Grado en Ingeniería de Tecnologías de Telecomunicación

Trabajo Fin de Grado

"Simulador de una ECU y diagnóstico mediante CAN y OBD-II"

Jorge Sánchez Carrizo
Cuenca, Agosto, 2017
AGRADECIMIENTOS

En primer lugar, quiero dar las gracias a todos aquellos que han hecho posible que hoy en día esté estudiando esta carrera, mi familia. Sin ellos no hubiera llegado tan lejos en la vida y no sería la persona que hoy soy. Pero especialmente quiero agradecer a mi hermana el que me haya dado todo su apoyo en todo momento.

En segundo lugar, también agradecer a mis amigos y amigas, compañeros y compañeras del Grado en Ingeniería de Tecnologías de Telecomunicación (GITT), que me han acompañado en todo momento.

Además, agradecer a todos los profesores que me han impartido clase a lo largo de estos años por todos los conocimientos que me han enseñado.

Y especialmente, quiero dar las gracias a mis directores César Sánchez Meléndez y Raúl Alcaraz Martínez por su gran ayuda, dedicación y por darme la oportunidad de tenerles en este gran proyecto.
INDICE

RESUMEN ................................................................................................................................. 1

ABSTRACT ............................................................................................................................... 3

PARTE I: MEMORIA .................................................................................................................. 5

INTRODUCCIÓN ....................................................................................................................... 6

  Motivaciones ......................................................................................................................... 6
  Objetivos ............................................................................................................................... 6

FUNDAMENTOS TEÓRICOS: OBDII ..................................................................................... 7

  Historia e introducción del OBD .......................................................................................... 7
  Funcionamiento OBD II ....................................................................................................... 9
  Controles realizados por OBD II .......................................................................................... 10
    Control Contaminación ...................................................................................................... 10
    Control Motores Gasolina ................................................................................................. 11
    Control Motores Diesel ..................................................................................................... 12
  Modos de mediación OBD II ............................................................................................... 12
    Modo 01: Obtención de datos de diagnóstico actualizados .............................................. 13
    Modo 02: Acceso al cuadro de los datos congelados ......................................................... 15
    Modo 03: Códigos de diagnóstico almacenados ............................................................... 16
    Modo 04: Borrado de códigos de falla (DTC) y cuadro de almacenados ............................ 18
    Modo 05: Test de los sensores de oxígeno ........................................................................ 19
    Modo 06: Resultado de pruebas de control de otros transductores .................................. 20
    Modo 07: Mostrar código de diagnóstico pendiente ......................................................... 21
    Modo 08: Control de funcionamiento de componentes (C. de a bordo) ............................. 21
    Modo 09: Información del vehículo .................................................................................... 21

PROTOCOLOS PARA LA COMUNICACIÓN OBD II ............................................................ 22

  SAE J1850 PWM y VPW ..................................................................................................... 25
    Capa física SAE J1850 ....................................................................................................... 27
    Capa de enlace SAE J1850 ............................................................................................... 28
  ISO 9141-2 .......................................................................................................................... 28
  ISO 14230-4 (KWP2000) ...................................................................................................... 30
  CAN (11898/15765) .............................................................................................................. 31
    Características y ventajas del CAN .................................................................................... 33
    Topología en una red CAN ............................................................................................... 34
    Capas OSI en protocolo CAN: CAPA FÍSICA ................................................................. 35
    Capas OSI en protocolo CAN: CAPA de Enlace ............................................................... 37
    Capas OSI en protocolo CAN: CAPAS Intermedias y la de Aplicación ............................ 40
    Temporización en un CAN-Bus ........................................................................................ 41
    Resincronismo en un CAN-Bus ....................................................................................... 41

DESARROLLO DEL PROYECTO: HARDWARE ..................................................................... 43

  Sistema Arduino ..................................................................................................................... 44
    Arduino UNO ..................................................................................................................... 48
    Placa CAN-BUS Shield para Arduino ................................................................................ 51
  Dispositivo de diagnóstico .................................................................................................... 54
    ELM 327 ............................................................................................................................. 54
    Ordenador SONY VAIO .................................................................................................... 55
  Circuito y montaje ................................................................................................................ 56
INDICE FIGURAS

