transmisión de una trama de error de tipo pasivo no afectará a ningún nodo en la red, excepto cuando el error es detectado por el propio nodo que está transmitiendo. En ese caso los demás nodos detectarán una violación de las reglas de relleno y transmitirán a su vez tramas de error.

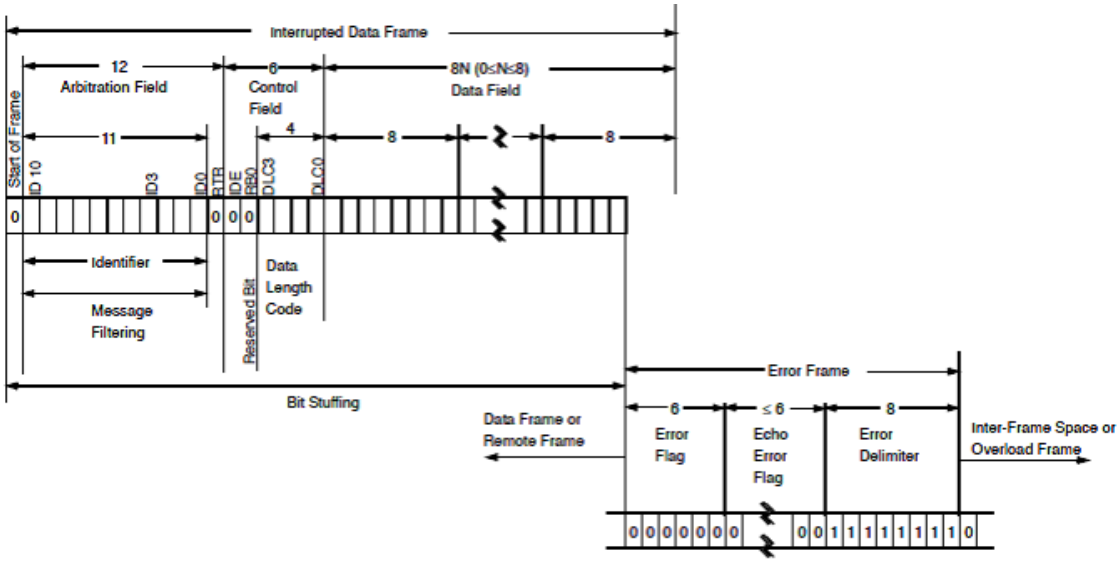


Figura 21: Trama de error CAN.

Tras señalar un error por medio de la trama de error apropiada (véase la Figura 21) cada nodo transmite bits recesivos hasta que recibe un bit también recesivo, luego transmite 7 bits recesivos consecutivos antes de finalizar el tratamiento de error.

2.4.4.4.3.4 Trama de sobrecarga

Las utiliza para forzar a los módulos un tiempo de espera entre tramas. La trama de sobrecarga consta de dos campos como se observa en la Figura 22:

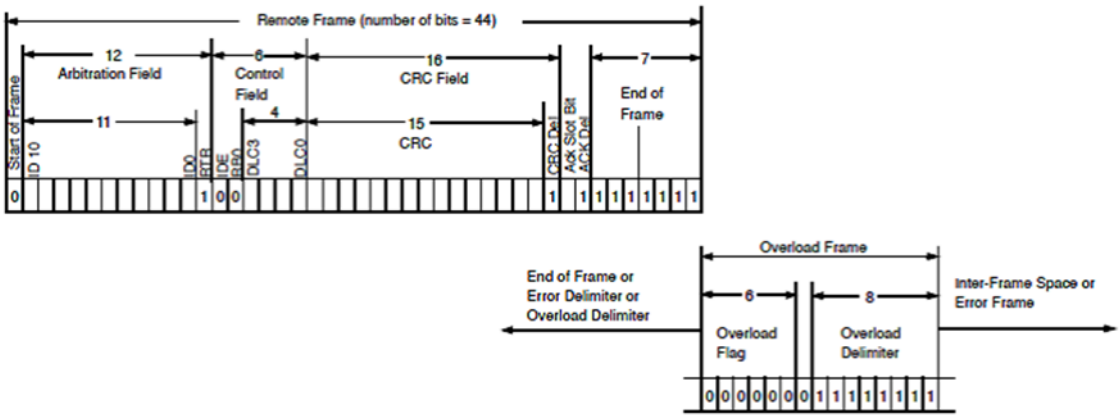


Figura 22: Trama de sobrecarga CAN.

- **El indicador de sobrecarga (*Overload Flag*):** consta de 6 bits dominantes que pueden ser seguidos por los generados por otros nodos, dando lugar a un máximo de 12 bits dominantes.
- **El delimitador de sobrecarga (*Overload Delimiter*):** es de 8 bits recesivos y marca el final de la trama de sobrecarga.

2.4.4.4.3.5 Espacio entre tramas

El espacio entre tramas separa una trama de cualquier tipo de la siguiente trama de datos o interrogación remota. El espacio entre tramas ha de constar de, al menos, 3 bits recesivos. Esta secuencia de bits se denomina *intermission*. Una vez transcurrida esta secuencia, un nodo en estado de error activo puede iniciar una nueva transmisión o el bus permanecerá en reposo (manteniendo constante el nivel recesivo del bus).

Para un nodo en estado error pasivo la situación es diferente, deberá esperar una secuencia adicional de 8 bits recesivos antes de poder iniciar una transmisión. De esta forma se asegura una ventaja en inicio de transmisión a los nodos en estado activo frente a los nodos en estado pasivo.

2.4.4.5 **Niveles superiores**

Aunque el protocolo CAN solo engloba el nivel físico y en el nivel de enlace, existen un gran número de protocolos orientados a facilitar el uso del bus CAN en aplicaciones de mayor complejidad o que necesiten características específicas. Los más conocidos son ISO TP (Transport Protocol) y CAN Open:

- **ISO 15765-2**: comúnmente conocido como ISO-TP, es una norma internacional que define un protocolo de transporte sobre bus de comunicaciones CAN, empleado en automoción. El propósito del protocolo es permitir el transporte de mensajes con una

longitud superior a los 8 bytes de información útil que permite una trama CAN normal. En el modelo OSI, ISO-TP corresponde a los niveles 3 de red y 4 de transporte. Un mensaje ISO-TP segmenta los mensajes largos en varias tramas CAN, completándolas con información que permite su interpretación por parte del receptor. Así, el tamaño máximo de datos útiles (*payload*) que ofrece es de 4095 bytes por TPDU ("tramas ISO-TP").

- **CAN Open:** CAN Open fue originalmente diseñado para la industria de sistemas de control, pero las redes CAN Open también son usadas para aplicaciones de campo como transporte público, equipos médicos, etc. Las especificaciones cubren el nivel de aplicación, el perfil de la comunicación, el armazón de los aparatos programables de los nodos y recomendaciones de cables y conectores. Es uno de los HLP (*Higher Layer Protocols*) más utilizados en las aplicaciones basadas en el bus CAN.

3 Hardware usado

El próximo capítulo trata sobre los dispositivos que conforman el sistema desarrollado y los usados para llevar a cabo su configuración. En este apartado se detallará qué equipos han sido escogidos para desarrollar este proyecto, sus características, el escenario montado y la forma de uso. La elección de los dispositivos del sistema se basó en buscar un equipo que pudiese simular una ECU OBD-II, que fuera compatible con CAN y que no fuera excesivamente caro. Otro componente necesario era la interfaz que nos permitiera conectar dicha ECU a cualquier PC comercial. Con todo esto se pretende simular un sistema OBD-II de forma sencilla y sin necesidad de utilizar un hardware complejo, y que permita su uso para posteriores proyectos de diseño e implementación de ordenadores de a bordo en el laboratorio, sin tener que instalarlos directamente en un vehículo.

3.1 Dispositivos hardware usados

Dentro del montaje usado en el laboratorio se puede distinguir entre los dispositivos que formarían parte del simulador de la ECU, los dispositivos utilizados para la visualización de los resultados y los dispositivos usados como interfaz para conectar la ECU con estos últimos.

Los dispositivos que se encargarán de simular una ECU serán, como ya se mencionó en el capítulo introductorio, una placa Arduino UNO junto con una placa CAN-BUS Shield. Esta última está dotada de un controlador CAN con interfaz SPI (para comunicarse con Arduino UNO) y de un transceiver CAN, dotando así a la placa Arduino UNO de capacidad para enviar tramas CAN.

En cuanto al dispositivo utilizado para visualizar los resultados, se ha utilizado un ordenador portátil. De hecho, se podría utilizar cualquier dispositivo compatible con un software de diagnóstico OBD-II y que tenga conectividad RS-232, USB o disponga de una tarjeta Bluetooth. Es decir podríamos utilizar desde un ordenador personal, hasta Tablets-PC, Tablets o Smartphones.

Esto nos lleva a seleccionar cuidadosamente el siguiente dispositivo utilizado como interfaz, dependiendo de la conectividad necesaria para comunicarse con el dispositivo encargado de visualizar los datos. En este caso, al usar un ordenador portátil como equipo de visualización, se ha optado por una interfaz USB OBD-II. Dentro de este tipo de interfaces destaca por su precio y popularidad la interfaz ELM 327.

3.1.1 Arduino UNO

Arduino es una plataforma de prototipos electrónicos de código abierto (*open-source*) basada en hardware y software flexibles y fáciles de usar. Las placas se pueden ensamblar a mano o comprarlas preensambladas, el software se puede descargar gratuitamente de su sitio web. Los diseños de referencia del hardware están disponibles bajo licencia *open-source*, por lo que el usuario es libre de adaptarla a sus necesidades.

La plataforma Arduino está basada en una sencilla placa con entradas y salidas (E/S) analógicas y digitales. El elemento principal de dicha placa es el microcontrolador ATmegaxx8 (el microcontrolador puede cambiar de un modelo a otro), un chip sencillo y de bajo coste que permite el desarrollo de múltiples diseños. La plataforma Arduino puede captar señales del entorno, mediante la recepción de entradas desde una variedad de sensores, y puede utilizarse para controlar luces, motores u otros sistemas.

3.1.1.1 Justificación

¿Por qué optar por Arduino y no por otro tipo de plataformas?

Existen muchos otros microcontroladores y plataformas microcontroladoras disponibles en el mercado: Parallaz Basic Stamp, Phidgets o Netmedia's BX-24, son algunas de las muchas ofertas de funcionalidad similar a Arduino. Al igual que otras plataformas, Arduino también simplifica el proceso de trabajo con microcontroladores, pero ofrece una serie de ventajas que la diferencia del resto:

- **Barato:** Las placas Arduino son relativamente baratas comparadas con otras plataformas microcontroladoras. Aunque el precio de la placa preensamblada ya es bastante bajo, puedes ensamblarla tú mismo abaratando aún más su coste.
- **Multiplataforma:** El software para programar placas Arduino se ejecuta en sistemas operativos Windows, Macintosh OS X y GNU/Linux. Mientras la mayoría del software de programación de otras plataformas de microcontroladores se ejecutan principalmente en Windows.
- **Entorno de programación claro y simple:** basado en el entorno de programación y el lenguaje denominado *Processing*.
- **Código abierto y software extensible:** El software Arduino está publicado como herramientas de código abierto, disponible para extensión por programadores experimentados. El lenguaje puede ser expandido mediante librerías, y aquellos que quieran profundizar en los detalles técnicos pueden dar el salto desde la programación típica de Arduino a la programación en lenguaje AVR C en el cual está basado.
- **Código abierto y hardware extensible:** El hardware está basado en microcontroladores ATmega de la marca Atmel. Los planos para los módulos están publicados bajo licencia *Creative Commons*, por lo que cada uno puede diseñar su propia versión del módulo.

3.1.1.2 **Hardware de la placa Arduino UNO**

Arduino UNO es una placa electrónica basada en el ATmega328P. Cuenta con 14 pines digitales de entrada/salida (de los cuales 6 se pueden utilizar como salidas PWM), 6 entradas analógicas, un cristal de cuarzo de 16 MHz, una conexión USB, un conector de alimentación, una cabecera ICSP y un botón de reinicio (véase la Figura 23).

El nombre "UNO" fue elegido para conmemorar el lanzamiento del entorno de programación integrado Arduino (IDE) 1.0. UNO es el primero de una serie de placas Arduino USB y el modelo de referencia para la plataforma Arduino.

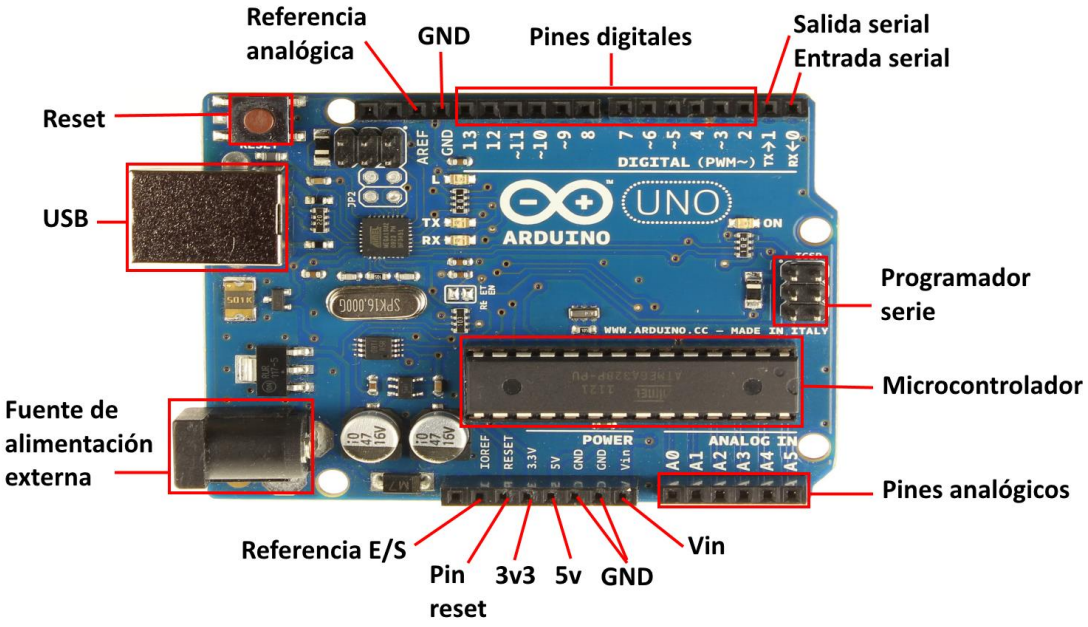


Figura 23: Arduino UNO Rev 3.

Las especificaciones técnicas de la placa Arduino UNO son las mostradas en la Tabla 13:

Microcontrolador	ATmega328P
Voltaje de operación	5V
Voltaje de entrada (recomendado)	7-12V
Voltaje de entrada (límite)	6-20V
Pines de E/S digital	14 (6 de ellos utilizables como salida PWM)
Pines de E/S PWM	6
Pines de entrada analógica	6
Corriente DC para los pines de E/S	20 mA
Corriente DC para los pines 3.3V	50 mA
Memoria Flash	32 KB (0,5 KB para el *bootloader*)
SRAM	2 KB
EEPROM	1 KB
Frecuencia de reloj	16 MHz
Longitud	68.6 mm
Ancho	53.5 mm
Peso	25 g

Tabla 13: Especificaciones Técnicas Arduino UNO.

En los siguientes apartados se describe el estudio que se llevó a cabo con el microcontrolador y sus pines.

3.1.1.2.1 ATmega328P

El microprocesador ATmega328P de la marca Atmel, trabaja con una memoria *flash* de 32 KB (0.5 KB son usados para el *bootloader o cargador de arranque*). Además, dispone de 2KB de memoria SRAM (*Static Random Access Memory*) y 1 KB de memoria EEPROM (*Electrically Erasable Programmable Read-Only Memory*).

En la Figura 24 se muestra el mapa de pines entre los pines Arduino y los puertos de la ATmega328P. Los microprocesadores ATmega8, ATmega168 y ATmega328 comparten el mismo mapa de pines.

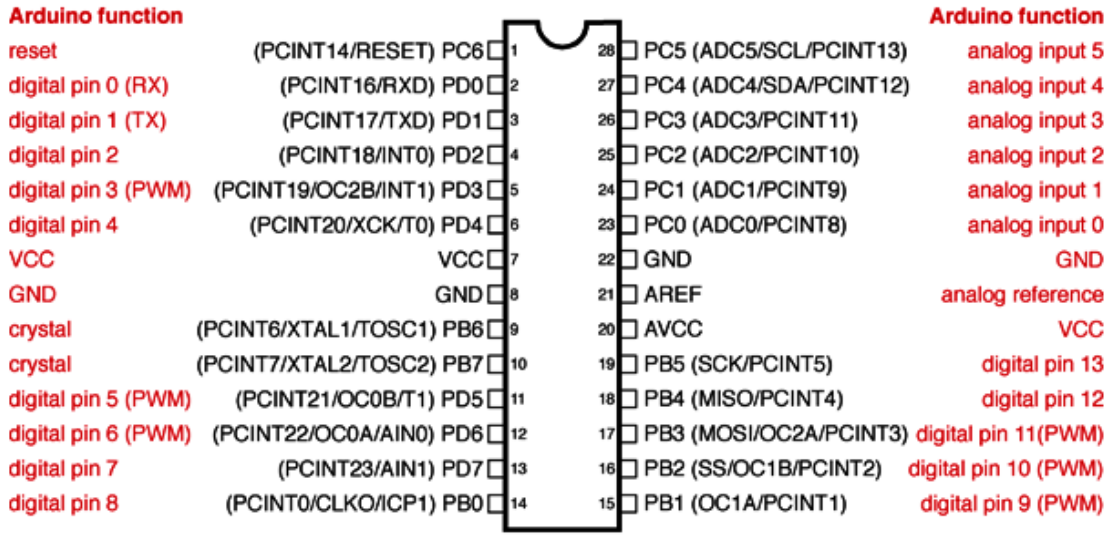


Figura 24: Mapa de pines de ATmega328P.

3.1.1.2.2 Pines de alimentación

La placa puede ser alimentada mediante la conexión USB o mediante una fuente de alimentación externa (recomendada de 7-12V). Se tienen unas salidas de tensión continua debido a unos reguladores de tensión y condensadores de estabilización. Estos pines son:

- **VIN:** se trata de una fuente de tensión de entrada que contendrá la tensión a la que estamos alimentando al Arduino mediante la fuente externa.
- **5V:** fuente de tensión regulada de 5V, esta tensión puede venir ya sea del pin VIN a través de un regulador interno, o se suministra a través de USB o de otra fuente de 5V regulada.
- **3.3V:** fuente de 3.3 voltios generados por el regulador interno con un consumo máximo de corriente de 50mA.
- **GND:** pines de tierra.
- **IOREF:** voltaje de referencia con el que el microcontrolador opera.

3.1.1.2.3 Pines de entrada y salida

Cada uno de los 14 pines digitales de la placa Arduino UNO pueden ser usados como entrada o salida, usando las funciones *pinMode ( ), digitalWrite ( ) y digitalRead ( )*. Estos pines operan a 5 voltios, cada pin puede proporcionar o recibir un máximo de 20 mA y tiene una resistencia interna *pull-up* (desconectada por defecto) de 20-50 KOhms. Ningún puerto de E/S deberá exceder el valor máximo de 40 mA para evitar dañar permanentemente al microcontrolador.

Además algunos de estos pines tienen funciones especiales:

- **Serial:** 0 (Rx) y 1 (Tx). Usados para recibir (Rx) y transmitir (Tx) datos TTL (*transistor-transistor logic*) en serie. Estos pines están conectados a los correspondientes pines USB-a-TTL serie de la ATmega8U2.
- **Interrupciones externas:** pines 2 y 3. Estos pines pueden ser configurados para disparar una interrupción.
- **PWM:** pines 3, 5, 6, 9, 10 y 11. Proporcionan salida PWM de 8 bits con la función *analogWrite ( )*.
- **SPI:** pines 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). Estos pines soportan la comunicación SPI haciendo uso de la librería SPY.
- **LED:** pin 13. Hay un LED empotrado conectado al pin digital 13. Cuando el pin está a valor HIGH, el LED esta encendido, cuando el pin está a LOW, está apagado.

Arduino UNO dispone de 6 entradas analógicas, cada una de las cuales proporciona 10 bits de resolución (por ejemplo 1024 valores diferentes). Por defecto miden 5 voltios desde tierra, aunque es posible cambiar el valor más alto de su rango usando el pin AREF y la función *analogReference ( )*. Algunos de estos pines tienen funcionalidad especializada:

- **TWI:** los pines A4 (SDA) y A5 (SCL), soportan la comunicación TWI (*Two Wired Serial*), también conocida como I2C.
- **AREF:** Voltaje de referencia para las entradas analógicas.
- **Reset:** resetea el microcontrolador si este pin está a LOW. Normalmente usado para dotar a la placa de un botón de reset adicional.

3.1.2 Hardware de la placa CAN-BUS Shield

Como se ha estudiado en apartados anteriores, CAN es uno de los protocolos de comunicación bus más usados debido a su largo alcance, su velocidad de comunicación y su alta fiabilidad. Se encuentra comúnmente en máquinas de control y en el bus de diagnóstico automotriz. La placa CAN-BUS Shield dota de conectividad CAN a la placa Arduino, para ello

la placa CAN-BUS Shield cuenta con el controlador CAN MCP2515 con interfaz SPI y con un *transceiver* CAN MCP2551. Las características principales de la interfaz CAN-BUS Shield son las siguientes:

- Implementa CAN 2.0B a velocidades de hasta 1 Mbps.
- Interfaz SPI de hasta 10 MHz.
- Soporta tramas estándar (11 bits), extendidas (29 bits) y tramas remotas.
- Dos buffers de recepción para el almacenamiento de mensajes con prioridad.
- Conector industrial estándar de 9 pines sub-D (véase la Figura 25).

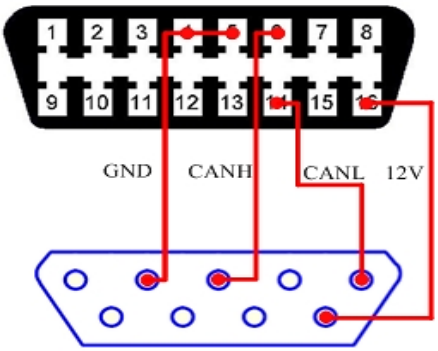


Figura 25: Conector sub-D.

- Dos indicadores LED.
- Voltaje de operación: 5V.
- Dimensiones: 68x53 mm.
- Peso: 50 g.

Como se observa en la Figura 26, la placa CAN-BUS Shield está diseñada para ser "pinchada" directamente sobre una placa Arduino UNO.

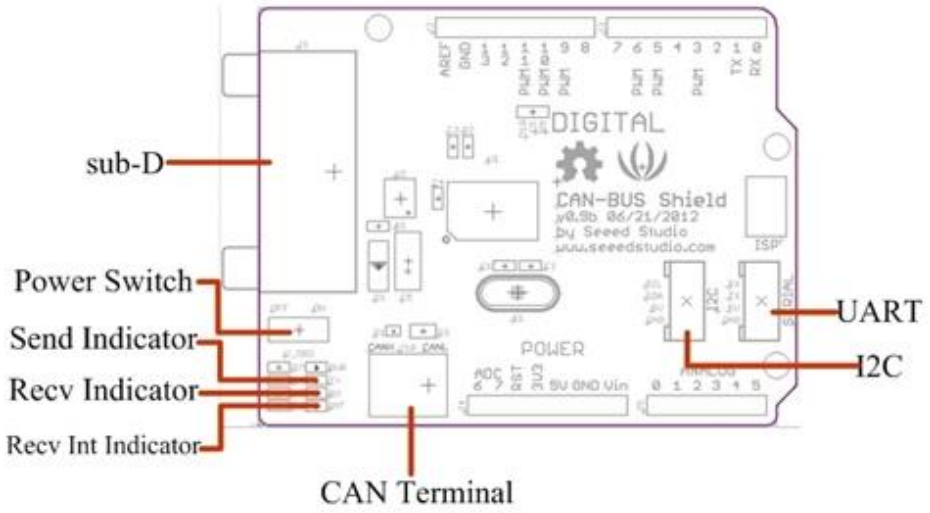


Figura 26: CAN-BUS Shield.

3.1.2.1 **Controlador MCP2515**

El MCP2515 es un controlador CAN de segunda generación. Respecto de la primera generación (MCP2510) incluye características mejoradas como un mejor rendimiento, filtrado y soporte para protocolos *time-triggered* (TTP). Las principales características del MCP2515 son las siguientes:

- Implementa CAN 2.0B de hasta 1Mbps.
- Encapsulado de 18 pines (véase la Figura 27).
- Interfaz SPI de alta velocidad (10MHz).
- El modo *One-shot* (disparo único) asegura que se intente la transmisión de mensajes solo una vez.
- Filtrado *Databyte*.
- Uso de indicadores de inicio de trama (SOF).
- Tecnología CMOS de baja potencia.

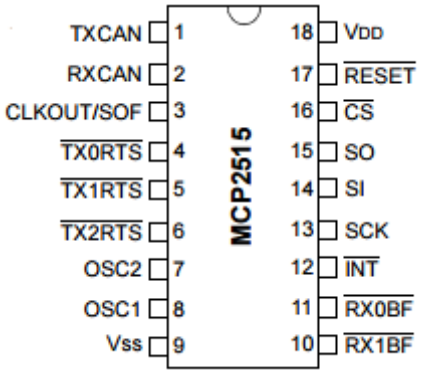


Figura 27: Encapsulado del MCP2515.

3.1.2.2 **Transceiver MCP2551**

El MCP2551 es un *transceiver* CAN de alta velocidad, dispositivo tolerante a fallos que actúa de interfaz entre el controlador CAN y el bus físico. El MCP2551 proporciona señales diferenciales para dar capacidad de transmisión y recepción al controlador CAN. Las principales características del MCP2551 son las siguientes:

- Entrada de control de pendientes (*slope control*).
- Velocidad de hasta 1 Mbps.
- Implementa los requisitos físicos del estándar ISO-11898.
- Adecuado para aplicaciones industriales y de automoción (12V-24V).
- Detección permanente del dominante en la entrada.
- Modo de espera (*standby*).
- Alta inmunidad al ruido.

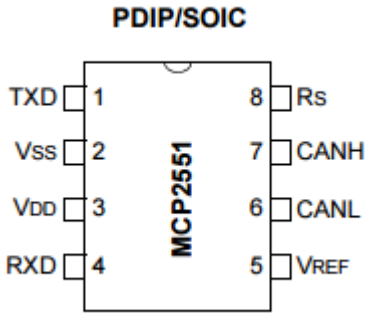


Figura 28: Encapsulado del MCP2551.

3.1.3 **Interfaz OBD-II ELM327**

Actualmente todos los vehículos incorporan una interfaz desde la cual un equipo de prueba pueda obtener información de diagnóstico, desafortunadamente estos equipos manejan estándares de comunicación que, en la mayoría de los casos, no son directamente compatibles con PCs, tablets o móviles.

Por este motivo fue diseñado el ELM327 por la empresa ELM Electronics, para actuar como puente entre el estándar OBD-II y el estándar RS-232. A partir de este intérprete se han comercializado distintas interfaces utilizando este dispositivo como núcleo, de entre estas las más comunes son las interfaces RS-232, USB (mostrada en Figura 29 y que es la que se ha utilizado en este proyecto) y Bluetooth.

Figura 29: Interfaz ELM327 USB.

El intérprete ELM327 reconoce automáticamente los 9 tipos de protocolos de OBD-II definidos en el estándar:

- SAE J1850 PWM
- SAE J1850 VPW
- ISO 9141-2
- ISO 14230-4 KWP (5 *baud init*)
- ISO 14230-4 KWP (*fast init*)
- ISO 15765-4 CAN (11 bit ID, 500 kbaud)
- ISO 15765-4 CAN (29 bit ID, 500 kbaud)
- ISO 15765-4 CAN (11 bit ID, 250 kbaud)
- ISO 15765-4 CAN (29 bit ID, 250 kbaud)
- SAE J1939 CAN (29 bit ID, 250 kbaud)

Además se pueden configurar algunos parámetros adicionales mediante los comandos AT. El protocolo de comandos que usa el ELM327 es uno de los estándares para interfaces PC-a-OBD más populares, tanto que otros vendedores también implementan dicho protocolo.

En las Figuras 30 y 31 se ilustran el diagrama de pines y el diagrama de bloques del ELM327 respectivamente.

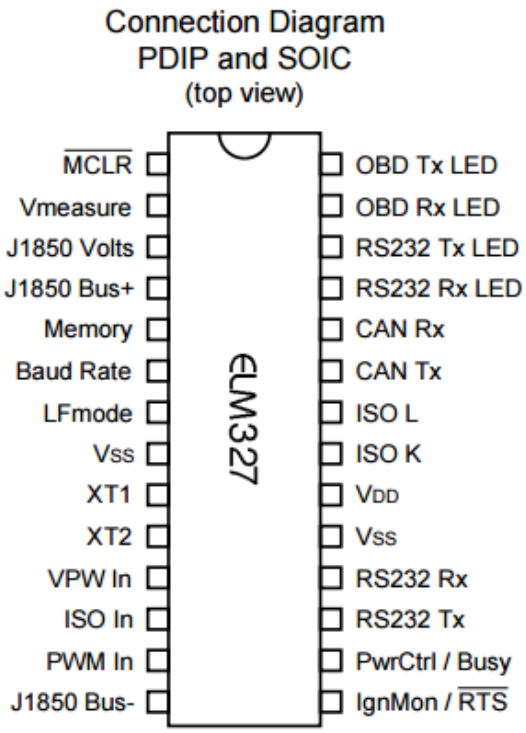


Figura 30: Diagrama de pines del ELM327.

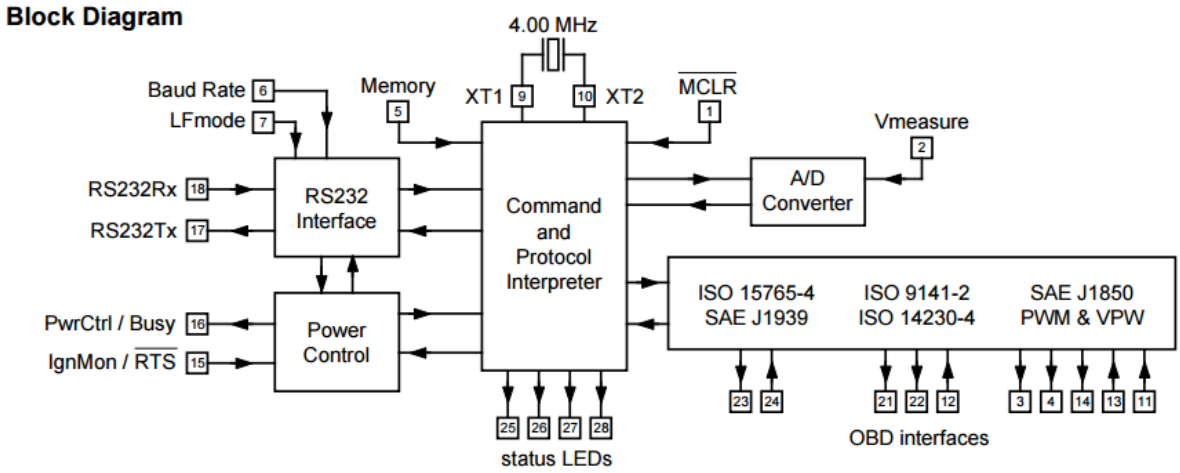


Figura 31: Diagrama de bloques del ELM327.

3.1.3.1 **Comandos AT**

En el intérprete ELM327 se pueden configurar algunos parámetros para modificar su comportamiento. La mayoría de los parámetros son autoconfigurables ya que el intérprete verifica el protocolo soportado por el módulo OBD-II y ajusta automáticamente la velocidad de transferencia y algunos parámetros para la transmisión de datos, aunque se pueden personalizar algunas configuraciones, por ejemplo para ajustar los tiempos de espera, habilitar o deshabilitar el "*echo*", resetear o reiniciar al intérprete, entre otros. Todas estas configuraciones se realizan a través de los comandos AT.

El intérprete ELM327, al igual que los *modems*, reconoce a los comandos que inicien con las letras "AT" como un comando de configuración interno, cuando ejecuta correctamente el comando devuelve la respuesta "OK" o el valor solicitado. Se debe tener en cuenta que los valores numéricos deben ser enviados o recibidos en hexadecimal. En la Tabla 14 se muestran algunos comandos.

Comando	Descripción
AT CEA	Deshabilita el direccionamiento CAN extendido.
AT CF hhh	Inserta un filtro ID en hhh
AT CP hh	Inserta prioridad CAN en hh
AT CRA	Resetea los filtros
AT RTR	Envía un mensaje RTR

Tabla 14: Ejemplo de comandos AT.

3.1.4 **Otros dispositivos**

Hasta ahora se han descrito los dispositivos que forman parte del simulador de la ECU que ha sido desarrollado (Arduino UNO y CAN-BUS Shield) y la interfaz que usaremos para conectar está a un PC.

En este apartado analizaremos el resto de elementos que son necesarios para el montaje del sistema final.

3.1.4.1 **Equipo de visualización**

El equipo de visualización podría variar desde un ordenador personal hasta una tablet. Los principales requisitos a cumplimentar por este dispositivo son:

- Disponer de puertos USB conectar la interfaz ELM327.
- Sistema operativo compatible con los drivers de la interfaz ELM327.
- Existencia de software gratuito para la visualización del diagnóstico OBD-II (software de diagnóstico o escáner OBD-II).

En primer lugar se barajó la posibilidad de usar una Tablet-PC disponible, en concreto: Ikar PC-W08A. Esta opción se descartó debido a la dificultad de encontrar de drivers de la interfaz ELM327 para su sistema operativo, Windows XP Embedded, y a la escasez de software de diagnóstico OBD-II compatible.

Por estos motivos finalmente se optó por usar un ordenador personal con Windows 8, que además de usarse como equipo para visualizar los resultados del diagnóstico, también se utiliza

para instalar el entorno de programación de Arduino y programar el software necesario del simulador de la ECU.

3.1.4.2 **Placa de simulación de sensores del vehículo**

Con el objetivo de simular señales de importancia para la ECU a la hora de realizar un diagnóstico OBD-II, se ha diseñado una placa de simulación de sensores del vehículo (Figura 32).

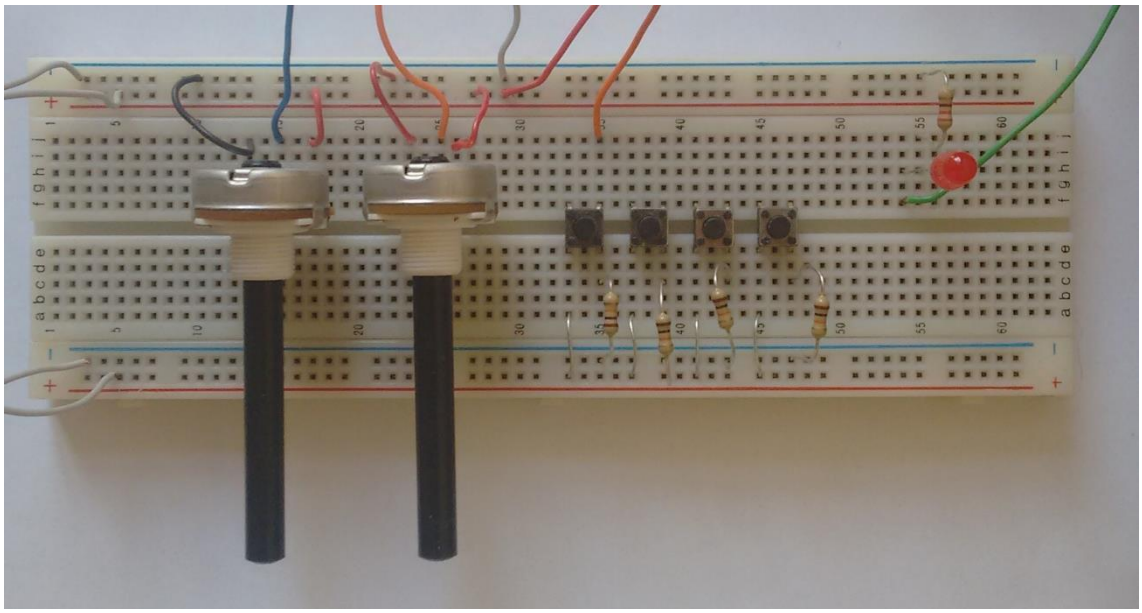


Figura 32: Placa para la simulación de señales de los sensores del vehículo.

La placa consiste en una protoboard con los siguientes dispositivos:

- Dos potenciómetros para simular señales analógicas como puede ser la posición del pedal del acelerador o el nivel de batería.
- Pulsadores que simulan señales de error (por limitaciones de la placa Arduino solo se ha usado uno de ellos).
- Un led para simular el indicador luminoso MIL.
- Cableado para su conexión y alimentación.

3.1.4.3 **Cableado**

Además de los cables utilizados para alimentar y conectar el simulador de la ECU (Arduino UNO y CAN-BUS Shield) con la placa de simulación de sensores, se ha utilizado el siguiente cableado:

- Cable USB tipo A/B: usado para la alimentación y programación de la placa Arduino UNO, que a su vez alimenta al CAN-BUS Shield y a la placa de simulación de sensores.

Figura 33: Cable USB tipo A/B.

- Cable con terminales cocodrilo: utilizados para conectar las salidas CAN-H y CAN-L de la placa CAN-BUS Shield con los correspondientes terminales de la interfaz ELM327.

3.2 **Montaje de los dispositivos**

El entorno de simulación consiste en un dispositivo simulador de una ECU formada por una placa Arduino UNO sobre la que se conecta la placa CAN-BUS Shield, dotando así a la placa Arduino de capacidad para enviar y recibir tramas CAN.

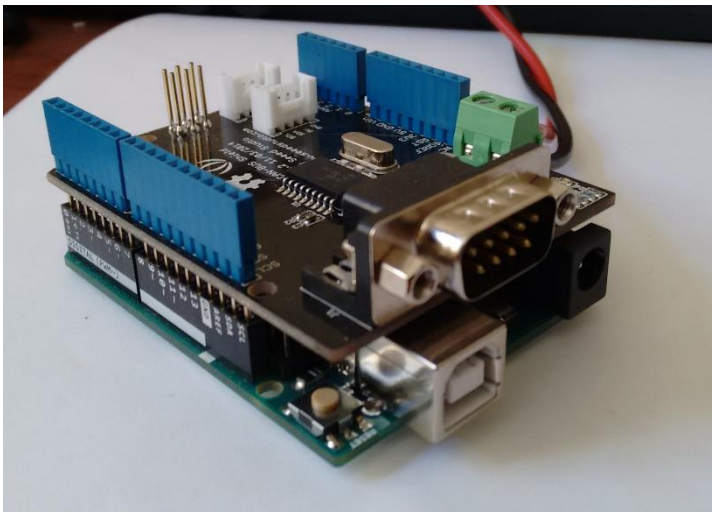


Figura 34: CAN-BUS Shield conectada a Arduino UNO.

Para completar el simulador de la ECU desarrollado se añade la placa de simulación con el fin de generar señales reales que simulen a las generadas por los sensores de un vehículo real.

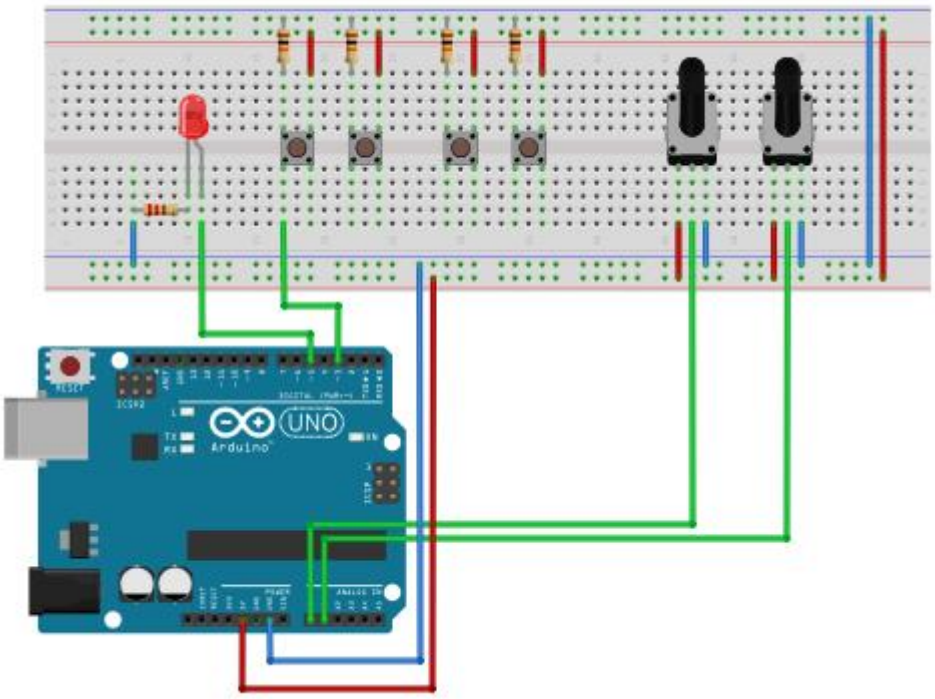


Figura 35: Conexión Arduino-protoboard.