FIGURA 1. COMPARATIVA TERMINALES OBD.................................................................8
FIGURA 2. EJEMPLO OBD-II .........................................................................................9
FIGURA 3. ECU .............................................................................................................10
FIGURA 4. CONECTOR SERIE RS232C .......................................................................11
FIGURA 5. EJEMPLO DE DTC ....................................................................................17
FIGURA 6. EJEMPLO MONITOREADO OXÍGENO .....................................................20
FIGURA 7. CONTENIDO DE OXÍGENO FIGURA 8. MEZCLA DE GASES .....................20
FIGURA 9. CAPAS MODELO OSI................................................................................24
FIGURA 10. MODULACIÓN PWM ..............................................................................27
FIGURA 11. MODULACIÓN VPW ...............................................................................27
FIGURA 12. TRAMA SAE J1850 ................................................................................28
FIGURA 13. TRAMAS DEL PROTOCOLO ISO 9141-2 ..............................................29
FIGURA 14. SISTEMA CAN BUS EN UN VEHÍCULO ..............................................32
FIGURA 15. TOPOLOGÍA DE UNA RED CAN............................................................34
FIGURAS 16 Y 17. EJEMPLO COMPORTAMIENTO NRZ ........................................35
FIGURA 18. CODIFICACIÓN EN NRZ A ALTA VELOCIDAD ....................................36
FIGURA 19. CODIFICACIÓN EN NRZ A BAJA VELOCIDAD ....................................36
FIGURA 20. TRAMA CAPA DE ENLACE DEL CAN ................................................38
FIGURA 21. EJEMPLO TRAMA ..................................................................................39
FIGURA 22. RESINCRONISMO ................................................................................41
FIGURA 23. ESQUEMA ARDUINO UNO .....................................................................48
FIGURA 24. ATMega328P .........................................................................................49
FIGURA 25. CONEXIONES ATMega328P ................................................................50
FIGURA 26. CAN BUS SHIELD V1.2.........................................................................51
FIGURA 27. PATILLAS CAN BUS SHIELD V1.2 .......................................................52
FIGURA 28. ESQUEMA CAN BUS SHIELD V1.2 .....................................................52
FIGURA 31. ELM 327 .................................................................................................54
FIGURA32. MAPA PINES ELM 327 ..........................................................................55
FIGURA 33. ORDENADOR SONY VAIO ..................................................................55
FIGURA 34. CIRCUITO SIMULADOR ECU ...............................................................56
FIGURA 35. CIRCUITO SIMULADOR ECU 2 ............................................................57
FIGURA 36. ESQUEMA SIMULADOR ECU ................................................................57
INDICE TABLAS

TABLA 1. MODO 01 PETICIONES OBD-II.................................................................13
TABLA 2. MODO 01 RESPUESTAS OBD-II ..............................................................14
TABLA 3: PRINCIPALES PIDS ESTÁNDAR OBD-II ..................................................15
TABLA 4: PETICIONES ESTÁNDAR OBD-II EN MODO 02 ......................................16
TABLA 5: RESPUESTAS ESTÁNDAR OBD-II EN MODO 02 ......................................16
TABLA 6: PETICIONES ESTÁNDAR OBD-II EN MODO 03 ........................................18
TABLA 7: RESPUESTAS ESTÁNDAR OBD-II EN MODO 03 ........................................18
TABLA 8: CODIFICACIÓN DTC ..............................................................................18
TABLA 9: PETICIONES ESTÁNDAR OBD-II EN MODO 04 ......................................18
TABLA 10: RESPUESTAS ESTÁNDAR OBD-II EN MODO 04 ....................................19
TABLA 11: RESPUESTAS NEGATIVAS ESTÁNDAR OBD-II EN MODO 04 ..............19
TABLA 12: PROTOCOLOS ESTÁNDAR OBD-II MODELO OSI ................................24
TABLA 13: CARACTERÍSTICAS SAE J1850 PWM .................................................25
TABLA 14: CARACTERÍSTICAS SAE J1850 VPW ...................................................26
TABLA 15: CARACTERÍSTICAS ISO 9141-2 ...........................................................29
TABLA 16: CARACTERÍSTICAS ISO 14230-4 (KWP2000) ......................................30
TABLA 17: CARACTERÍSTICAS CAN (11898/15765) .............................................32
TABLA 18. PROTOCOLOS CAPA FÍSICA DEL CAN .............................................37
TABLA 19. MODELOS ARDUINO .............................................................................47
TABLA 20. ACCESOS RÁPIDOS EN EL ENTORNO DE ARDUINO .........................63
TABLA 21. PINES DE LAS LÍNEAS SPI .................................................................67
RESUMEN