El dispositivo simulador de la ECU se alimenta y programa mediante un cable USB tipo A/B, con el extremo USB tipo A conectado en el PC y el extremo tipo B en el conector USB de la placa Arduino UNO. Para comenzar la simulación una vez que tengamos cargado el programa en el simulador de la ECU, se conectan las salidas CAN-H y CAN-L a los pines 6 y 14 del conector OBD-II de la interfaz ELM327. Una vez conectado solo tenemos que conectar el extremo USB de la interfaz ELM327 a uno de los puestos USB del PC, tras ello el sistema estará listo para que se inicie el diagnóstico OBD-II a falta de iniciar la conexión vía software.

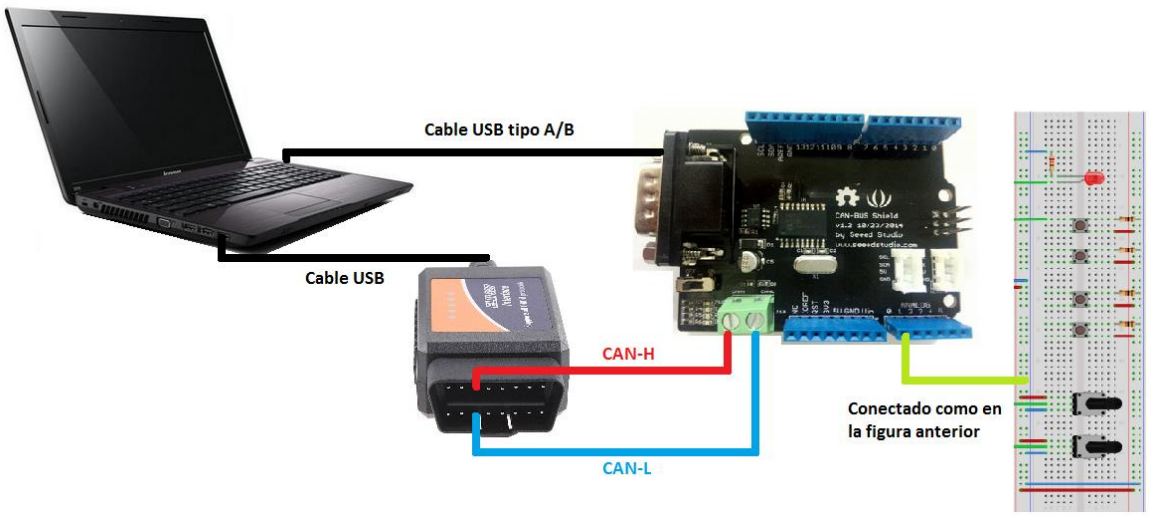


Figura 36: Montaje final para la simulación.

3.3 **Consideraciones sobre el hardware usado**

Con anterior montaje se consigue un simulador de una ECU barato respecto a los simuladores existentes en el mercado. Entre ellos destaca el simulador *Engine Simulator I* del fabricante *AutoSim* (Figura 37).

Figura 37: Engine Simulator I.

El precio del montaje presentado en este proyecto ronda los 70€, mientras que el precio de mercado del simulador Egine Simulator I ronda los 1.500€. No obstante el montaje realizado presenta una serie de inconvenientes respecto a este tipo de simuladores profesionales, ligados principalmente a las limitaciones hardware.

El trabajo con una placa como la Arduino UNO conlleva una serie de limitaciones derivadas de estar diseñada para aplicaciones generales. En un sistema de un vehículo real, una ECU está conectada vía CAN con otros módulos que concentran todo tipo de sensores, de tal forma que no vayan a parar todas las señales analógicas a la ECU. El número de E/S unido a la escasez de presupuesto han limitado el número de variables OBD-II (PIDs) que se han podido monitorizar en tiempo real.

Otro inconveniente de Arduino UNO es el número de interrupciones hardware, que está limitado a 2, una de ellas es usada por la placa CAN-BUS Shield para la comunicación, por lo que solo se ha dispuesto de una interrupción para simular una señal de error proveniente de algún tipo de sensores simulados del vehículo.

4 Software empleado

Para que la ECU sea capaz de interpretar las peticiones OBD-II, de leer la información física de sus puertos de entrada y contestar a dicha petición, es necesario dotarla de un software que le dé dicha funcionalidad. Dicho software deberá ser programado en el microcontrolador de la placa Arduino UNO haciendo uso de un entorno de programación adecuado. Otro elemento software necesario para simular el funcionamiento de la ECU será el software de diagnóstico OBD-II, que se encargará de realizar las peticiones y mostrar los resultados en gráficas de cara al usuario. El presente capítulo se centra en el estudio de todo el software necesario para poner en marcha el simulador de la ECU y también para la visualización de los parámetros del vehículo simulados. En primer lugar se estudiará el entorno de desarrollo de Arduino (IDE), instalación y conceptos básicos a la hora de programar una placa Arduino. Acto seguido se explica con detalle el programa desarrollado para la simulación de la ECU en la placa Arduino UNO, y de las librerías facilitadas para el uso del CAN-BUS Shield. Por último se detallan las principales características del software de diagnóstico OBD-II seleccionado para la simulación.

4.1 Entorno de programación Arduino

Arduino necesita de un programa externo ejecutado en otro ordenador para poder escribir programas para la placa Arduino. Éste software es lo que llamamos Arduino IDE (*Integrated Development Environment*) o Entorno de Desarrollo Integrado en español. Arduino IDE es muy sencillo y basado en Processing. Para usarlo el procedimiento es el siguiente: se escribe un programa en el IDE, se carga en Arduino, y el programa se ejecutará automáticamente en la placa. Processing es un lenguaje de programación y entorno de desarrollo integrado de código abierto basado en Java, que sirve como medio para la enseñanza y producción de proyectos multimedia e interactivos de diseño digital.

4.1.1 Instalación del software Arduino

En la sección actual se enumeran los pasos a seguir para instalar Arduino IDE y cargar un primer programa de ejemplo en una placa Arduino UNO.

1) **Descargar el entorno Arduino:** Se puede descargar el entorno de desarrollo Arduino de forma gratuita en la sección de descargas de la web oficial de Arduino. El entorno de desarrollo Arduino se encuentra disponible para plataformas Windows, Mac OS X y Linux. Una vez finalizada la descarga, se descomprime el archivo descargado para obtener el directorio que contiene dicho entorno de desarrollo.

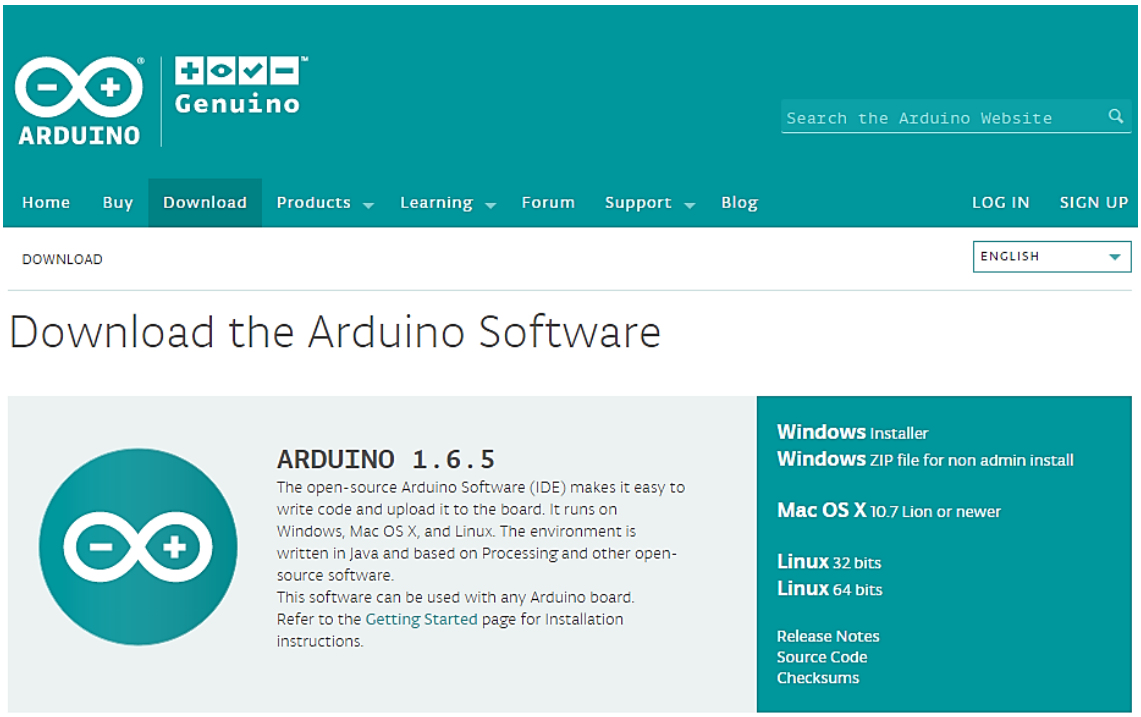


Figura 38: Web oficial de Arduino.

2) **Instalar *drivers* USB:** se conecta la placa Arduino al PC mediante un cable USB de tipo A/B; en este punto se pueden dar dos situaciones: el PC encuentra e instala los *drivers* automáticamente, o no se encuentran los *drivers* y avisa del error. En el segundo caso se deben instalar los *drivers* de forma manual. Desde el panel de control se accede al administrador de dispositivos, una vez allí, se actualizará el software del controlador de forma manual, seleccionando la carpeta *drivers* del directorio anteriormente descargado. Tras ello se tendrá la placa Arduino en uno de los puertos COM del PC.
3) **Ejecutar el entorno Arduino:** el entorno desarrollo de Arduino tiene una apariencia como la mostrada en la Figura 38, en ella también se pueden ver las utilidades de los iconos de su barra de herramientas.

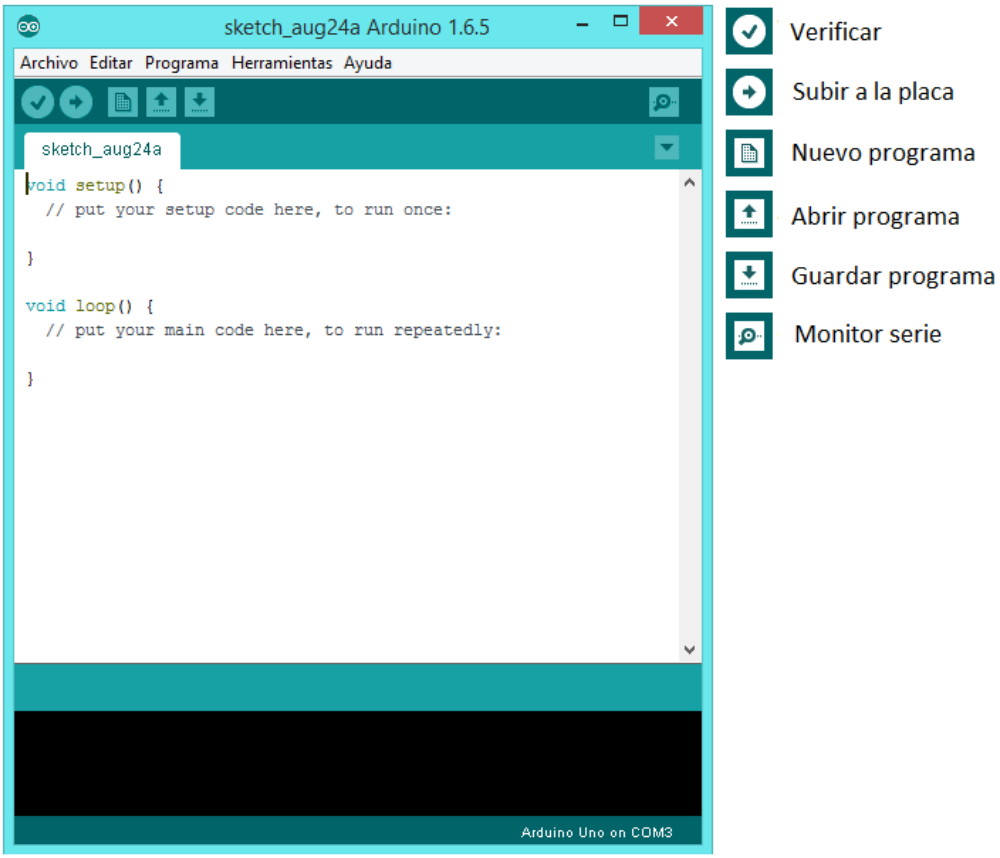


Figura 39: Entorno de desarrollo Arduino.

4) **Configurar comunicaciones entre Arduino UNO y el PC:** abrir el menú Herramientas del entorno de desarrollo, seleccionar en la opción Puerto el puerto COM donde se encuentre conectada la placa y en la opción Placa seleccionar el tipo de placa Arduino con la que se esté trabajando.
5) **Cargar un programa:** los pasos a seguir para cargar un programa son simples, primero se abre o escribe el programa que deseamos cargar, a continuación se verifica el código para comprobar que no tiene errores, por último se carga el programa en la placa. Todas

estas acciones pueden realizarse haciendo uso de los iconos correspondientes de la barra de herramientas.

4.1.2 Lenguaje de programación Arduino

La plataforma Arduino se programa mediante el uso de un lenguaje propio basado en el lenguaje de programación de alto nivel Processing, y utiliza librerías escritas en los lenguajes C/C++. En las siguientes secciones se estudiaran los conceptos básicos del lenguaje de programación Arduino.

4.1.2.1 Estructura básica

La estructura básica del lenguaje de programación de Arduino es bastante simple y se compone de al menos dos partes. Estas dos partes necesarias, o funciones, encierran bloques que contienen declaraciones, estamentos o instrucciones. Estas dos funciones son necesarias para el funcionamiento de cualquier programa Arduino:

- **Función *setup*:** es la parte encargada de recoger la configuración, es la primera función en ser ejecutada dentro del programa y se ejecuta una única vez. Esta función se utiliza principalmente para inicializar los modos de trabajo de los pines E/S y para configurar la comunicación serie, aunque tiene muchas utilidades adicionales.
- **Función *loop*:** también conocida como función bucle contiene el código que se ejecutará continuamente (lectura de entradas, activación de salidas, etc.). Esta función forma el núcleo de todos los programas de Arduino y la que realiza la mayor parte del trabajo.

Adicionalmente se pueden añadir otras funciones. Una función es un bloque de código que tiene un nombre y un grupo de declaraciones que se ejecutan cuando se llama a la función. Las funciones se escriben para ejecutar tareas repetitivas y reducir el desorden en un programa. Las funciones se declaran asociadas a un tipo de dato (*Type*) que será el que devolverá la función. Después de declarar el tipo de dato que devuelve la función se debe escribir el nombre de la función y entre paréntesis se escribirán, si es necesario, los parámetros que se deben pasar a la función para que se ejecute.

4.1.2.2 Variables y tipos de datos

Una variable es una manera de nombrar y almacenar un valor numérico para su uso posterior por el programa. Como su nombre indica, las variables son números que se pueden

variar a lo largo de la ejecución del programa, para ello deben ser declaradas definiendo su tipo y nombre y, opcionalmente, asignarles un valor.

Las variables pueden ser de los siguientes tipos:

- **Byte:** Byte almacena un valor numérico de 8 bits sin decimales. Tienen un rango entre 0 y 255.
- ***Int:*** el formato entero (*int*) es un tipo de datos primarios que almacenan valores numéricos de 16 bits sin decimales comprendidos en el rango -32.767 a 32.768.
- ***Long:*** el formato de variable numérica de tipo extendido (*long*) se refiere a números enteros de 32 bits sin decimales que se encuentran dentro del rango -2147483648 a 2147483647.
- ***Float:*** el formato de dato del tipo punto flotante (*float*) se aplica a los números con decimales con una resolución de 32 bits.
- ***Arrays:*** Un *array* es un conjunto de valores a los que se accede mediante un índice numérico. Cualquier valor puede ser recogido haciendo uso del nombre de la matriz y el número del índice.

4.1.2.3 **Constantes**

El lenguaje de programación de Arduino tiene unos valores predeterminados, que son llamados constantes. Se utilizan para hacer los programas más fáciles de leer. Las constantes se clasifican en los siguientes grupos:

- ***TRUE/FALSE*:** son constantes booleanas que definen niveles lógicos. *FALSE* se define como cero y *TRUE* como cualquier valor distinto de cero.
- ***HIGH/LOW*:** son constantes que definen los niveles de los pines digitales. *HIGH* se define como nivel 1 lógico, encendido o 5V, mientras que *LOW* se define como como nivel 0 lógico, apagado o 0V.
- ***INPUT/OUTPUT*:** se utilizan para definir un pin digital como entrada (*input*) o salida (*output*).

4.1.2.4 **Aritmética**

Los operadores aritméticos que incluye Arduino son: suma (+), resta (-), multiplicación (*) y división (/). Además de las operaciones aritméticas básicas Arduino permite hacer las siguientes operaciones:

- **Asignaciones compuestas:** combinan una operación aritmética con una asignación de variable.

x ++	x = x + 1
x --	x = x – 1
x += y	x = x + y
x -= y	x = x – y
x *= y	x = x * y
x /= y	x = x / y

Tabla 15: Asignaciones compuestas.

- **Operadores de comparación:** Las comparaciones de una variable o constante con otra se utilizan con frecuencia en las estructuras condicionales para testear si una condición es verdadera.

x = = y	x es igual a y
x != y	x no es igual a y
x < y	x es menor que y
x > y	x es mayor que y
x <= y	x es menor o igual que y
x >= y	x es mayor o igual que y

Tabla 16: Operadores de comparación.

- **Operadores lógicos:** Los operadores lógicos son normalmente una forma de comparar dos expresiones y devolver *TRUE* o *FALSE* dependiendo del operador. Existen tres operadores lógicos, *AND* (&&), *OR* (||) y *NOT* (!).

4.1.2.5 **Estructuras de control**

El lenguaje de programación de Arduino maneja, de forma similar a otros lenguajes, las siguientes estructuras de control:

- **if… else:** permite tomar decisiones del tipo, si la condición se cumple ejecuta estas intrucciones y si no ejecuta estas otras.
- **for:** se usa para repetir un bloque de sentencias un número determinado de veces.
- **while:** es un bucle de ejecución continua mientras se cumpla la expresión colocada entre paréntesis en la cabecera del bucle.
- **do… while:** funciona de la misma manera que el bucle while, con la salvedad de que la condición se prueba al final del bucle y no al principio.

4.1.2.6 **Funciones básicas**

En la Tabla 17 se enumeran las principales funciones predefinidas por Arduino, estas funciones facilitan la programación, permitiendo desde manejar las entradas y salidas de la placa, hasta permitir comunicaciones serie.

Función	Descripción
pinMode(pin,mode)	Esta instrucción es utilizada en la parte de configuración (*setup*) y sirve para configurar el modo de trabajo de un pin como entrada (*INPUT*) o salida (*OUTPUT*).
digitalRead(pin)	Lee el valor de un pin (definido como digital) dando un resultado nivel 1 lógico (HIGH) o nivel 0 lógico (LOW).
digitalWrite(pin,valor)	Asigna el valor HIGH o LOW a un pin digital.
analogRead(pin)	Lee el valor de tensión de un pin analógico con una resolución de 10 bits.
analogWrite(pin,valor)	Escribe un valor pseudo analógico usando modulación por ancho de pulso (PWM).
delay(ms)	Pausa el programa durante el tiempo especificado en milisegundos.
millis( )	Devuelve el número de milisegundos desde que la placa se empezó a ejecutar.
min(x,y)	Calcula el mínimo de dos números.
max(x,y)	Calcula el máximo de dos números.
randomSeed(seed)	Asigna un punto de partida para la función random ( ).
random(min,max)	Devuelve números pseudo aleatorios en un rango especificado.
serial.begin(rate)	Abre el puerto serie y fija su velocidad.
serial.println(data)	Imprime los datos en el puerto serie, seguido por salto de línea.

Tabla 17: Funciones básicas de Arduino.

4.2 **Software desarrollado para el simulador**

El programa de simulación es el programa desarrollado para emular una ECU real en la placa Arduino UNO. Dicho programa se encarga de atender a las peticiones OBD-II que le llegan a través del bus CAN haciendo uso de la placa CAN-BUS Shield. Para que la información OBD-II llegue a Arduino UNO es necesario el uso de dos librerías: la primera posibilita la comunicación serie (SPI) entre Arduino UNO y el módulo CAN-BUS Shield, la segunda permite a Arduino UNO enviar, recibir y configurar tramas CAN a través de la placa CAN-BUS Shield.

En esta sección estudiaremos en primera instancia el programa desarrollado que se ejecuta en la placa Arduino UNO, analizando su funcionabilidad y utilidad. Como se ha comentado anteriormente dicho programa principal necesita ciertas librerías para su correcto funcionamiento. Es por ello que acto seguido se estudiará la librería que el fabricante *Seeed* facilita, para habilitar de comunicación CAN a la placa Arduino UNO, y la librería SPI que a su vez es usada por esta última para la comunicación serie.

4.2.1 **Programa principal**

El programa principal que ha sido desarrollado en este proyecto, también conocido como programa de simulación es el que se encarga de que la placa Arduino UNO simule una ECU, procesando los mensajes de petición OBD-II y realizando las operaciones necesarias para contestar a dichas peticiones.

Este programa, al igual que cualquier programa Arduino, basa su funcionamiento en el uso de dos funciones que a su vez llaman a otras. Estas funciones son, como ya se ha comentado en el capítulo anterior, *setup y loop*. No obstante, estas funciones hacen uso de llamadas a unas segundas funciones, consiguiendo así un programa ordenado y bien estructurado.

El uso de todas estas funciones en conjunto, dotan al programa principal de las siguientes funcionalidades:

- Leer e interpretar mensajes OBD-II, siendo capaz de detectar peticiones en los Modos 01, 03 y 04.
- Contestar a las peticiones OBD-II en Modo 01, leyendo las entradas digitales y analógicas de la placa de que simula señales reales de un vehículo, generando la respuesta OBD-II con los datos obtenidos en tiempo real y enviando dicha respuesta vía CAN.
- Testeo continuo del vehículo simulado, almacenando DTCs en caso de error y encendiendo el indicador luminoso MIL. Además implementa un contador de kilómetros y de minutos desde que ocurrió un dicho error.
- Contestar a las peticiones OBD-II en Modo 03, comprobando el estado del vehículo simulado y respondiendo con el DTC correspondiente en caso de error.
- Borrar códigos de error cuando sea solicitado mediante el Modo 04 de OBD-II.

Para comprender mejor el funcionamiento del programa principal se estudiará a continuación la funcionalidad de cada una de sus funciones.

4.2.1.1 **Función setup**

Como ya se ha explicado en secciones anteriores la función *setup* es la encargada de recoger la configuración, es la primera función en ser ejecutada dentro del programa y se ejecuta una única vez.

En este caso se usa la función setup para inicializar la comunicación serie entre la placa Arduino y el PC, para definir los pines de E/S, para definir la interrupción que simulará un error y generará un DTC y para inicializar la comunicación CAN.

La Figura 40 contiene el diagrama de flujo de la función *setup*:

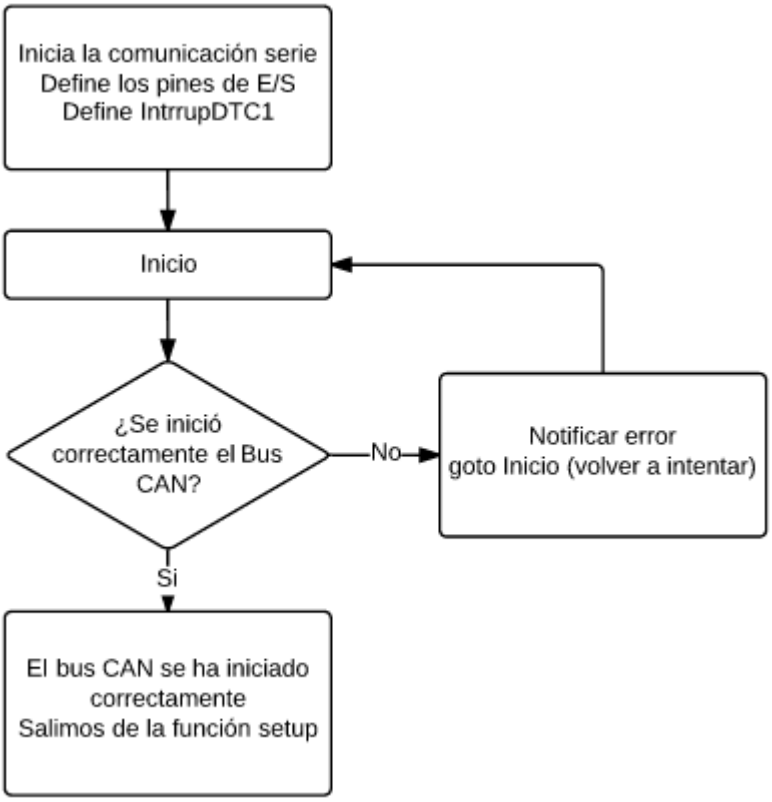


Figura 40: Diagrama de flujo de la función setup.

4.2.1.2 **Función loop**

Esta función forma el núcleo principal del programa, contiene el código que se ejecutará continuamente mientras la placa siga alimentada.

El programa desarrollado se encarga de recibir los mensajes OBD-II y llamar a la función encargada de responder dependiendo del modo OBD-II en el que se ha recibido la petición. Además comprueba si se ha producido la interrupción de fallo, en caso afirmativo, actuará encendiendo el indicador luminoso y llamando a la función encargada de calcular los kilómetros recorridos mientras dicho indicador esteé encendido.

La Figura 41 contiene el diagrama de flujo de la función *loop*:

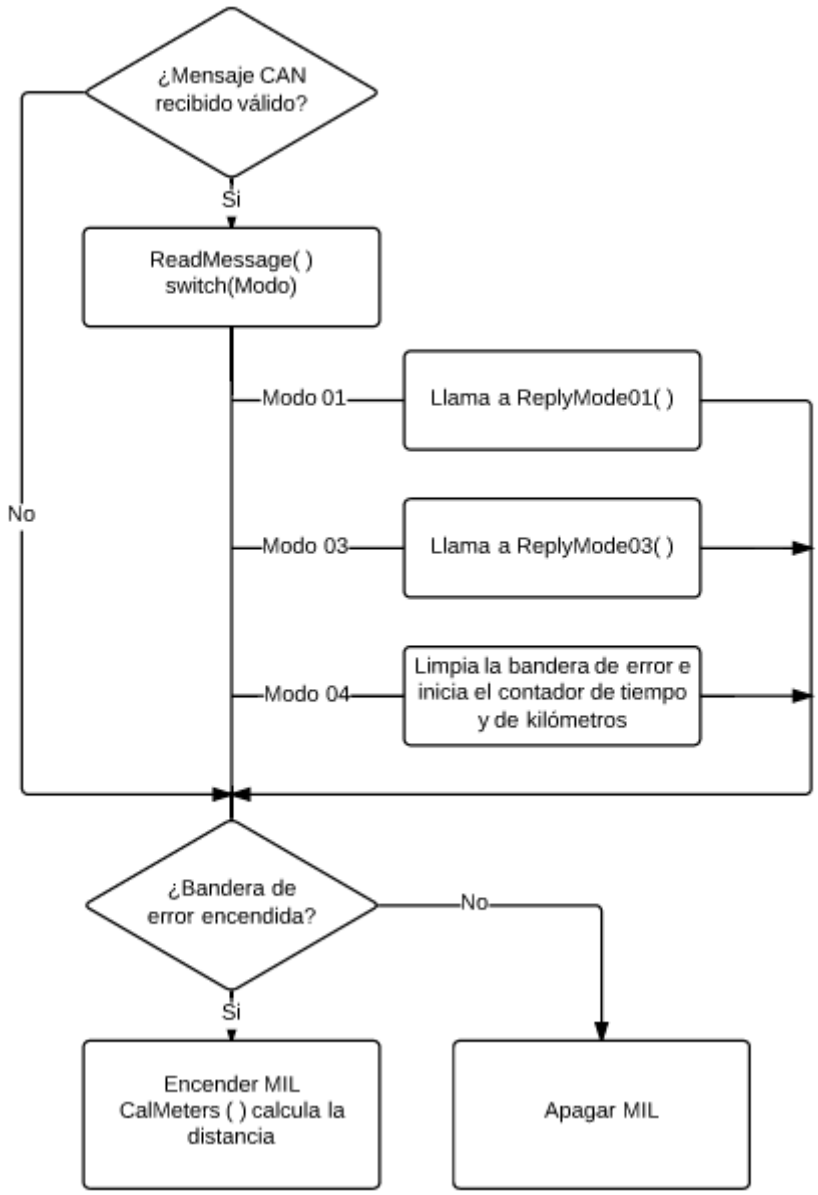


Figura 41: Diagrama de flujo de la función loop.

4.2.1.3 **Función InterruptDTC1**

Las interrupciones hardware, se diseñaron por la necesidad de reaccionar a suficiente velocidad en tiempos muy cortos a los que la electrónica trabaja habitualmente. Estas interrupciones activarán una función que se ejecutará de forma asíncrona, sin planificación, cuando ocurra un cierto suceso electrónico.

La interrupción *InterruptDTC1* se usa para detectar un error, que se simula mediante un pulsador, activando la bandera de error e iniciando los contadores de tiempo y kilómetros recorridos.

4.2.1.4 **Función ReadMessage**

La función *ReadMessage* es la encargada de leer los mensajes OBD-II, devolviendo dicho mensaje en una tabla y almacenando el modo OBD-II y otros datos de importancia de la trama OBD-II. Para realizar dicha tarea la función *ReadMessage* hace uso de las funciones *readMsgBuf* y *getCanId* de la librería mcp_can.h, librería que facilita el fabricante Seeed para dar conectividad CAN a la placa Arduino a través del módulo CAN-BUS Shield.

La Figura 42 contiene el diagrama de flujo de la función *ReadMessage*:

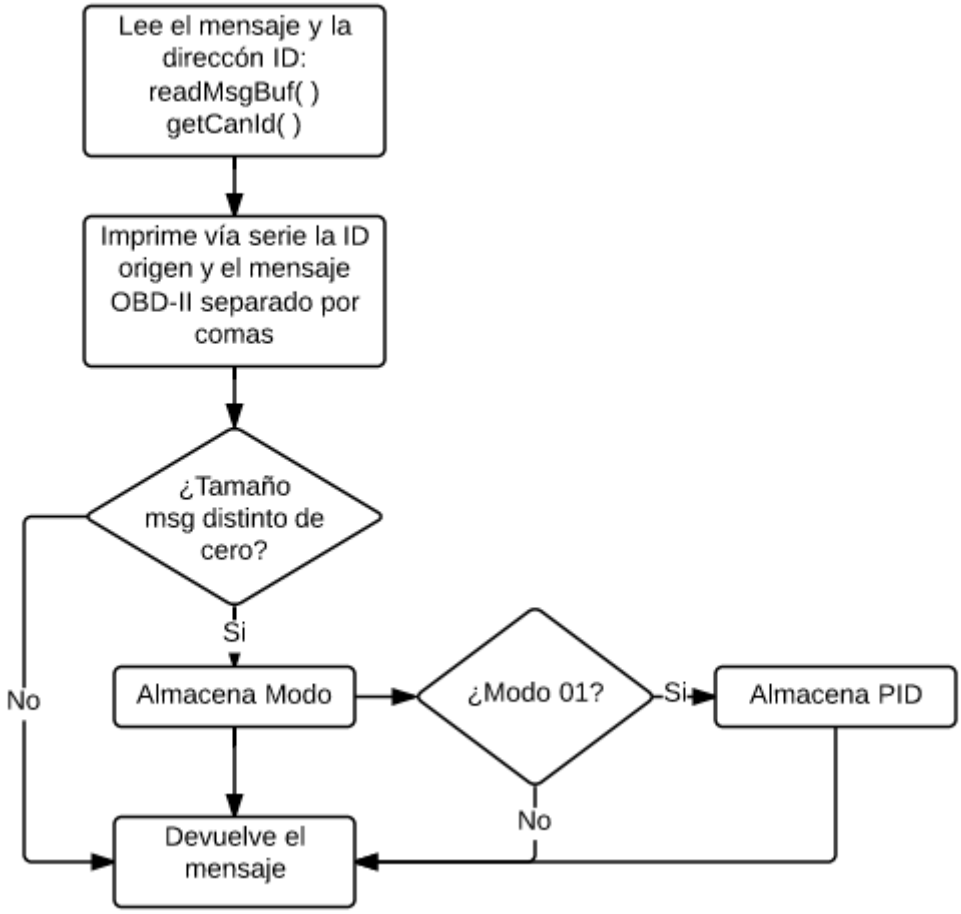


Figura 42: Diagrama de flujo de la función ReadMessage.

4.2.1.5 **Función ReplyMode01**

La función *ReplyMode01* es la función encargada de responder, mediante un mensaje de respuesta OBD-II, a una petición realizada previamente en este modo. Como ya se ha descrito en la sección dedicada a OBD-II las peticiones en este modo van acompañadas de un identificador (PID), este identificador indica el parámetro requerido a la ECU, por lo tanto que es el cual debemos incluir al responder.

El funcionamiento de esta función está basado en una estructura *switch-case*, en la cual se formará la respuesta OBD-II dependiendo del PID que se haya requerido. El funcionamiento es

muy simple, primero se define un mensaje OBD-II de respuesta por defecto vacío, este mensaje se rellena dependiendo del valor de la variable que indica el PID y por último se envía la trama CAN haciendo uso de la función *sendMsgBuf*.

La Figura 43 contiene el diagrama de flujo de la función *ReplyMode01*:

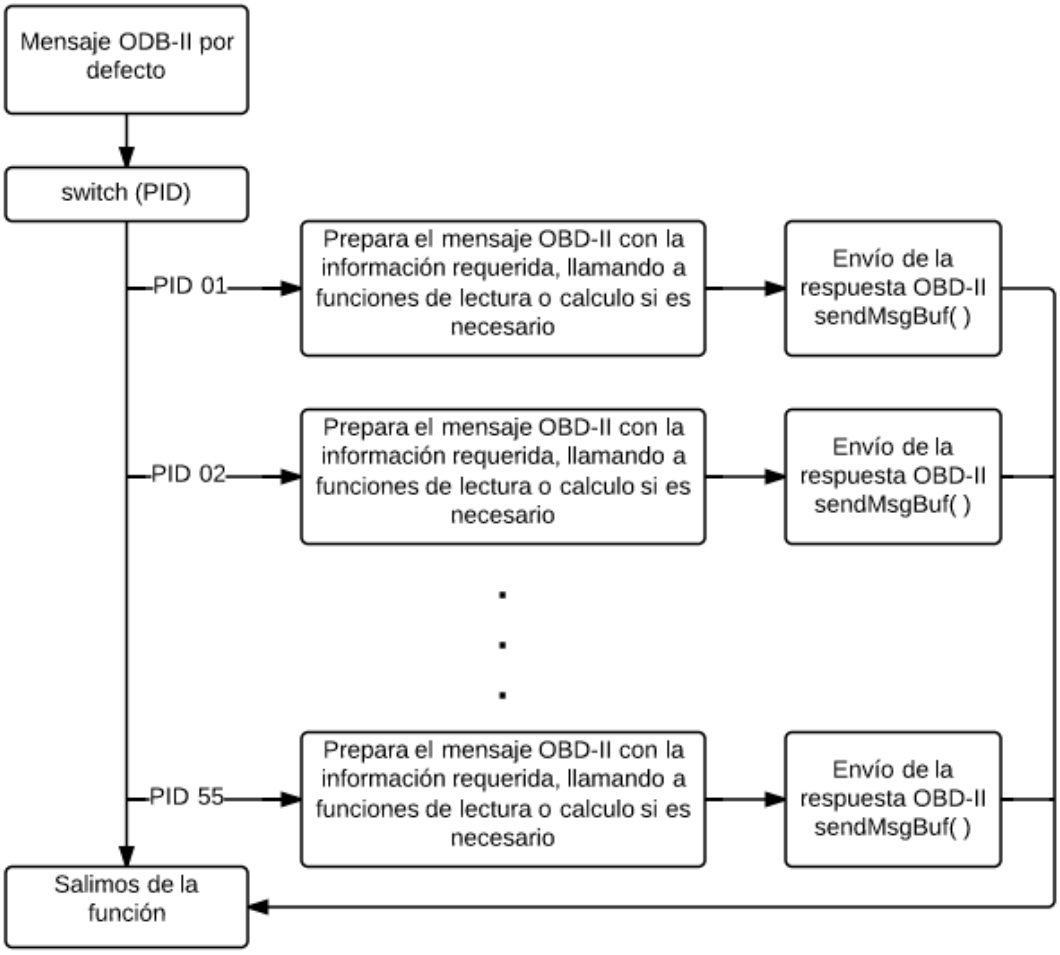


Figura 43: Diagrama de flujo de la función ReplyMode01.

4.2.1.6 **Función ReplyMode03**

La función *ReplyMode03* es la equivalente a *ReplyMode01* para el Modo 03 OBD-II. Esta función es la encargada de enviar los códigos de error almacenados en la ECU en caso de existir alguno. De lo contrario enviará el mensaje definido por la norma OBD-II para indicar que no hay error.

El funcionamiento básico de *ReplyMode03* se basa en suponer que no hay error, por lo que se define un mensaje OBD-II por defecto para indicar que no existen DTCs almacenados. Acto seguido se comprueban la bandera de error, en caso de existir error alguno se crea el mensaje OBD-II que informe al programa de diagnóstico del DTC almacenado, de lo contrario se envía el mensaje por defecto.

La Figura 44 contiene el diagrama de flujo de la función *ReplyMode03*:

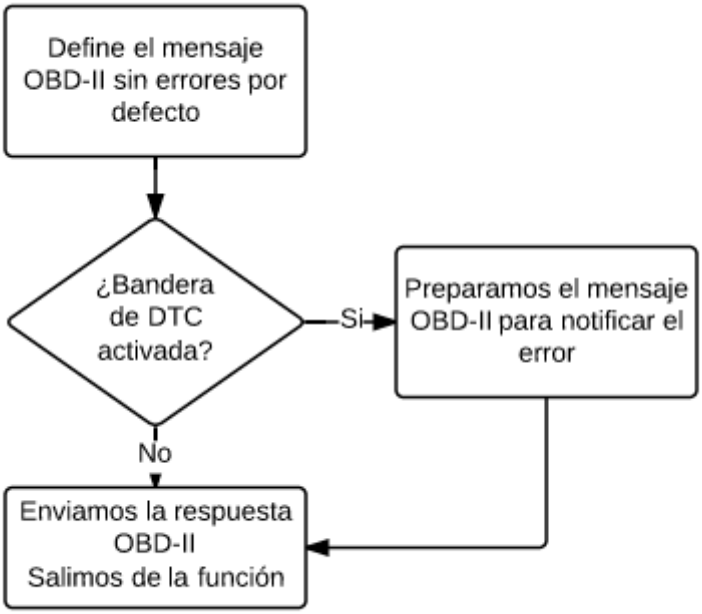


Figura 44: Diagrama de flujo de la función ReplyMode03.

4.2.1.7 **Otras funciones**

El resto de funciones son utilizadas para adquirir los datos necesarios de la placa de simulación con el fin de responder a los mensajes OBD-II en Modo 01. Para responder a las peticiones en dicho modo leen las entradas analógicas y realizan todos los cálculos necesarios. Dentro de este grupo de funciones se encuentran las siguientes:

- **ReadSpeed:** Función para calcular la velocidad actual según la posición del potenciómetro 1 de la placa de simulación.
- **ReadRPM:** Función para calcular el número de revoluciones por minuto según la posición del potenciómetro 1 de la placa de simulación.
- **ReadThrottle:** Función para calcular la posición del acelerador según la posición del potenciómetro 1 de la placa de simulación.
- **ReadBateryLvl:** Función para leer el nivel de batería según la posición del potenciómetro 2 de la placa de simulación.
- **CalMeters:** Calcula los metros por segundo recorridos mientras que el indicador MIL este encendido.

4.2.2 **Librerías**

El programa principal hace uso de la librería mcp_can.h que el fabricante Seeed facilita de forma totalmente gratuita para facilitar el manejo del módulo CAN-BUS Shield. A su vez, esta

librería necesita de la librería SPI.h de Arduino para la comunicación serie entre la placa Arduino y el módulo CAN.

4.2.2.1 **Librería mcp_can.h**

La librería mcp_can desarrollada por el fabricante *Seeed Technology* facilita el uso del módulo CAN-BUS Shield, como ya se ha comentado en secciones anteriores. Esta librería es software libre, es decir, se encuentra disponible para su descarga de forma gratuita y puede ser modificada según las necesidades del programador.