El dispositivo conocido como OBD (cuyas siglas significan “On Board Diagnostics”), es un sistema de diagnóstico integrado que permite monitorizar y gestionar diversos parámetros de un vehículo, como puede ser el motor, el sistema electrónico de los frenos (ABS), nivel de batería y diversos elementos más. Esto nos permite comprobar el correcto funcionamiento de todos los sistemas y detectar posibles problemas antes de que puedan agravarse. Gracias a este dispositivo también podemos conocer en todo momento la velocidad que mantiene el vehículo, la cantidad de gases que expulsa y por supuesto el nivel de combustible, agua, etc.

La labor desempeñada en el Trabajo Fin de Grado (TFG) está basada en su totalidad en el protocolo de OBD-II que actúa ante un CAN-BUS. Se realiza un análisis teórico de los diversos protocolos existentes haciendo principal insistencia en el protocolo relativo a la ECU (Unidad de Control de Motor).

Tras todo el análisis teórico, el trabajo se centrará en el diseño y montaje de un sistema que simule el funcionamiento de la ECU de un vehículo real, lo cual tiene gran utilidad para poder conocer un poco mejor su funcionamiento, permiéndonos poder hallar nuevas funciones o sensores que se le podrían aplicar. Este tipo de experimentos tienen una gran utilidad para la industria automovilística, ya que cada vez son más los distintos tipos y modelos de vehículos en el mercado, lo que provoca la aparición de nuevos parámetros distintos a los que habitualmente suelen ser usados.

Para el montaje del simulador se utilizarán materiales de bajo coste, como es el caso de resistencias, potenciómetros, pulsadores, etc. El simulador funcionará gracias a la implementación del estándar OBD-II, utilizando programación en Arduino, por lo cual se utilizará una placa de Arduino UNO, junto con una placa CAN-Bus Shield (La cual actuará como CAN). Toda la información será recogida por el ELM 327 (OBD-II) y se mostrará a través de la pantalla del ordenador gracias a un software de diagnóstico (ScanMaster-ELM), el cual recibe los datos a partir del puerto USB en el que se encuentra conectado el ELM 327.
Abstract

The device known as OBD (On Board Diagnostics) is an integrated diagnostic system that allows the monitoring and management of different parameters of a vehicle, such as the engine, the electronic brake system (ABS) Battery level and many other elements. This allows us to verify the correct functioning of all systems and to detect possible problems before they are aggravated. Thanks to this device we can also know at all times the speed that maintains the vehicle, the amount of gases it expels and, of course, the level of fuel, water, etc.

The work performed in the Final Work (TFG) is based entirely on the OBD-II protocol that acts before a CAN-BUS. A theoretical analysis of the various existing protocols is carried out with the main emphasis on the protocol related to the ECU (Motor Control Unit).

After the theoretical analysis, the work will focus on the design and assembly of a system that simulates the operation of an ECU of a real vehicle, which is very useful in order to understand a little better its operation, allowing us to find new functions or Sensors that could be applied. This type of experiment is very useful for the automotive industry, as more and more types and models of vehicles are on the market, resulting in the appearance of new parameters other than those that are usually used.

For the assembly of the simulator, low-cost materials will be used, such as resistors, potentiometers, push-buttons, etc. And the simulator will work thanks to the implementation of the OBD-II standard, using Arduino programming, which will use an Arduino UNO board, together with a CAN shield Shield (which will act as CAN). All information will be collected by the ELM 327 (OBD-II) and displayed through the computer screen using diagnostic software (ScanMaster-ELM), which receives the data from the USB port on which The ELM 327 is connected.
PARTE I: MEMORIA
Introducción

Motivaciones

Actualmente el sector automovilístico tiene un gran potencial en el mundo de la industria y en el comercial debido a su continua evolución en el ámbito tecnológico, lo que los hace ver como una apuesta de futuro. Esta industria enfoca gran parte de su desarrollo en amoldar los vehículos a las comodidades de los usuarios, su seguridad, etc.