La librería mcp_can.h permite el uso de las siguientes funciones para dotar a la placa Arduino de conectividad CAN:

- ***mcp_can:*** establece el pin CS para la comunicación serie.
- ***begin:*** inicia el bus CAN a la velocidad que se le especifica como parámetro de entrada.
- ***init_Mask:*** establece una máscara para los identificadores CAN (ID).
- ***init_Filt:*** establece filtros para los identificadores CAN (ID).
- ***sendMsgBuf:*** envía un mensaje a través del bus CAN, pasando como parámetros de entrada el identificador, el RTR, la longitud de los datos y los datos.
- ***readMsgBuf:*** lee un mensaje CAN recibido.
- ***readMsgBufID:*** lee un mensaje CAN recibido de una dirección ID específica.
- ***checkReceive:*** comprueba si se ha recibido algún mensaje CAN.
- ***checkError:*** comprueba si ha ocurrido algún error.
- ***getCanId:*** toma la dirección ID, cuando se ha detectado un mensaje recibido.
- ***isRemoteRequest:*** activa una bandera si se ha recibido una trama de petición remota (*remote request*).
- ***isExtendedFrame:*** activa una bandera si se ha recibido una trama CAN *Extended*.

4.2.2.2 **Librería SPI.h**

SPI (*Serial Peripheral Interface*) es básicamente un bus de comunicación a nivel de circuitos integrados. La transmisión de datos se realiza en serie, es decir un bit después de otro. El bus SPI se define mediante 4 pines:

- **SCLK o SCK**: Señal de reloj del bus. Esta señal rige la velocidad a la que se transmite cada bit.
- **MISO (*Master Input Slave Output*):** Es la señal de entrada a nuestro dispositivo, por aquí se reciben los datos desde el otro circuito integrado.

- **MOSI (*Master Output Slave Input*):** Transmisión de datos hacia el otro circuito integrado.
- **SS o CS:** *Chip Select o Slave Select*, habilita el circuito integrado hacia el que se envían los datos. Esta señal es opcional y en algunos casos no se usa.

La gestión del protocolo SPI en Arduino se realiza a través de la librería SPI. Esta interfaz se inicializa automáticamente cuando la librería SPI se incluye en un código. En el caso de Arduino UNO, los pines se inicializan de la siguiente forma:

Arduino	MOSI	MISO	SCK	SS (slave)	SS (master)
UNO	11 or ICSP-4	12 or ICSP-1	13 or ICSP-3	10	-

Tabla 18: Pines para la comunicación SPI en Arduino UNO.

La configuración por defecto del protocolo SPI en Arduino establece que el esclavo será él mismo, que se van a transferir en primer lugar los bytes más significativos de cada byte (MSB), que el modo seleccionado es el 0 y se establece la frecuencia de reloj en una cuarta parte de la frecuencia del sistema. Las funciones disponibles para modificar estos parámetros son:

- ***mode (byte config):*** Permite modificar el registro de configuración del SPI. Si hay varios dispositivos usando SPI y cada uno necesita un modo distinto, esta función se deberá llamar antes de acceder a cada dispositivo en particular.
- ***Byte.transfer (byte b):*** Envía y recibe un byte a través del bus SPI.
- ***Byte.transfer (byte b, byte delay)***: Esta función es idéntica a la anterior, sólo se diferencia en que antes de acometer la operación espera un número de microsegundos especificados en el parámetro delay. Es útil cuando hay que esperar un tiempo antes de iniciar una transferencia de datos.

4.3 Software de diagnóstico

El último elemento software a analizar es el software encargado de realizar el diagnóstico OBD-II, también conocido como escáner OBD-II. Este software se instalará en un ordenador personal con sistema operativo Windows 8, por lo que debe ser compatible con dicho sistema operativo. Por otro lado dicho software deberá soportar el protocolo CAN, los distintos modos de funcionamiento de OBD-II y la interfaz ELM327.

Un software que reúne las características anteriormente nombradas es el ScanMaster-ELM, dicho programa reúne una serie de ventajas frente a otros similares como ScanTool. ScanMaster-ELM presenta una interfaz gráfica muy parecida a las que usan los programas de

diagnóstico profesionales. Esta interfaz resulta mucho más amigable e intuitiva que la presentada por ScanTool.

Por estos motivos se ha optado por el software de diagnóstico ScanMaster-ELM. Este software reúne las siguientes características básicas:

- Este software soporta los siguientes protocolos:
    - SAE J1850 PWM (41.6 Kbaud)
    - SAE J1850 VPW (10.4 Kbaud)
    - ISO 9141-2 (5 baud init, 10.4 Kbaud)
    - ISO 14230-4 KWP (5 baud init, 10.4 Kbaud)
    - ISO 14230-4 KWP (fast init, 10.4 Kbaud)
    - ISO 15765-4 CAN (11 bit ID, 500 Kbaud)
    - ISO 15765-4 CAN (29 bit ID, 500 Kbaud)
    - ISO 15765-4 CAN (11 bit ID, 250 Kbaud)
    - ISO 15765-4 CAN (29 bit ID, 250 Kbaud)
- Detecta automáticamente el protocolo correspondiente al vehículo que está conectado.
- Soporta identificadores de parámetros (PIDs) genéricos SAE J1979 (Modo $01) de $00 a $4E.
- Muestra con gráficos o indicadores toda la información o solamente la seleccionada, respecto de las mediciones soportadas por el controlador OBD-II del vehículo.
- Permite ver la condición del sistema del vehículo cuando una emisión relacionada con un código de error se ha obtenido en el modo *FreezeFrame*.
- Lee códigos de error DTC (y sus descripciones estándar según SAE) que provocan que la luz del "*Check Engine*" esté encendida. Además, muestra los códigos de error no estandarizados por SAE y las descripciones entregadas por la ECU del vehículo. Soporta más de 4200 descripciones de códigos de error genéricos de SAE J2012 y más de 3600 códigos OBD-II mejorados, o los códigos definidos por el mismo fabricante del equipo (OEM).
- Elimina toda la información proveniente de diagnósticos realizados (Modo $04).
- Revisa los resultados de los test de sensores de oxígeno realizados por el módulo de control del tren de fuerza del vehículo en modo "Sensor de Oxígeno" (Modo $05).
- Muestra los resultados de los Test No-Continuos específicos del fabricante del vehículo realizados (Datos Modo $06) con el modo "Resultado de Test de Monitoreo".
- Presenta diagnóstico de monitoreo continuo realizado mientras el vehículo ha estado en funcionamiento. Esto incluye reporte de fallas que no han sido detectadas aun a través del modo "Códigos de Error Pendientes".

- Muestra información específica correspondiente al vehículo (VIN, ID de calibración, número de verificación de calibración, seguimiento del desempeño en funcionamiento).
- Unidades de medidas métricas e inglesas.
- Métodos de conexión directa para interfaces USB, Bluetooth y WLAN (*Wireless Local Area Network*).

ScanMaster-ELM presenta una interfaz gráfica organizada en pestañas (véase la Figura 45), cada una de ellas ofrece una utilidad OBD-II. En las siguientes secciones se tratará con más profundidad las posibilidades que ofrecen cada una de estas pestañas.

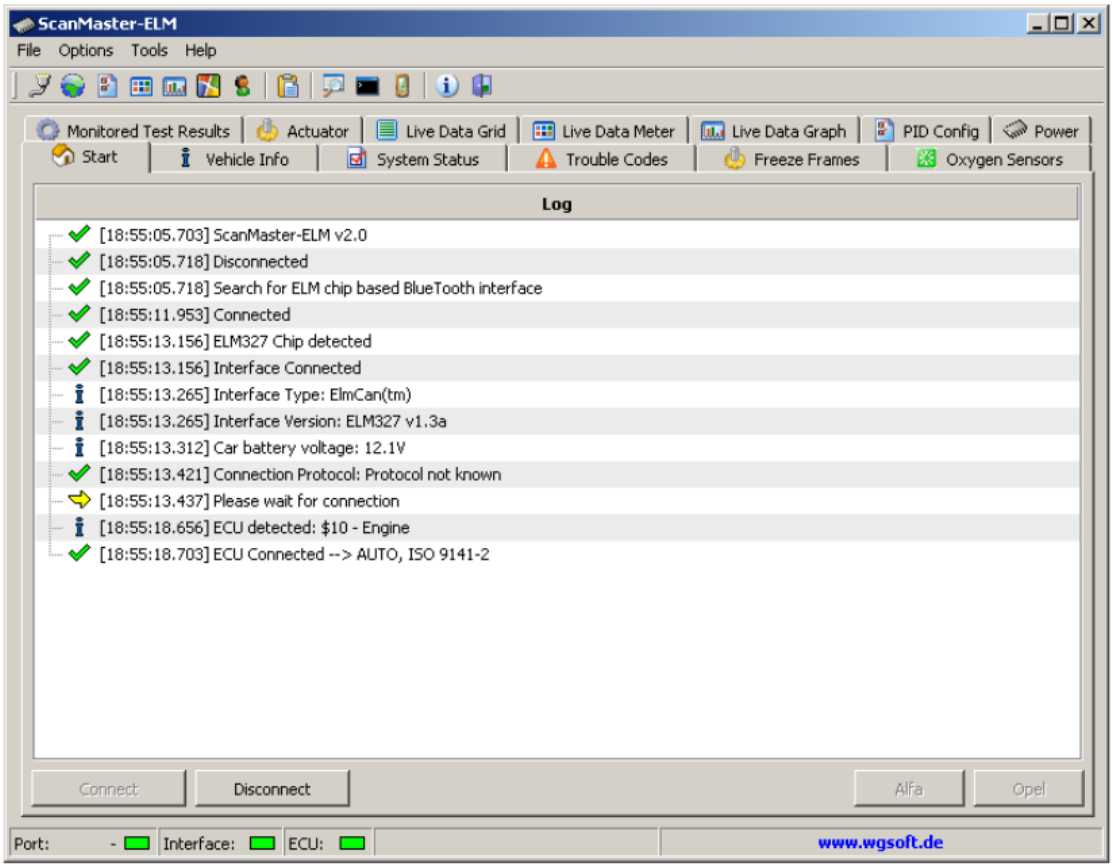


Figura 45: Interfaz basada en ventanas ScanMaster-ELM.

Antes de empezar a usar el programa, es recomendable fijar una serie de opciones, para ello hay que ir al menú *Options*, que se encuentra en la esquina superior izquierda de la ventana principal. Una vez abierto dicho menú permite configurar las siguientes opciones:

- ***Communication***: permite establecer el tipo de interfaz entre serie, USB, Bluetooth y WLAN.
- ***Protocol***: permite seleccionar el protocolo de diagnóstico (SAE J1850, CAN, etc.), además permite usar la opción protocolo automático por si no se conoce dicho protocolo.

- ***Language***: permite seleccionar el lenguaje y el sistema de medida.
- ***General***: permite activar opciones de guardado de datos y archivos de ayuda.
- ***PIDs***: permite priorizar unos PIDs frente a otros.
- ***Graph***: opciones relacionadas con los gráficos del programa.
- ***Skins***: permite cambiar la apariencia del programa.
- ***User Info***: contiene los datos del usuario que pueden ser modificados.

4.3.1 Ventana Start

La ventana de inicio (*start*) tiene dos botones, uno para conectar el programa con la ECU (*Connect*) y otro para desconectarlo (*Disconnect*). Además añade otro par de botones *Alfa* y *Opel*, que se utilizan con algunos coches antiguos de estas marcas para lograr la conexión con sus ECUs.

En la ventana de inicio se muestra toda la información relativa a la conexión y el resultado, *Connected* o *Disconnected*.

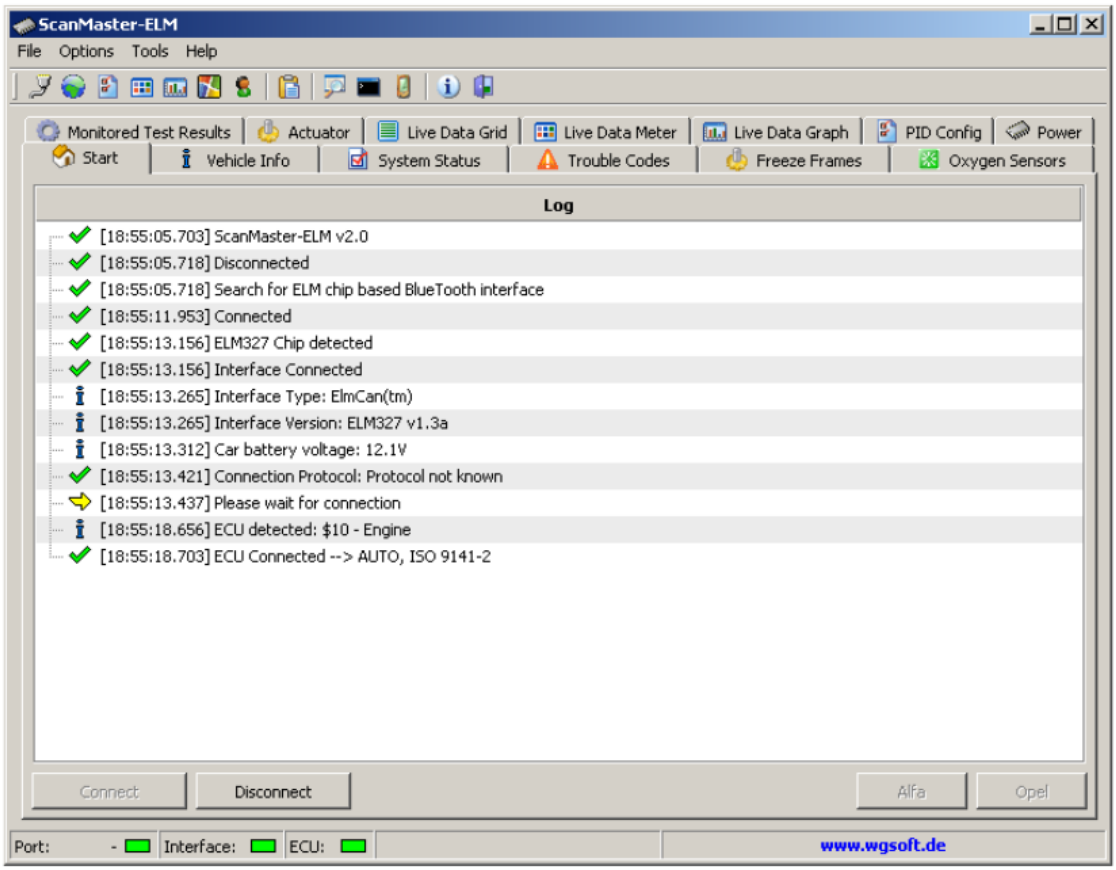


Figura 46: Ventana Start.

4.3.2 Ventana *Vehicle Info*

Esta ventana muestra la información general sobre el vehículo siempre y cuando sea soportada por el vehículo.

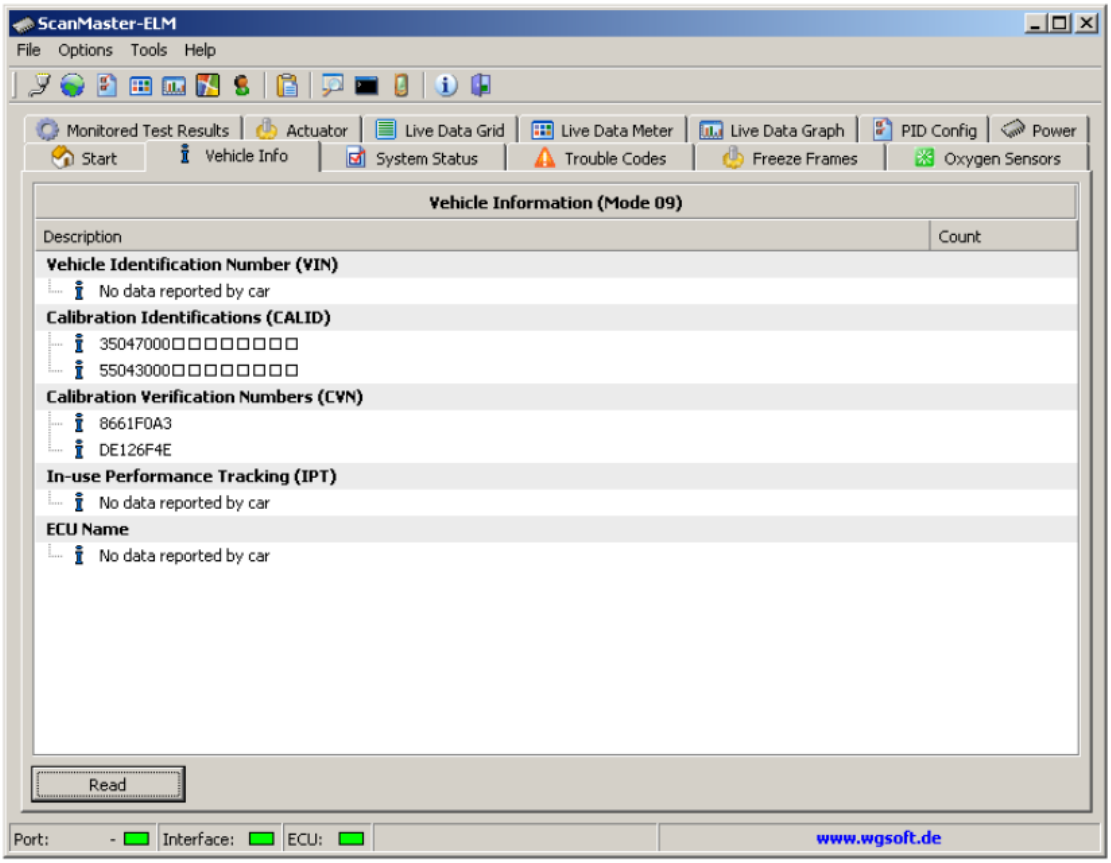


Figura 47: Ventana Vehicle Info.

4.3.3 **Ventana *System Status***

Muestra el estado del sistema (*System Status*), es decir, el estado del indicador luminoso MIL, el número de DTCs almacenados y los tests que soporta la ECU.

Al igual que muchas otras ventanas incluye el botón *Read* para actualizar el estado actual del sistema.

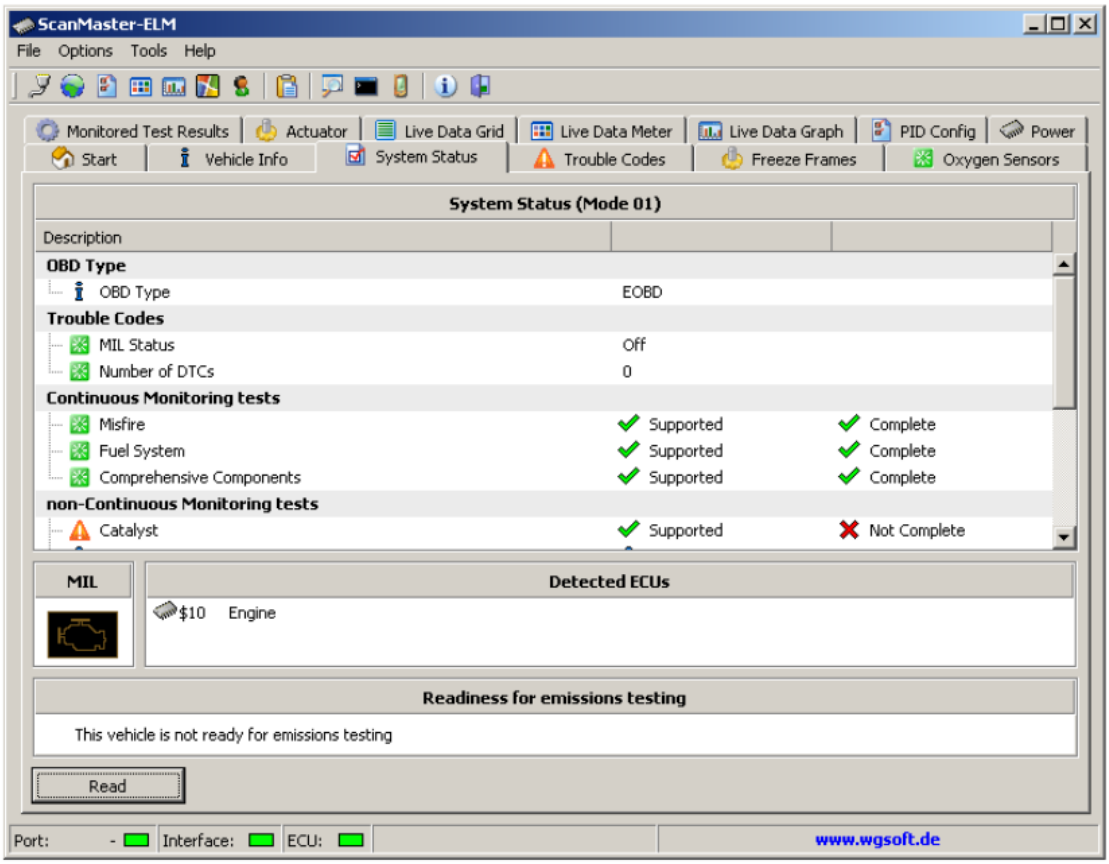


Figura 48: Ventana System Status.

4.3.4 **Ventana *Trouble Codes***

La ventana *Trouble Codes* muestra los DTCs almacenados, pendientes y permanentes. Estos últimos en caso de ser soportados por la ECU.

Añade los botones *Read* y *Clear*. El primero para leer los DTCs almacenados, pendientes y permanentes de la ECU, mientras el segundo borra los DTCs y apaga el indicador luminoso MIL.

Además dispone de una pestaña en la parte inferior derecha que permite seleccionar que tipos de DTCs maneja la ECU:

- ***Generic:*** son los DTCs definidos por el estándar.
- ***Manufacturer:*** son los DTCs definidos por el fabricante, al desplegar la pestaña permite seleccionar DTCs específicos de ciertos fabricantes.

Además el programa integra una base de datos con los códigos de error que usan las diferentes marcas. Se puede acceder a dicha base de datos a través del icono en forma de lupa de la barra de herramientas. El anterior icono se encuentra resaltado en la Figura 49.

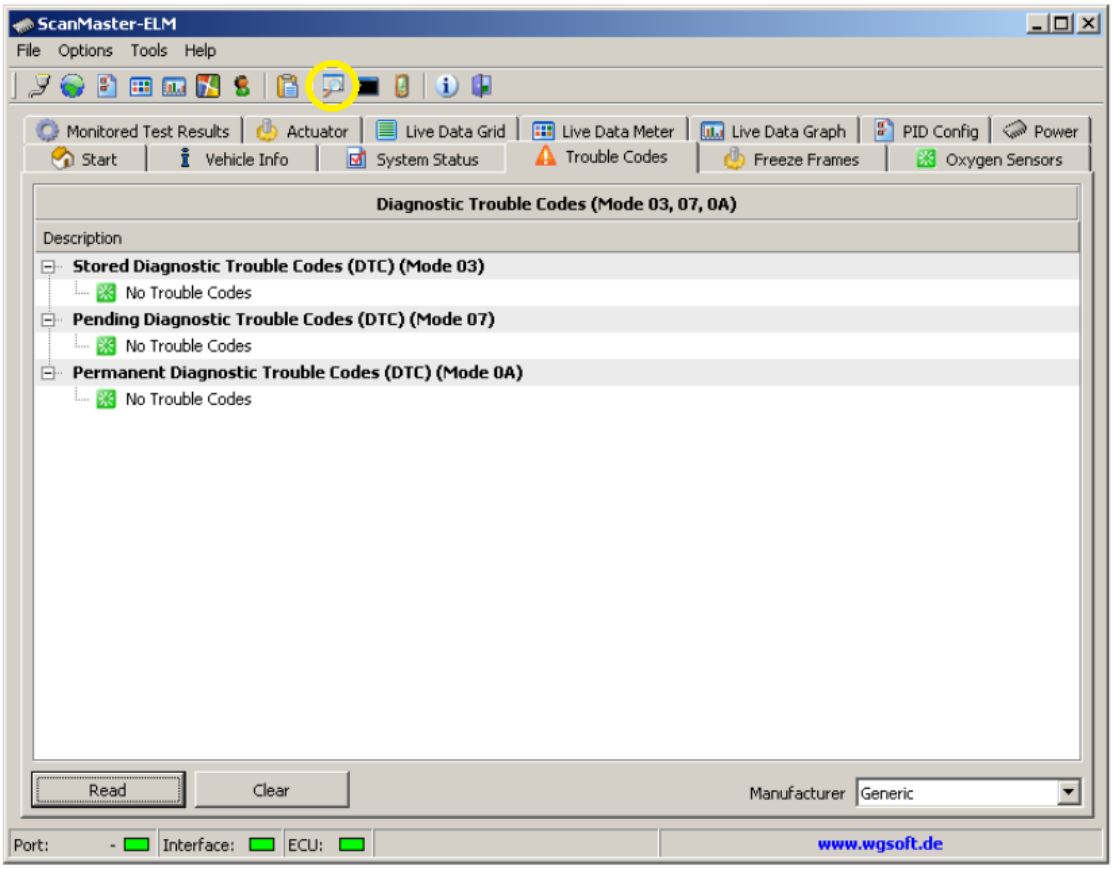


Figura 49: Ventana Trouble Codes.

4.3.5 **Ventana *Freeze Frames***

En esta ventana se puede pulsar el botón *Read* para acceder al cuadro de datos congelados (*Freeze Frames*) de la ECU. También permite seleccionar el número de trama requerido.

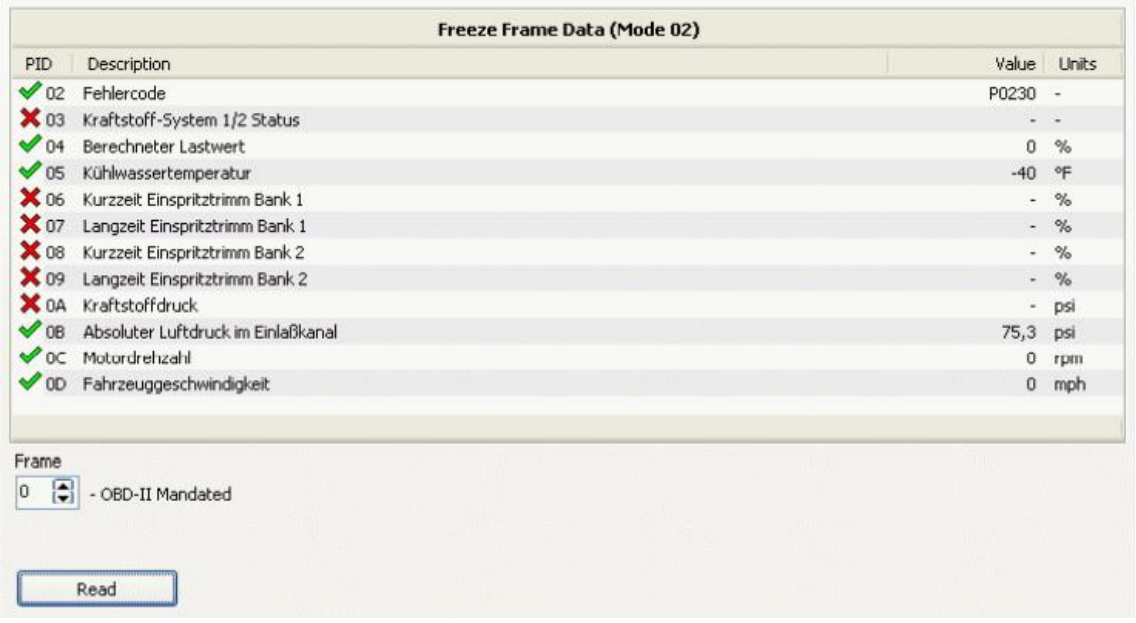


Figura 50: Ventana Freeze Frame Data.

4.3.6 **Ventana *Oxygen Sensor***

La ventana Sensor de Oxigeno (*Oxigen Sensor*) muestra los resultados del test de oxígeno del vehículo. Los resultados son medidos por la ECU y representados por ScanMaster. La mayoría de los coches no soporta este modo por lo que para leer los resultados de dicho test se debe utilizar la ventana *Monitored Test Resuls*.

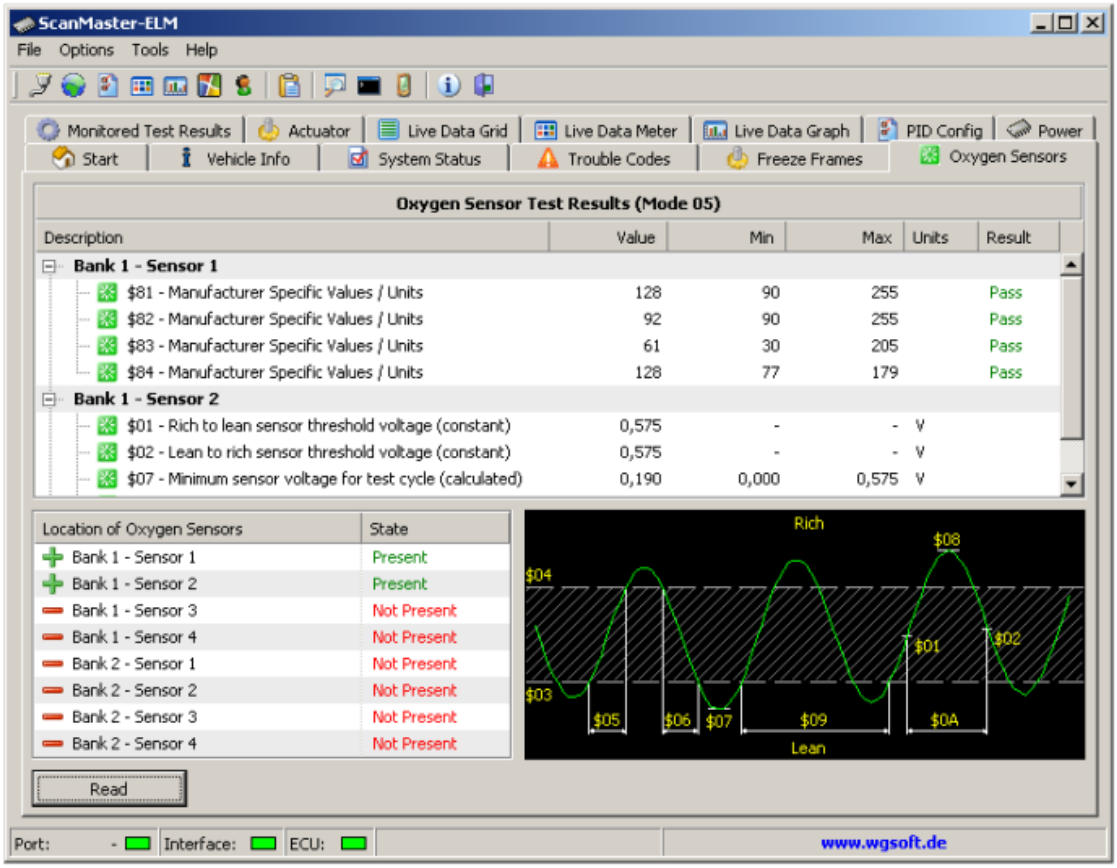


Figura 51: Ventana Oxygen Sensors.

4.3.7 **Ventana *Monitored Test Results***

Este modo permite el acceso a los resultados de los tests de diagnóstico de a bordo. Los distintos tests son requeridos según su identificador.

El vehículo es el encargado de realizar el test y almacenar los datos con su respectivo identificador, estos valores son retransmitidos una vez seleccionados.

En caso de que desde el último reseteo o limpiado de la ECU no se hubiera realizado el test que se ha solicitado, la ECU responderá con valores nulos.

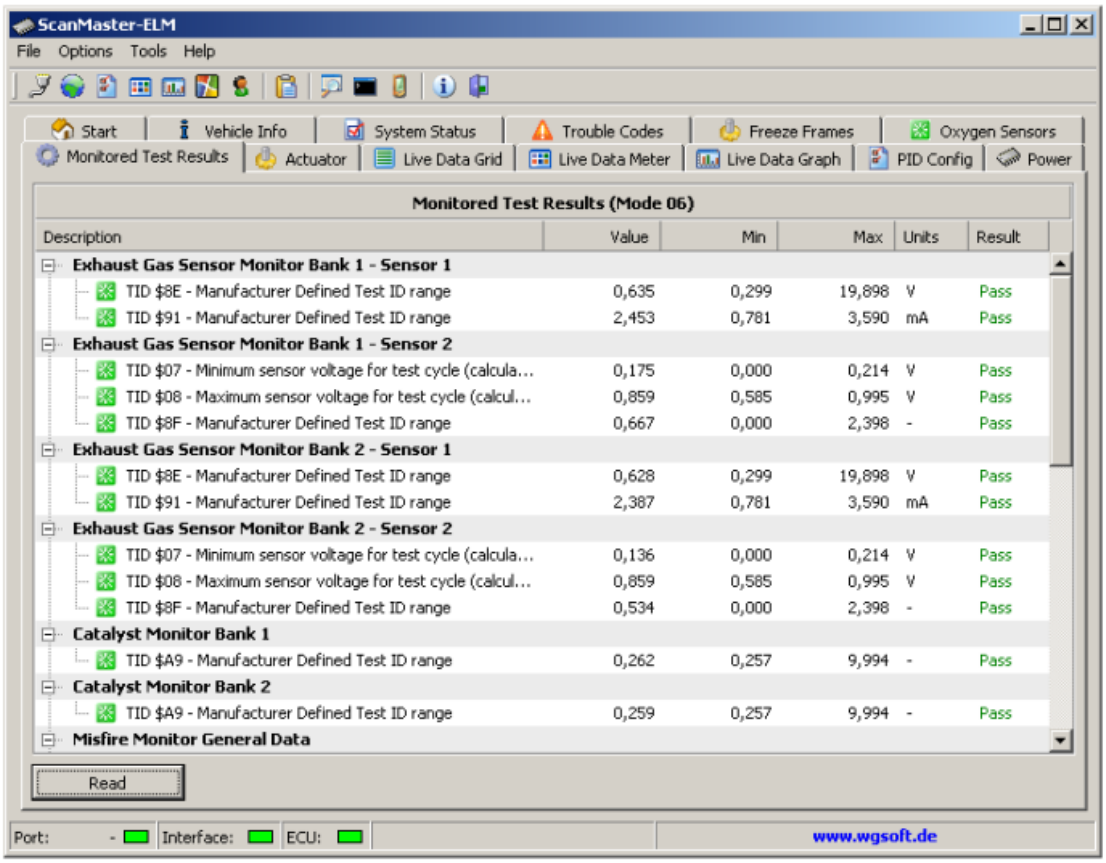


Figura 52: Ventana Monitored Test Results.

4.3.8 **Ventanas *Live Data***

Tres ventanas son las encargadas de mostrar datos en vivo, y aunque todas muestran la misma información, difieren las unas de las otras en la forma de plasmar dicha información gráficamente. Estas ventanas son:

- ***Live Data Meter*:** muestra los datos anteriores en formato digital. Solo pueden ser mostrados cuatro datos al mismo tiempo, que pueden ser seleccionados mediante su número PID.

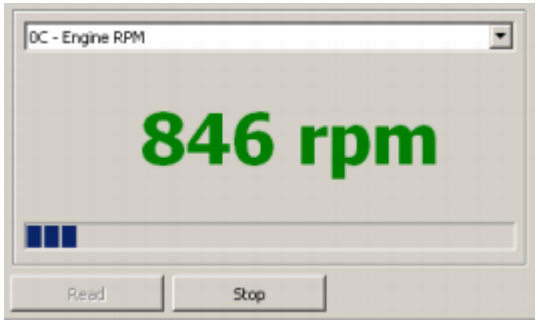


Figura 53: Ventana Live Data Meter.

- ***Live Data Grid*:** muestra la lista de datos en vivo que soporta el vehículo, acompañados de su valor actual, máximo, mínimo y unidades.

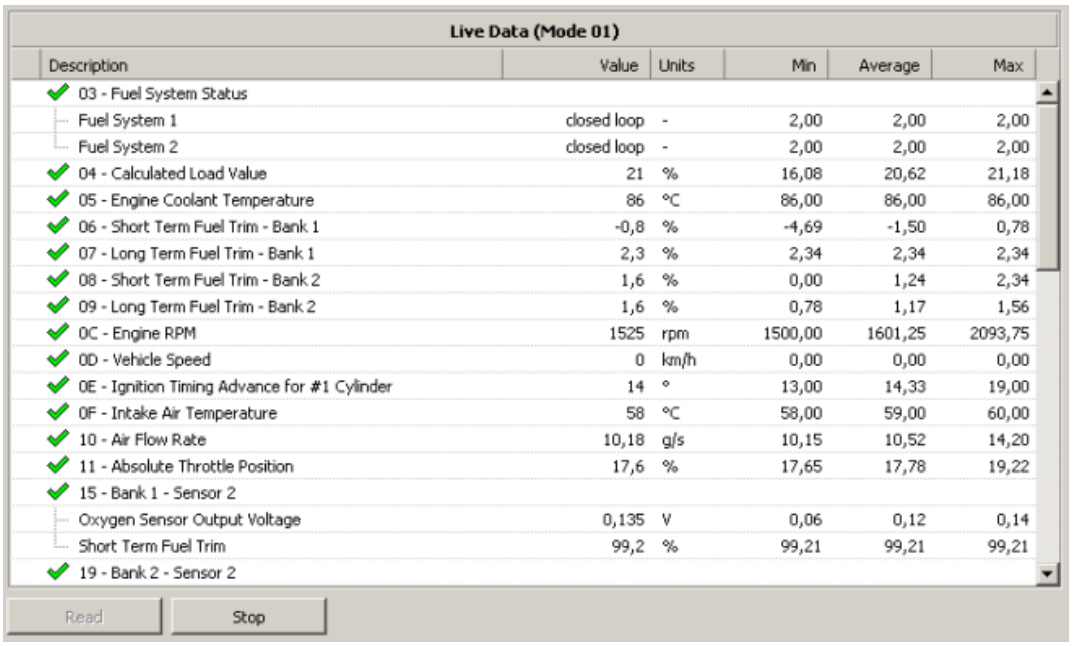


Figura 54: Ventana Live Data.

- ***Live Data Graph*:** muestra los datos en formato gráfico, solo uno puede ser mostrado al mismo tiempo. El programa permite guardar estas graficas en distintos formatos para su posterior análisis.

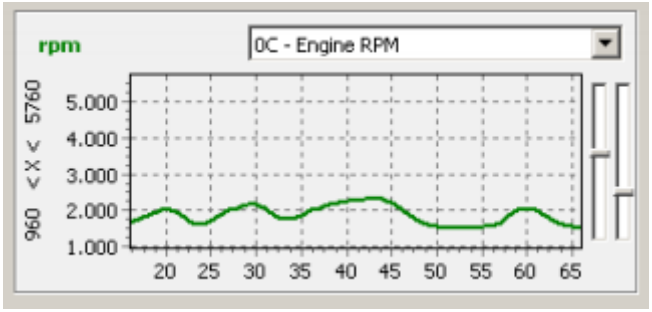


Figura 55: Ventana Live Data Graph.

4.3.9 **Ventana *PID Config***

En esta ventana se pueden modificar los parámetros que serán mostrados en las ventanas *Live Data*. No todos los vehículos soportan todos los parámetros (PIDs), al conectar con la ECU se le solicitan los PIDs soportados y se muestran en esta ventana.

Adicionalmente esta ventana permite configurar cada PID manualmente, es decir, se puede definir la descripción del PID, las unidades de medida, la prioridad y los límites para su representación gráfica.

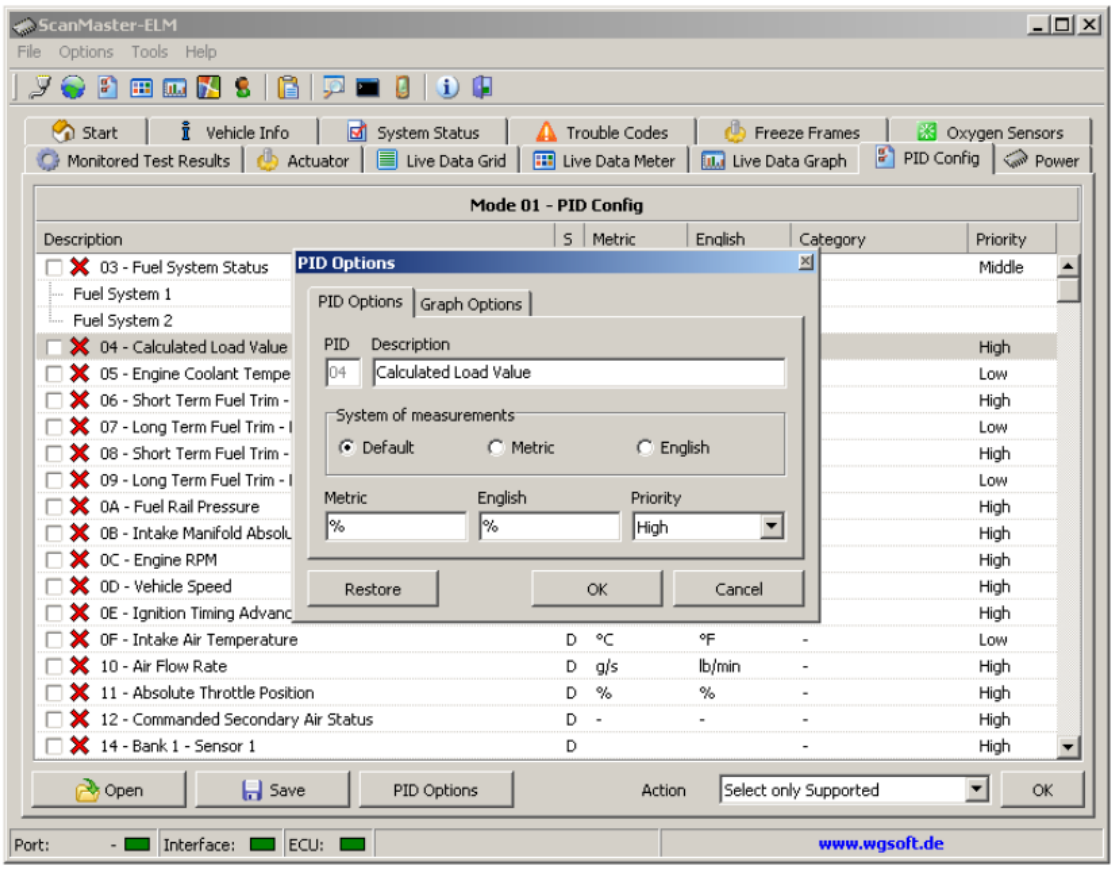


Figura 56: Ventana PID Config.