El principal motivo que me incita a realizar este proyecto es el desarrollo que están sufriendo los sistemas de comunicación interna de los vehículos debido a las circunstancias mencionadas antes. Y aún a pesar de ello todos estos desarrollos giran alrededor de la interfaz OBD-II, gracias a la gran revolución que supuso su aparición que provocó la estandarización de los sistemas de diagnóstico de todos los vehículos, lo que demuestra su utilidad. Además, la interfaz OBD-II goza de gran acogida en el mundo de los “mecánicos aficionados”, lo que hace que sienta la necesidad de conocer más sobre la interfaz y si es posible encontrar nuevas formas para mejorarla y aplicarla a nuevos usos.

Objetivos

El objetivo principal que se busca al optar por realizar este proyecto es el de conseguir desarrollar el diseño de una ECU que se comporte lo más parecido posible al funcionamiento de la ECU de un vehículo de verdad. Y una vez desarrollado el montaje y la programación de la ECU la intención es que mediante un dispositivo OBD-II y software de diagnóstico se pueda establecer comunicación con dicha ECU y visualizar los datos que ahí se encuentren y se simulen.

Para conseguir el funcionamiento adecuado es totalmente necesario utilizar los protocolos de comunicación existentes y que actualmente están homologados, porque son esos protocolos con los que trabaja la interfaz OBD-II. Por lo que otro objetivo es realizar un estudio detallado de los diferentes protocolos, dando la importancia principal al protocolo CAN-BUS ya que la ECU que se diseñe funcionara gracias a él.

Dentro de lo que respecta a la parte de hardware, se hará a lo largo de toda la memoria un extenso análisis de los componentes y sus conexiones, a la vez que se mencionarán otros dispositivos o componentes similares que también podrían utilizarse para un proyecto semejante.

Y en lo que respecta al software, se busca conseguir comprender de forma clara cómo funciona el lenguaje de Arduino. Se explicarán sus funciones, las variables, las constantes y sus estructuras. Y se indicará cómo se implementan en el código del programa de simulación en el caso de que se utilicen. Además, también será importante comprender cómo trabaja el software de diagnóstico ScanMaster ya que tiene diversas herramientas aunque no de difícil uso.
Fundamentos teóricos:
OBDII

Historia e introducción del OBD

El sistema de diagnóstico a bordo, mayormente conocido como OBD (On Board Diagnostics), nace como tal a mediados de la década de los años 70 y comienzo de los años 80. Su aparición se debe básicamente a que en este periodo, principalmente en Estados Unidos, comienza a implantarse de una forma más notable el uso de electrónica de diagnóstico, es decir, se empieza a introducir en los vehículos componentes electrónicos cuya función primordial es el diagnóstico y control de los diversos parámetros de los automóviles, además de la detección de posibles errores o fallos que puedan llegar a dañar tanto el sistema eléctrico como el mecánico de los vehículos.

Los primeros parámetros que se buscó controlar fueron las emisiones de gases contaminantes como el CO₂, ya que se buscaba reducir la emisión de los gases conocidos como “gases de efecto invernadero” al mínimo posible, debido a que por aquel entonces se comenzaba a endurecer la normativa encargada de regular dichas emisiones, gracias a la aprobación de la “Ley de Aire Limpio” y el nacimiento de la Agencia de Protección Medioambiental (EPA) aprobada en el año 1970. Pero a partir de esto, también se buscó rentabilizar aún más la implantación de estos sistemas buscando la forma de que a partir de estos sistemas de monitorización, diagnóstico, etc., se consiguiera alargar la vida de los vehículos y reducir su número de averías, provocando que el sistema OBD se fuese desarrollando a un ritmo bastante elevado, ampliando su uso a gran diversidad de funciones de los automóviles, provocando el nacimiento al llegar a los años 90, exactamente el año 1991, del estándar OBD I que se volvió obligatorio su uso para todos los vehículos motorizados a partir de ese año, pero debido a que su eficiencia no era lo suficientemente elevada que se buscaba ya que principalmente monitorizaba las emisiones y no dejaba apenas ampliar su control a otros parámetros, apareció el estándar OBD II (Estándar que entró en vigor en el año 1996), el cual es el utilizado actualmente.