4.3.10 **Ventana *Power***

La ventana Power permite realizar dos tipos de test:

- ***Acceleration Test***: permite realizar pruebas de aceleración, para ello se tiene que fijar una velocidad final, pulsar el botón *Run*, y pisar el acelerador hasta llegar a la velocidad marcada. El resultado obtenido es una gráfica de aceleración como la que se muestra en la Figura 57.

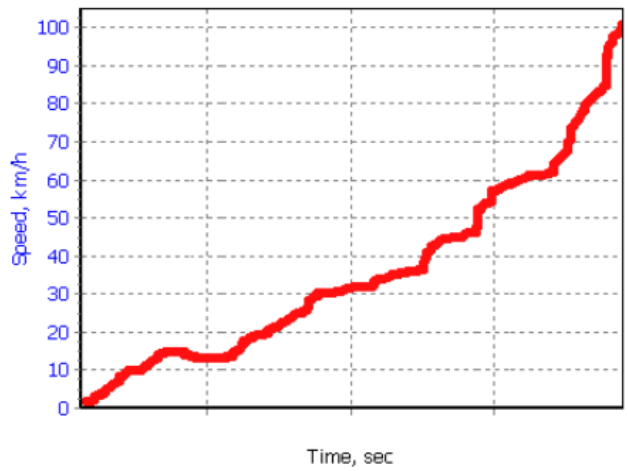


Figura 57: Resultados del Acceleration Test.

- ***Dyno Test:*** en esta ventana se pueden realizar pruebas para representar la aceleración frente a las RPM, la relación potencia-par y las RPM frente a la velocidad en kilómetros por hora. En la Figura 57 se pueden apreciar los resultados de las pruebas anteriormente nombradas.

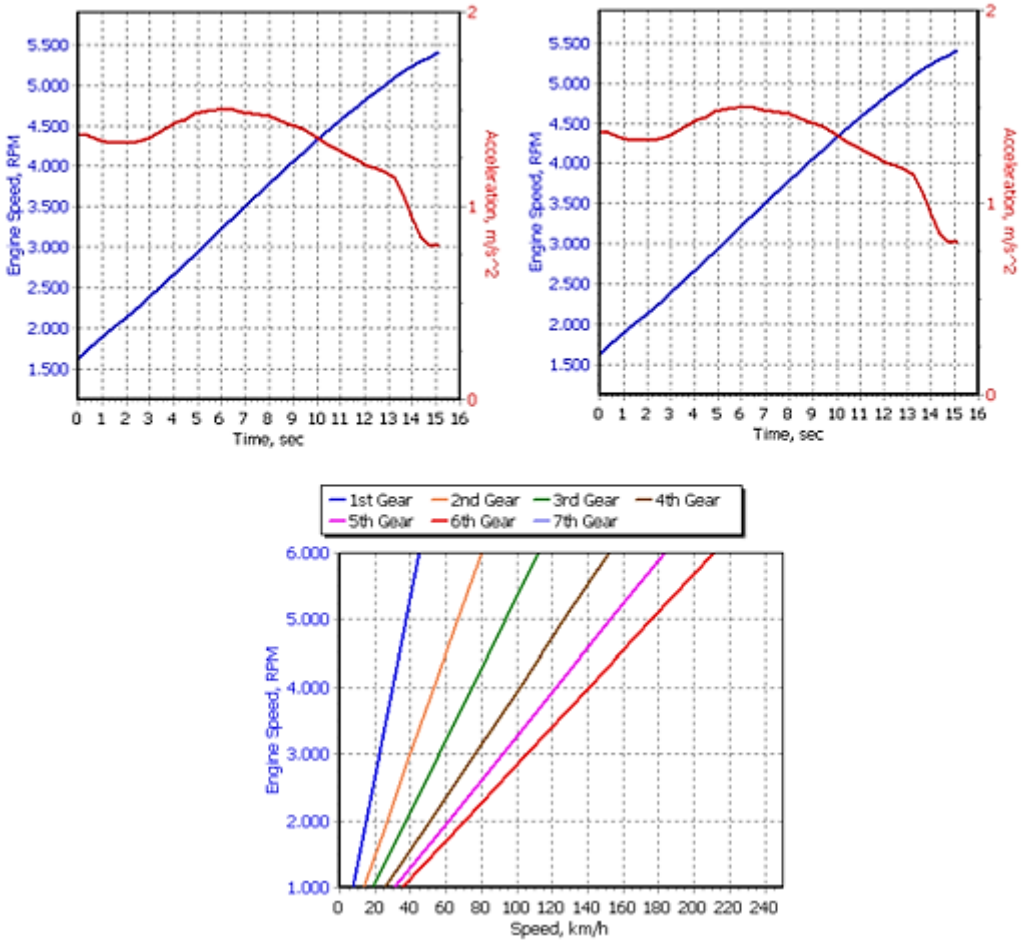


Figura 58: Resultados Dyno Test.

5 Conclusiones finales

A modo de conclusiones, en este último capítulo se sintetizan los puntos principales que se han ido tratando a lo largo del proyecto. Asimismo se aprovecha para describir posibles mejoras sobre el trabajo realizado, muchas de las cuales se han ido ya exponiendo.

5.1 Resumen general

Con el trabajo realizado se ha conseguido desarrollar un sistema de diagnóstico OBD-II de un vehículo, desarrollando un simulador de una ECU de bajo coste basada en Arduino UNO, un módulo CAN y una pequeña placa de simulación. En primer lugar se ha realizado un acercamiento teórico al concepto de sistemas de diagnósticos y a la normativa existente en este campo. Durante esta aproximación, se ha estudiado el principal estándar en materia de diagnósticos de vehículos a nivel mundial, el estándar OBD-II. En cuanto a los protocolos que implementan dicho estándar, se ha hecho especial hincapié en el protocolo CAN, ya que es el protocolo más usado en materia de las comunicaciones internas de los vehículos en la actualidad.

Tras dicho estudio teórico, se han detallado todos los dispositivos hardware que han formado parte del actual Trabajo Fin de Grado. Haciendo especial hincapié en la placa Arduino UNO, encargada de ejecutar el programa software desarrolado que simula el funcionamiento de una ECU. También se han descrito el módulo que dota a la placa Arduino UNO de conectividad CAN (CAN-BUS Shield), y la interfaz ELM327 que posibilita realizar el diagnostico OBD-II desde un PC comercial.

Se ha introducido el entorno de programación de Arduino, mediante el cual se ha llevado a cabo la programación de la placa Arduino UNO. Después de conocer dicho entorno, se ha descrito el funcionamiento del programa software desarrollado en este proyecto y que es el encargado de simular una ECU OBD-II en dicha placa.

Por último se ha introducido el programa de diagnóstico ScanMaster utilizado en este Trabajo Fin de Grado para obtener los resultados finales y demostrar que es posible simular una ECU OBD-II de bajo coste y con funcionabilidades similares a las de una ECU real.

5.2 Propuesta de mejora

En primer lugar se propone mejorar la ECU, añadiendo sensores adicionales, dotándola de funcionabilidad en todos los modos OBD-II, desarrollando PIDs propios y DTCs que sean más útiles para el vehículo eléctrico en el que se pretende implantarla. Para mejorar el sistema de detección de errores se tendrían que añadir distintos módulos que controlen los sensores encargados de detectar errores. Por ejemplo, un módulo para los fallos relacionados con el motor, otro para el confort y chasis, y un último para la red de comunicaciones. Cada uno de ellos informaría a la ECU central de un error detectado mediante una trama CAN. El error se registraría y notificaría al conductor.

Aunque existen muchos programas de software gratuito y totalmente configurables para el diagnóstico OBD-II (como el usado en este proyecto: ScanMaster), se podría optar por

desarrollar un software propio, que sea capaz de entender y representar los datos OBD-II procedentes de la ECU, ya sea conectándolo directamente al bus CAN (Tablet PC con conectividad CAN) o haciendo uso de una interfaz como el ELM327.

Otra mejora posible consistiría en dotar de conectividad inalámbrica al sistema OBD-II. Con ello se posibilita el diagnostico OBD-II desde dispositivos móviles y Tablets, en este caso al igual que antes se podría hacer uso de un software libre o desarrollar uno propio.

El objetivo final sería tener un sistema OBD-II completo que permita conocer al conductor todos los datos de relevancia del vehículo en tiempo real, y de realizar pruebas de diagnóstico siempre que le sea necesario.

5.3 **Futuro de OBD**

El importante avance que experimentan las tecnologías de la Información y las Comunicaciones (TICs), ha hecho posible la creación de un nuevo concepto en la diagnosis, la tercera generación de OBD. El OBD-III es un sistema que permite reducir el tiempo entre la detección de un fallo o funcionamiento defectuoso, y la posterior reparación del mismo. Es por tanto una evolución del OBD-II, pues deberá recibir la información que se transmita sobre OBD-II, y después de interpretarla, enviar conclusiones al conductor y talleres de reparación, con las claves de los fallos detectados. Además deberá de crear un histórico en un centro de datos, para ser utilizados en la diagnosis del propio vehículo, y en futuros diseños.

En el OBD-III, los fallos del vehículo son enviados vía radio terrestre o satélite, a un Centro de Atención al Cliente que los detecta y analiza. Este centro notificará al cliente el proceso a seguir para subsanar el problema. Además, también se notificará al taller de la información necesaria para acometer la reparación y las pruebas necesarias para verificar el posterior buen funcionamiento. Todo esto deja claro porqué el OBD-III, se empieza a conocer como diagnosis remota. Lo más importante de todo esto, es la capacidad de comunicación del vehículo con el mundo exterior, a corta y a larga distancia. En esta dirección, se están proponiendo diferentes alternativas para permitir leer los datos almacenados por OBD-II, y enviarlos a centros de Atención al Cliente, Centros de Datos, Servicios Móviles de Mantenimiento.

Bibliografía

[1] **Roy Cox**. (2006). *Introduction to On-Board Diagnostics II (OBDII).*

[2] **Al Santini.** (2011). *OBD-II: Functions, Monitors and Diagnostic Techniques.*

[3] **Diego Q. Aranda.** (Agosto de 2013). *Electrónica del automóvil: Conceptos y fundamentos sobre tecnología electrónica aplicada al sistema automotriz.*

[4] **ISO**. (2011). *ISO 15031-5:2011. Road vehicles -- Communication between vehicle and external equipment for emissions-related diagnostics -- Part 5: Emissions-related diagnostic services.*

[5] **ISO**. (2011). *ISO 15765-4:2011. Road vehicles -- Diagnostic communication over Controller Area Network (DoCAN) -- Part 4: Requirements for emissions-related systems.*

[6] **ISO**. (2003). *ISO 11898-1:2003. Road vehicles -- Controller area network (CAN) -- Part 1: Data link layer and physical signalling.*

[7] **ISO**. (2003). *ISO 11898-2:2003. Road vehicles -- Controller area network (CAN) -- Part 2: High-speed medium access unit.*

[8] **ISO**. (2006). *ISO 11898-3:2006. Road vehicles -- Controller area network (CAN) -- Part 3: Low-speed, fault-tolerant, medium-dependent interface.*

[9] **ISO**. (2012). *ISO 14230-1:2012. Road vehicles -- Diagnostic communication over K-Line (DoK-Line) -- Part 1: Physical layer.*

[10] **ISO**. (2013). *ISO 14230-2:2013. Road vehicles -- Diagnostic communication over K-Line (DoK-Line) -- Part 2: Data link layer.*

[11] **SAE**. (2010). *SAE J1850. Class B Data Communications Network Interface.*

[12] **CANopen.** (s.f.). *Portal web de CANopen.* (Último acceso en Agosto de 2015), http://www.canopensolutions.com/index.html

[13] **Adam Shaw**. (Junio de 2011). *Automotive OBD-II Simulator.*

[14] **Arduino UNO**. (s.f.). *Especificaciones técnicas y documentación de la placa Arduino UNO, web oficial de Arduino.* (Último acceso en Agosto de 2015), https://www.arduino.cc/en/Main/arduinoBoardUno

[15] **ATmega328P**. (2014). *Especificaciones técnicas de la ATmega328P.* (Último acceso en Agosto de 2015), http://www.atmel.com/devices/atmega328p.aspx

[16] **CAN-BUS Shield**. (Julio de 2014). *CAN-BUS Shield wiki.* (Último acceso Septiembre 2015). http://www.seeedstudio.com/wiki/CAN-BUS_Shield

[17] **Microchip**. (2010). *Especificaciones del MCP2515 y del MCP2551.* (Último acceso Agosto de 2015). http://www.microchip.com/

[18] **ELM327**. (s.f.). *Especificaciones técnicas de la interfaz ELM327.* (Último acceso Agosto de 2015). http://elmelectronics.com/

[19] **AutoSim**. (s.f.). *Portal web de AutoSim.* (Último acceso Agosto de 2015). http://www.turbofast.com.au/AutoSim/index.html

[20] **Rafael Enríquez Herrador**. (Noviembre de 2009). *Guía de Usuario Arduino.*

[21] **Arduino IDE**. (s.f.). *Guía de instalación en Windows.* (Último acceso Agosto de 2015) https://www.arduino.cc/en/Guide/HomePage

[22] **Lenguaje Arduino**. (s.f.). *Lenguaje de programación Arduino.* (Último acceso Agosto de 2015). https://www.arduino.cc/en/Reference/HomePage

[23] **OBD DTCs**. (s.f.). *Listado de DTCs.* (Último acceso Septiembre de 2015). http://www.obd-codes.com/trouble_codes/

[24] **ScanMaster-ELM**. (s.f.). *Web oficial de ScanMaster-ELM.* (Último acceso Agosto de 2015). https://www.scantool.net/software/scanmaster.html

Anexo A

Este anexo muestra el código desarrollado para simular la ECU OBD-II en la placa Arduino UNO, no se muestran las librerías SPI.h y mcp_can.h pues son código abierto y se encuentran disponibles en la web de Arduino y Seeed respectvamente[2].

CAN-OBDII.ino

```
// CAN-OBDII.ino
// Simula una ECU respondiendo con mensajes OBD-II sobre CAN a las peticiones del software de
//escáner OBD-II.
// Puede trabajar en Modo 1: Flujo de Datos en Vivo, Modo 3: Leer Códigos de Falla DTC y Modo 4:
// Borrar Fallos.
// Autor: José Beltrán Zambrano, Fecha: 02/08/15

// Incluimos las librerías necesarias:
// SPI.h: permite la comunicación entre Arduino-Uno y CAN-BUS Shield.
// mcp_can.h: librería que el Seeed Studio facilita para la comunicación CAN haciendo uso del
// CAN-BUS Shield.

#include <SPI.h>
#include "mcp_can.h"

// Constantes

const int SPI_CS_PIN = 9;              // Pin CS
const int LedMIL = 5;                  // Led rojo en el pin 5 para el indicador MIL
const char SpeedPin = A0;              // Entrada de velocidad analógica (potenciómetro 1)
const char BateryPin = A1;             // Entrada de batería analógica (potenciómetro 2)



// Variables

INT32U canId = 0x000;                  // Identificador CAN
```