El estándar OBD-II supuso una gran revolución debido a que a las funciones que ya desempeñaba se le incorporaron una gran variedad de nuevas aplicaciones las cuales supusieron el gran salto que se buscaba dar en el sector automovilístico, ya que incorporaba un control prácticamente absoluto del motor, capacidad para el monitoreo del chasis y otras tantas funciones de control de sistemas electrónicos, lo que supuso que este sistema de diagnóstico se colocara en la vanguardia de esta tecnología y fuera absolutamente indispensable para cualquier tipo de vehículo en Estados Unidos. [1]
DIFERENCIA ENTRE LOS TERMINALES OBD I Y OBD II

**Figura 1. Comparativa terminales OBD**

Estados Unidos una vez que implantó el sistema OBD II en todos sus vehículos comenzó a implantarlo también en aquellos que se fabricaban en USA pero eran exportados a Europa lo cual provocó la entrada con enorme exito de este sistema en el mercado europeo. Su irrupción fue tan potente que en pocos años la industria automovilística europea lo utilizaba para prácticamente todos los vehículos que fabricaba, ya que es un sistema de fácil integración y muy eficiente. Tal fue la fuerza con la que se instauró este estándar, que la DIRECTIVA 98/69EG impuso su uso obligatorio en los vehículos de gasolina en el año 2000, en los vehículos Diesel en 2003, y en los vehículos pesados como son los camiones o autobuses en el año 2005. [3]
Funcionamiento OBD II

El sistema OBD II al ser un sistema para el diagnóstico, su función primordial es la monitorización, y hacia ello es donde están enfocadas prácticamente la totalidad de herramientas que funcionan para OBD II.

OBD II vigila de manera continuada el correcto funcionamiento de los diversos sistemas del vehículo (o sensores como en este proyecto) como el motor, el sistema ABS, etc. En el momento en que se produce un fallo, el OBD lo detecta, se carga en la memoria y avisa al usuario a partir de la iluminación del LED correspondiente.

Figura 2. Ejemplo OBD-II

El testigo luminoso de color amarillo del que se dispone se enciende en el vehículo al conectarse la llave de contacto (en nuestro proyecto esto no ocurrirá así ya que nuestro simulador estará constantemente alimentado desde el puerto USB del ordenador), si esto no sucede no está funcionando correctamente. Esta luz nos permite conocer si ocurren averías en el vehículo, lo cual se indica con los tres posibles casos que se indican a continuación.

- **Casos:**

  1. Si se producen destellos ocasionales esto indica la existencia de averías de tipo esporádico.

  2. Si el indicador o testigo permanece encendido de manera constante, esto es indicativo de la existencia de una avería de naturaleza seria que puede afectar a la emisión de gases o a la propia seguridad del vehículo.

  3. Si se detecta una avería muy grave susceptible de dañar el motor o afectar a la seguridad, el testigo de averías luce de manera intermitente sin parar en ningún momento. De darse este caso se deberá parar el motor inmediatamente ante la posibilidad de riesgo no solo para el sistema del vehículo, sino también por la propia seguridad de sus ocupantes.

Las labores de monitorización y diagnóstico explicadas anteriormente, que realiza el OBD II, se realizan gracias a que disponemos en los automóviles de una Unidad de control, conocida bajo el nombre de ECU, a la cual se le conecta nuestro dispositivo y a partir de esa conexión es de donde se recoge toda la información. Los vehículos modernos disponen por regla general de
más de una ECU para gestionar ciertos parámetros independientes de nuestro vehículo, y para asegurar su correcto funcionamiento. Aunque se debe tener en cuenta que las relaciones entre estos parámetros deben mantenerse acotadas, ya que dependiendo de las condiciones externas variarían ciertos rangos, si esto no sucede, es indicativo de que se está produciendo algún mal funcionamiento en los sistemas de nuestro vehículo.

A partir de la conexión con la ECU se verifica una serie de parámetros que dictaminarán el funcionamiento del motor:

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

**Figura 3. ECU**

Todas las mediciones que se registran las realizan diversos sensores, los cuales están preparados para resistir un determinado número de condiciones que se pueden considerar de carácter extremo.

La comunicación entre la Unidad de Control (ECU) y el equipo de diagnóstico se establece mediante un protocolo. Hay diversos protocolos que afectan a la configuración de los pines (Los pines podemos verlos en la figura (Figura 1) y otras funciones, pero todo esto lo veremos más adelante, estudiando cada uno de los protocolos existentes. Aun así se pueden destacar 3 protocolos que actualmente destacan ligeramente sobre el resto:

- Protocolo ISO 9141 (Utilizado por Europa, diversos países de Asia y “Chrysler”)
- Protocolo SAE J1850 VPW (Utilizado casi exclusivamente por “General Motors”)
- Protocolo SAE J1850 PWM (Utilizado por “Ford”)

**Controles realizados por OBD II**

Los controles que puede realizar un OBD II se pueden calificar en tres grandes grupos, de los cuales dos de ellos dependen del tipo de motor del que dispone el vehículo. Estos motores son los motores de gasolina y los motores diesel.