[2] Estas librerías se pueden encontrar en los siguientes enlaces:
https://github.com/arduino/Arduino/tree/1.6.5-r5/libraries
https://github.com/Seeed-Studio/CAN_BUS_Shield

```
unsigned char len = 0;                  // Longitud de los datos recibidos
unsigned char buf[8];                   // Buffer de almacenamiento del mensaje OBD-II
String BuildMessage="";                 // Cadena para imprimir el mensaje recibido
int MODE = 0;                           // Modo de funcionamiento OBD-II
int PID = 0;                            // Parameters IDs del Modo 1
int MILflag = 0;                        // Bandera: 0=MIL Apagado / 1=MIL Encendido
int DTCflag = 0;                        // Bandera que indica que hay al menos un DTC en memoria
unsigned long TimeMIL = 0;              // Marca de tiempo en la que se encendió indicador MIL
float MetersMIL = 0;                    // Metros recorridos mientras MIL esta encendido
unsigned long Tans = 0;                 // Marca de tiempo se usa para contar un segundo

// Definimos el pin CS para la comunicación entre Arduino-Uno y CAN-BUS Shield

MCP_CAN CAN(SPI_CS_PIN);

// Funciones

// int ReadSpeed()
// Entradas: ninguna /*/ Salidas: Speed
// Función para leer la velocidad según la posición del potenciómetro 1

int ReadSpeed(){
  // Lee la entrada analógica del PIN A0
  int sensorValue = analogRead(SpeedPin);
  // Convierte el valor leído entre 0-1023 en un valor entre 0-150
  int Speed = sensorValue * (150.0 / 1023.0);

  return Speed;
}

// int ReadRPM()
// Entradas: ninguna /*/ Salidas: RPM
// Función para leer las rpm según la posición del potenciómetro 1

int ReadRPM(){
  // Lee la entrada analógica del PIN A0
  int sensorValue = analogRead(SpeedPin);
  // Convierte el valor leído entre 0-1023 en un valor entre 0-4000
```

```
  int RPM = sensorValue * (4000.0 / 1023.0);

  return RPM;
}

// int ReadThrottle()
// Entradas: ninguna /*/ Salidas: TPosition
// Función para leer la posición del acelerador según la posición del potenciómetro 1

int ReadThrottle(){
  // Lee la entrada analogica del PIN A0
  int sensorValue = analogRead(SpeedPin);
  // Convierte el valor leído entre 0-1023 en un valor entre 0-255
  int TPosition = sensorValue * (255.0 / 1023.0);

  return TPosition;
}

// int ReadBateryLvl()
// Entradas: ninguna /*/ Salidas: Level
// Función para leer el nivel de batería según la posición del potenciómetro 2

int ReadBateryLvl(){
  // Lee la entrada analógica del PIN A1
  int sensorValue = analogRead(BateryPin);
  // Convierte el valor leído entre 0-1023 en un valor entre 0-255
  int Level = sensorValue * (255.0 / 1023.0);

  return Level;
}

// void CalMeters()
// Entradas: ninguna /*/ Salidas: ninguna
// Calcula los metros recorridos por segundo mientras que MIL esta encendida
// los metros se van almacenando en la variable golbal MetersMIL

void CalMeters(){
```

```
  // Si se acaba de encender el indicador MIL empezamos a contar segundos
  if(Tans == 0){
     Tans = millis();
    }
    else{
     // Vamos acumulando metros con el indicador encendido
     // Pasamos los km/h a metros/segundo y se suman cada segundo transcurrido
     int minCicle = (millis()-Tans);
     if (minCicle >= 1000){                                    // Si ha trascurrido 1 segundo
       // Calculamos los metros recorridos por segundo y sumamos
       MetersMIL = MetersMIL + (ReadSpeed()*10/36);
       Tans = millis();                                        // Actualizamos Tans
     }
    }
}

// String ReadMessage()
// Entradas: ninguna /*/ Salidas: Message
// Función para leer los datos OBD-II dentro de la trama CAN, devuelve la cadena leída
// Además si el mensaje es una petición valida guarda el Modo y el PID requerido

String ReadMessage(){

  String Message="";
  CAN.readMsgBuf(&len, buf);       // Almacenamos el mensaje OBD-II en buf y su longitud en len

  // Imprime la dirección Id
  canId = CAN.getCanId();                                  // Identificador del mensaje
  Serial.print("<");Serial.print(canId);Serial.print(",");

  // Construye el mensaje y lo imprime
  for(int i = 0; i<len; i++)
  {
    if( i == len-1 ){
      Message = Message + buf[i];
    }
    else {
      Message = Message + buf[i] + ",";
```

```
   }
  }
  Serial.println(Message);

  // Guardamos el modo y el PID requerido
  if ( buf[0] != 0 ){                              // Comprobamos que el mensaje no está vacío
    MODE = buf[1];
    Serial.print("Modo: ");Serial.println(MODE);
    // Si trabaja en modo uno guardamos el PID
    if (MODE == 1){
      PID = buf[2];
      Serial.print("PID: ");Serial.println(PID);
    }
  }
  return Message;
}

// void ReplyMode01()
// Entradas: ninguna /*/ Salidas: ninguna
// Función para responder a una petición OBD-II en Modo 1

void ReplyMode01()
{
 // Mensaje OBD-II predefinido para responder en el modo 1
 // {len datos, 01+40hex, PID, DatoA, DatoB, DatoC, DatoD}
 unsigned char OBDIImsg[8] = {4, 65, PID, 0, 0, 0, 0, 0};          // Inicializamos len a 4
 // Datos = 85 o 0 en el caso de que no se usen
 int A=0;                                                          // Variables para cálculos
 int B=0;                                                          // Variables para cálculos
  switch (PID){
     case 0:                                                       // PID 00 hex
       // La longitud util en este caso es 6
       OBDIImsg[0] = 6;
       // PIDs soportados del 01-20 hex
       OBDIImsg[3] = 0x88;
       OBDIImsg[4] = 0x18;
       OBDIImsg[5] = 0x80;
       OBDIImsg[6] = 0x13;
```

```
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);                    // Respuesta
  break;
 case 1:                                                    // PID 01 hex
  // MIL encendido si DTCflag = 1
  if (DTCflag == 1){
  OBDIImsg[3] = 129;                                // Luz MIL ON y Numero de fallos detectados
  }
  // MIL apagado, 0 DTCs almacenados, ningun test posible
  //DatoA,DatoB,DatoC,DatoD = 0;
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);          // Respuesta
  break;
 case 5:                                                    // PID 05 hex
  // Temperatura del refrigentante
  OBDIImsg[3] = random(65,70);               // Como no lo conocemos ponemos un valor aleatorio
  // Construimos la respuesta
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
 case 12:                                                   // PID 0C hex
  // Calculo de las RPM ((A*256)+B)/4
  A = ReadRPM();
  OBDIImsg[3] = A*4/256;
  OBDIImsg[4] = A - (A*4/256);
  // Construimos la respuesta
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
 case 13:                                                   // PID 0D hex
  // Calculo de la velocidad
  OBDIImsg[3] = ReadSpeed();
  // Construimos la respuesta
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
 case 17:                                                   // PID 11 hex
  // Calculo de la posición del pedal del acelerador
  OBDIImsg[3] = ReadThrottle();
  // Construimos la respuesta
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
 case 28:                                                   // PID 1C hex
```

```
  // Estandar OBD-II utilizado
  OBDIImsg[3] = 3;                          // El valor 3 indica compatible con OBD y OBD-II
  // Construimos la respuesta
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
case 31:                                    // PID 19 hex
  // Tiempo en segundos desde el arranque
  // La función millis mide el tiempo en ms desde que el programa arrancó
  // Valor entre 0 y 65535s formula: (A*256)+B
  A = millis()/1000;
  OBDIImsg[3] = A/256;
  OBDIImsg[4] = A - A/256;
  // Construimos la respuesta
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
case 32:                                    // PID 20 hex
  // La longitud útil en este caso es 6
  OBDIImsg[0] = 6;
  // PIDs soportados del 21-40 hex
  OBDIImsg[3] = 0x80;
  OBDIImsg[4] = 0x02;
  OBDIImsg[5] = 0x00;
  OBDIImsg[6] = 0x01;
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
case 33:                                    // PID 21 hex
  // km recorridos desde que el indicador MIL esta encendido
   A = MetersMIL/1000;
  // Entre 0 y 65535 formula: (A*256)+B
  OBDIImsg[3] = A/256;
  OBDIImsg[4] = A - A/256;
  // Construimos la respuesta
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
  break;
case 47:                                    // PID 2F hex
  // Nivel de la batería
  OBDIImsg[3] = ReadBateryLvl();
  // Construimos la respuesta
```

```
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
    break;
  case 64:                                          // PID 40 hex
    // La longitud util en este caso es 6
    OBDIImsg[0] = 6;
    // PIDs soportados del 41-60 hex
    OBDIImsg[3] = 0x04;
    OBDIImsg[4] = 0x08;
    OBDIImsg[5] = 0x80;
    OBDIImsg[6] = 0x00;
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
    break;
  case 70:                                          // PID 46 hex
    // Temperatura ambiente A-40
    OBDIImsg[3] = random(70,75);              // Como no lo conocemos ponemos un valor aleatorio
    // Construimos la respuesta
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
    break;
  case 77:                                          // PID 4D hex
    // Tiempo desde que el indicador MIL esta encendido
    // Valor entre 0 y 65535s formula: (A*256)+B
    if (TimeMIL != 0){
      A = (millis() - TimeMIL)/60000;               // Tiempo en minutos
    }
    OBDIImsg[3] = A/256;
    OBDIImsg[4] = A - A/256;
    // Construimos la respuesta
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
    break;
  case 81:                                          // PID 51 hex
    // Indica el tipo de fuel: electrico 8
    OBDIImsg[3] = 8;
    // Construimos la respuesta
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
    break;

}
```

```
}

// ReplyMode03
// Entradas: ninguna /*/ Salidas: ninguna
// Función para responder a una petición OBD-II en Modo 3

void ReplyMode03(){
  // Mensaje OBD-II predefinido para responder en el modo 1
  // {len datos, 03+40hex, num DTCs, DTC1H, DTC1L, DTC2H, DTC2L}
  unsigned char OBDIImsg[8] = {6, 67, 0, 0, 0, 0, 0, 0};            // Inicializamos a 0 no hay DTCs

  // Se ha detectado un sobrecalentamiento del motor
  if(DTCflag == 1){
    OBDIImsg[2] = 1;                                    // Número de DTCs = 1
    OBDIImsg[3] = 2;                                    // P0217 Sobrecalentamiento del motor HIGH
    OBDIImsg[4] = 23;                                   // P0217 Sobrecalentamiento del motor LOW
  }
  // Si no hay errores se envía el mensaje inicial
  CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
}

// void InterrupDTC1()
// Función que atiende a la interrupción 1

void InterrupDTC1() {
  DTCflag = 1;
  TimeMIL = millis();
}

void setup()
{
  Serial.begin(115200);                                 // Inicio de la comunicación serie

  // Definición de los pines

  pinMode(LedMIL, OUTPUT);                              // Pin LedMIL configurado como salida

  // Definimos las interrupciones
```

```
 attachInterrupt( 1, InterrupDTC1, RISING);                // Sera atendida en flancos de subida

 // Iniciamos el bus CAN a 500 Kbps

 START_INIT:

  // Si se inicia correctamente continuamos

  if(CAN_OK == CAN.begin(CAN_500KBPS))
  {
     Serial.println("El BUS CAN se ha iniciado correctamente");
  }

  // De lo contrario reintentamos el inicio del bus

  else

     Serial.println("Error en el inicio del BUS CAN");
     Serial.println("Iniciando el BUS CAN de nuevo");
     delay(100);
     goto START_INIT;
  }

}


void loop()
{
 // Si recibe alguna trama CAN
 if(CAN_MSGAVAIL == CAN.checkReceive())
 {
  MODE = 0;
  PID = 0;
  // Lee el mensaje
  BuildMessage = ReadMessage();

  // En que modo estamos trabajando
```

```
  switch (MODE){
    case 1:                          // Si estamos trabajando en el modo 01 -> datos actuales
      ReplyMode01();                 // Llamamos a la función encargada de responder
      break;

    case 3:                          // Si estamos trabajando en el modo 03 -> DTCs
      ReplyMode03();                 // Llamamos a la función encargada de responder
      break;

    case 4:                          // Si estamos trabajando en el modo 04 -> limpiar DTCs
      DTCflag = 0;                   // Bajamos la bandera
      TimeMIL = 0;                   // Reiniciamos el tiempo con MIL
      MetersMIL = 0;                 // Reiniciamos los metros recorridos con MIL
      Tans = 0;                      // Reiniciamos el tiempo para calcular los metros
      break;
  }

  BuildMessage="";
 }
 // Comprobamos el estado del indicador MIL
 if( DTCflag == 1 ){
  digitalWrite(LedMIL, HIGH);
  // Calculamos la distancia recorrida mientras MIL encendido
  CalMeters();
 }
 else{
  digitalWrite(LedMIL, LOW);
 }
}
```

Anexo B

El presente anexo muestra todos los PIDs que contempla el estándar OBD-II.

Mode (hex)	PID (hex)	Data bytes	Description	Min value	Max value	Units	Formula
01	00	4	PIDs supported [01 - 20]				Bit encoded [A7...D0] == [PID 0x01...PID 0x20]
01	01	4	Monitor status since DTCs cleared. (Includes malfunction indicator lamp (MIL) status and number of DTCs.)				Bit encoded.
01	02	8	Freeze				
01	03	2	Fuel system status				Bit encoded.
01	04	1	Calculated engine load value	0	100	%	A*100/255
01	05	1	Engine coolant temperature	-40	215	°C	A-40
01	06	1	Short term fuel % trim—Bank 1	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
01	07	1	Long term fuel % trim—Bank 1	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
01	08	1	Short term fuel % trim—Bank 2	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
01	09	1	Long term fuel % trim—Bank 2	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
01	0A	1	Fuel pressure	0	765	kPa (gauge)	A*3
01	0B	1	Intake manifold absolute pressure	0	255	kPa (absolute)	A
01	0C	2	Engine RPM	0	16,383.75	rpm	((A*256)+B)/4
01	0D	1	Vehicle speed	0	255	km/h	A
01	0E	1	Timing advance	-64	63.5	° relative to #1 cylinder	A/2 – 64
01	0F	1	Intake air temperature	-40	215	°C	A-40
01	10	2	MAF air flow rate	0	655.35	g/s	((A*256)+B) / 100
01	11	1	Throttle position	0	100	%	A*100/255
01	12	1	Commanded secondary air status				Bit encoded.
01	13	1	Oxygen sensors present				[A0...A3] == Bank 1, Sensors 1-4. [A4...A7] == Bank 2…
01	14	2	Bank 1, Sensor 1:Oxygen sensor voltage, Short term fuel trim	0-100 (lean)	1.27599.2 (rich)	Volts%	A * 0.005(B-128) * 100/128 (if B==0xFF, sensor is not used in trim calc)
01	15	2	Bank 1, Sensor 2:Oxygen sensor voltage, Short term fuel trim	0-100 (lean)	1.27599.2 (rich)	Volts%	A * 0.005(B-128) * 100/128 (if B==0xFF, sensor is not used in trim calc)
01	16	2	Bank 1, Sensor 3:Oxygen	0-100	1.27599.2	Volts%	A * 0.005(B-128) *

			sensor voltage, Short term fuel trim	(lean)	(rich)		100/128 (if B==0xFF, sensor is not used in trim calc)
01	17	2	Bank 1, Sensor 4:Oxygen sensor voltage, Short term fuel trim	0-100 (lean)	1.27599.2 (rich)	Volts%	A * 0.005(B-128) * 100/128 (if B==0xFF, sensor is not used in trim calc)
01	18	2	Bank 2, Sensor 1:Oxygen sensor voltage, Short term fuel trim	0-100 (lean)	1.27599.2 (rich)	Volts%	A * 0.005(B-128) * 100/128 (if B==0xFF, sensor is not used in trim calc)
01	19	2	Bank 2, Sensor 2:Oxygen sensor voltage, Short term fuel trim	0-100 (lean)	1.27599.2 (rich)	Volts%	A * 0.005(B-128) * 100/128 (if B==0xFF, sensor is not used in trim calc)
01	1A	2	Bank 2, Sensor 3:Oxygen sensor voltage, Short term fuel trim	0-100 (lean)	1.27599.2 (rich)	Volts%	A * 0.005(B-128) * 100/128 (if B==0xFF, sensor is not used in trim calc)
01	1B	2	Bank 2, Sensor 4:Oxygen sensor voltage, Short term fuel trim	0-100 (lean)	1.27599.2 (rich)	Volts%	A * 0.005(B-128) * 100/128 (if B==0xFF, sensor is not used in trim calc)
01	1C	1	OBD standards this vehicle conforms to				Bit encoded.
01	1D	1	Oxygen sensors present				Similar to PID 13, but [A0...A7] == [B1S1, B1S2, B2S1, B2S2, B3S1, B3S2, B4S1, B4S2]
01	1E	1	Auxiliary input status				A0 == Power Take Off (PTO) status (1 == active) [A1...A7] not used
01	1F	2	Run time since engine start	0	65,535	seconds	(A*256)+B
01	20	4	PIDs supported 21-40				Bit encoded [A7...D0] == [PID 0x21...PID 0x40]
01	21	2	Distance traveled with malfunction indicator lamp (MIL) on	0	65,535	km	(A*256)+B
01	22	2	Fuel Rail Pressure (relative to manifold vacuum)	0	5177.265	kPa	(((A*256)+B) * 10) / 128
01	23	2	Fuel Rail Pressure (diesel)	0	655350	kPa (gauge)	((A*256)+B) * 10
01	24	4	O2S1_WR_lambda(1): Equivalence Ratio Voltage	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192
01	25	4	O2S2_WR_lambda(1): Equivalence Ratio Voltage	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192
01	26	4	O2S3_WR_lambda(1): Equivalence Ratio Voltage	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192
01	27	4	O2S4_WR_lambda(1):	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192

			Equivalence Ratio Voltage				
01	28	4	O2S5_WR_lambda(1): Equivalence Ratio Voltage	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192
01	29	4	O2S6_WR_lambda(1): Equivalence Ratio Voltage	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192
01	2A	4	O2S7_WR_lambda(1): Equivalence Ratio Voltage	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192
01	2B	4	O2S8_WR_lambda(1): Equivalence Ratio Voltage	00	28	N/AV	((A*256)+B)/32768((C*256)+D)/8192
01	2C	1	Commanded EGR	0	100	%	100*A/255
01	2D	1	EGR Error	-100	99.22	%	(A-128) * 100/128
01	2E	1	Commanded evaporative purge	0	100	%	100*A/255
01	2F	1	Fuel Level Input	0	100	%	100*A/255
01	30	1	# of warm-ups since codes cleared	0	255	N/A	A
01	31	2	Distance traveled since codes cleared	0	65,535	km	(A*256)+B
01	32	2	Evap. System Vapor Pressure	-8,192	8,192	Pa	((A*256)+B)/4 (A is signed)
01	33	1	Barometric pressure	0	255	kPa (Absolute)	A
01	34	4	O2S1_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/32768((C*256)+D)/256 – 128
01	35	4	O2S2_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/32768((C*256)+D)/256 – 128
01	36	4	O2S3_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/327685((C*256)+D)/256 – 128
01	37	4	O2S4_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/32768((C*256)+D)/256 – 128
01	38	4	O2S5_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/32768((C*256)+D)/256 – 128
01	39	4	O2S6_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/32768((C*256)+D)/256 – 128
01	3A	4	O2S7_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/32768((C*256)+D)/256 – 128
01	3B	4	O2S8_WR_lambda(1): Equivalence Ratio Current	0-128	2128	N/AmA	((A*256)+B)/32768((C*256)+D)/256 – 128
01	3C	2	Catalyst TemperatureBank 1, Sensor 1	-40	6,513.5	°C	((A*256)+B)/10 – 40
01	3D	2	Catalyst TemperatureBank 2, Sensor 1	-40	6,513.5	°C	((A*256)+B)/10 – 40
01	3E	2	Catalyst TemperatureBank 1,	-40	6,513.5	°C	((A*256)+B)/10 – 40

			Sensor 2				
01	3F	2	Catalyst TemperatureBank 2, Sensor 2	-40	6,513.5	°C	((A*256)+B)/10 – 40
01	40	4	PIDs supported 41-60				Bit encoded [A7...D0] == [PID 0x41...PID 0x60]
01	41	4	Monitor status this drive cycle				Bit encoded.
01	42	2	Control module voltage	0	65.535	V	((A*256)+B)/1000
01	43	2	Absolute load value	0	25,700	%	((A*256)+B)*100/255
01	44	2	Command equivalence ratio	0	2	N/A	((A*256)+B)/32768
01	45	1	Relative throttle position	0	100	%	A*100/255
01	46	1	Ambient air temperature	-40	215	°C	A-40
01	47	1	Absolute throttle position B	0	100	%	A*100/255
01	48	1	Absolute throttle position C	0	100	%	A*100/255
01	49	1	Accelerator pedal position D	0	100	%	A*100/255
01	4A	1	Accelerator pedal position E	0	100	%	A*100/255
01	4B	1	Accelerator pedal position F	0	100	%	A*100/255
01	4C	1	Commanded throttle actuator	0	100	%	A*100/255
01	4D	2	Time run with MIL on	0	65,535	minutes	(A*256)+B
01	4E	2	Time since trouble codes cleared	0	65,535	minutes	(A*256)+B
01	51	1	Fuel Type				From fuel type table
01	52	1	Ethanol fuel %	0	100	%	A*100/255
01	53	2	Absoulute Evap system Vapour Pressure	0	327675	Kpa	1/200 per bit
01	C3	?	?	?	?	?	Returns numerous data, including Drive Condition ID and Engine Speed*
01	C4	?	?	?	?	?	B5 is Engine Idle RequestB6 is Engine Stop Request*
02	02	2	Freeze frame trouble code				BCD encoded.
03	N/A	n*6	Request trouble codes				3 codes per message frame, BCD encoded.
04	N/A	0	Clear trouble codes / Malfunction indicator lamp (MIL) / Check engine light				Clears all stored trouble codes and turns the MIL off.
05	0100		OBD Monitor IDs supported ($01 – $20)				
05	0101		O2 Sensor Monitor Bank 1 Sensor 1	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0102		O2 Sensor Monitor Bank 1 Sensor 2	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0103		O2 Sensor Monitor Bank 1 Sensor 3	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0104		O2 Sensor Monitor Bank 1 Sensor 4	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0105		O2 Sensor Monitor Bank 2 Sensor 1	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0106		O2 Sensor Monitor Bank 2	0.00	1.275	Volts	0.005 Rich to lean

			Sensor 2				sensor threshold voltage
05	0107		O2 Sensor Monitor Bank 2 Sensor 3	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0108		O2 Sensor Monitor Bank 2 Sensor 4	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0109		O2 Sensor Monitor Bank 3 Sensor 1	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	010A		O2 Sensor Monitor Bank 3 Sensor 2	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	010B		O2 Sensor Monitor Bank 3 Sensor 3	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	010C		O2 Sensor Monitor Bank 3 Sensor 4	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	010D		O2 Sensor Monitor Bank 4 Sensor 1	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	010E		O2 Sensor Monitor Bank 4 Sensor 2	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	010F		O2 Sensor Monitor Bank 4 Sensor 3	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0110		O2 Sensor Monitor Bank 4 Sensor 4	0.00	1.275	Volts	0.005 Rich to lean sensor threshold voltage
05	0201		O2 Sensor Monitor Bank 1 Sensor 1	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0202		O2 Sensor Monitor Bank 1 Sensor 2	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0203		O2 Sensor Monitor Bank 1 Sensor 3	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0204		O2 Sensor Monitor Bank 1 Sensor 4	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0205		O2 Sensor Monitor Bank 2 Sensor 1	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0206		O2 Sensor Monitor Bank 2 Sensor 2	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0207		O2 Sensor Monitor Bank 2 Sensor 3	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0208		O2 Sensor Monitor Bank 2 Sensor 4	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0209		O2 Sensor Monitor Bank 3	0.00	1.275	Volts	0.005 Lean to Rich

			Sensor 1				sensor threshold voltage
05	020A		O2 Sensor Monitor Bank 3 Sensor 2	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	020B		O2 Sensor Monitor Bank 3 Sensor 3	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	020C		O2 Sensor Monitor Bank 3 Sensor 4	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	020D		O2 Sensor Monitor Bank 4 Sensor 1	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	020E		O2 Sensor Monitor Bank 4 Sensor 2	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	020F		O2 Sensor Monitor Bank 4 Sensor 3	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
05	0210		O2 Sensor Monitor Bank 4 Sensor 4	0.00	1.275	Volts	0.005 Lean to Rich sensor threshold voltage
09	00	4	Mode 9 supported PIDs 01 to 20				Bit encoded
09	02	5×5	Vehicle identification number (VIN)				Returns 5 lines, A is line ordering flag, B-E ASCII coded VIN digits.
09	04	varies	Calibration ID				Returns multiple lines, ASCII coded
09	06	4	Calibration				