**Control Contaminación**

Para este tipo de control a diferencia de los dos siguientes se incorpora una segunda sonda lambda que se instala detrás del catalizador para verificar el funcionamiento del mismo y de la sonda lambda anterior al catalizador.
El OBD controla:

- Verifica el estado de funcionamiento del sistema de regulación lambda.
- Analiza el estado de envejecimiento de la sonda.
- Controla la tensión que generan los gases.
- Analiza el estado de funcionamiento de los elementos calefactores.

**Control Motores Gasolina**

- Diagnóstico de envejecimiento de sondas lambda.
- Prueba de tensión de sondas lambda.
- Vigilancia del rendimiento del catalizador.
- Sistema de recuperación de vapores de combustible.
- Sistema de aire secundario (si el vehículo lo incorpora).
- Prueba de diagnóstico de fugas.
- Sistema de alimentación de combustible.
- Control del sistema de gestión electrónica.
- Fallos de la combustión.
- Funcionamiento del sistema de comunicación entre unidades de mando, por ejemplo el Can-Bus.
• Sensores y actuadores del sistema electrónico que intervienen en la gestión del motor o están relacionados con las emisiones de escape. [1]

Control Motores Diesel
• Fallos de la combustión.

• Regulación del comienzo de la inyección.

• Regulación de la presión de sobrealimentación.

• Recirculación de gases de escape.

• Sensores y actuadores del sistema electrónico que intervienen en la gestión del motor o están relacionados con las emisiones de escape.

• Control del sistema de gestión electrónica.

• Funcionamiento del sistema de comunicación entre unidades de mando, por ejemplo el Can-Bus. [1]

Modos de medición OBD II
Los modos de medición son aquellos que posibilitan la realización de las diversas funciones OBD-II, los cuales vienen implementadas por los diversos protocolos que veremos en los próximos apartados.

Lo que provocan los modos es que según la parte de información a la que queramos acceder necesitamos utilizar un modo diferente, lo cual contribuye a una mayor estructuración y orden del sistema, ya que dentro de cada uno de ellos podemos usar un abanico de parámetros bastante amplio, permitiéndonos desde registrar datos para su verificación, extraer códigos de averías, borrarlos y realizar pruebas dinámicas de actuadores. Estos modos no solo se encargan del registro, sino que también son los responsables del envío de la información registrada al software de diagnóstico correspondiente (el que estemos utilizando), para que este se encargue de interpretar y representar los datos correctamente, para facilitar el diagnóstico del problema del vehículo.

El formato estándar de trama de cada uno de los modos está compuesto por 7 bytes, y aunque no sea habitual el uso de todos se recomienda que el resto se pongan a $00 o $55. Cada byte tiene distinto contenido, que dependerá de que la trama sea 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, pero la existencia de este byte dependerá del protocolo sobre el que se envía el mensaje, debido a que únicamente el protocolo que utilizaremos en este proyecto (protocolo CAN) es el único con capacidad para soportar 8 bytes, a diferencia del resto que solo son capaces de soportar tramas de datos de 7 bytes.
**Modo 01: Obtención de datos de diagnóstico actualizados**

Este modo, conocido popularmente como “modo de flujo de datos”, es el encargado de acceder a los datos en tiempo real de valores tanto analógicos como digitales de salidas y entradas a la ECU en la cual se encuentra implementado el protocolo correspondiente.

El Modo 01 nos permite ver, o lo que es lo mismo, monitorizar la temperatura de motor y/o el voltaje generado por dicho motor a partir de una sonda lambda (la cual realiza un control en tiempo real) entre otra gama extensa de parámetros.

A continuación, vemos la siguiente tabla (Tabla 1), en la cual se resume la estructuración de una trama OBD-II en Modo 01, para poder ser capaces de realizar peticiones a dicho modo. También es importante comentar que trabajando en este modo tenemos la capacidad de trabajar realizando peticiones de diversos PID en una única petición, aunque no es totalmente aconsejable, ya que lo mejor para asegurar un correcto funcionamiento es solicitar un único PID por petición, debido a que por cada PID que se solicita a la ECU en la que estamos realizando la monitorización, ésta envía respuestas distintas. [1]

<table>
<thead>
<tr>
<th>BYTE</th>
<th>PARÁMETROS ABARCADOS</th>
<th>VALOR HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Indica que estamos realizando una petición en el Modo 01.</td>
<td>01</td>
</tr>
<tr>
<td>#2</td>
<td>Valor hexadecimal del PID requerido.</td>
<td>XX</td>
</tr>
<tr>
<td>#3</td>
<td>Valor hexadecimal del segundo PID requerido. (Opcional)</td>
<td>XX</td>
</tr>
<tr>
<td>#4</td>
<td>Valor hexadecimal del tercer PID requerido. (Opcional)</td>
<td>XX</td>
</tr>
<tr>
<td>#5</td>
<td>Valor hexadecimal del cuarto PID requerido. (Opcional)</td>
<td>XX</td>
</tr>
<tr>
<td>#6</td>
<td>Valor hexadecimal del quinto PID requerido. (Opcional)</td>
<td>XX</td>
</tr>
<tr>
<td>#7</td>
<td>Valor hexadecimal del sexto PID requerido. (Opcional)</td>
<td>XX</td>
</tr>
</tbody>
</table>

**Tabla 1. MODO 01 Peticiones OBD-II**

Los identificadores PID anteriormente mencionados, también son conocidos bajo el nombre de “Parámetros ID”, indican al sistema de diagnóstico de a bordo la información que se requiere, y existe un listado extenso de PID para el modo en el que nos encontramos:

- **PID $00**: Indica los PIDs soportados desde el PID $01 al $20.
- **PID $20**: indica los PIDs soportados desde el PID $21 hasta el $40.

Los dos que aquí se mencionan son los más comúnmente utilizados, pero existen más que siguen el mismo orden que estos dos. Además, hay que tener en cuenta que cualquier ECU debe ser capaz de soportar el PID $00.

Tras ver las peticiones del Modo 01, ahora veremos el formato de las respuestas, en donde se indica que la solicitud realizada por el modo 01 se responde sumándole 40 en hexadecimal. Los datos solicitados se traducen en A, B, C y D y según la información a pedir se usan todos o solo aquellos que son necesarios.
<table>
<thead>
<tr>
<th>BYTE</th>
<th>PARAMETROS ABARCADOS</th>
<th>VALOR HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Indica que estamos realizando una respuesta en el Modo 01.</td>
<td>41</td>
</tr>
<tr>
<td>#2</td>
<td>Valor hexadecimal del PID al que se responde.</td>
<td>XX</td>
</tr>
<tr>
<td>#3</td>
<td>DATO A</td>
<td>XX</td>
</tr>
<tr>
<td>#4</td>
<td>DATO B</td>
<td>XX</td>
</tr>
<tr>
<td>#5</td>
<td>DATO C</td>
<td>XX</td>
</tr>
<tr>
<td>#6</td>
<td>DATO D</td>
<td>XX</td>
</tr>
<tr>
<td>#7</td>
<td>No recibe ningún uso.</td>
<td>XX</td>
</tr>
</tbody>
</table>

*Tabla 2. MODO 01 Respuestas OBD-II*

Actualmente se dispone de gran variedad de PIDs, se cuenta con hasta más de 60 distintos, pero toda esta lista puede abreviarse, ya que mucho tienen una importancia muy relevante frente a otros que apenas tienen influencia en cualquier protocolo del estándar OBD-II. Todos estos PID de gran importancia vienen recogidos por el Protocolo SAE J1979, en el que se establece que todos los PIDs y sus modificaciones debes estar recogidos en la norma OBD-II, aunque se trate de PIDs propios de un fabricante en particular. En la siguiente tabla aparecen recogidos los principales PID en el estándar OBD-II.

<table>
<thead>
<tr>
<th>PID</th>
<th>Bytes</th>
<th>Descripción</th>
<th>Mínimo</th>
<th>Máximo</th>
<th>Unidades</th>
<th>Fórmula</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>4</td>
<td>PIDs soportados [01-20]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>32 Bits: BitX=0&gt;PIDX Sop. BitX=1 &gt; PIDX No sop.</td>
</tr>
<tr>
<td>01</td>
<td>4</td>
<td>Estado de MIL y número de DTCs almacenados.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bits codificados</td>
</tr>
<tr>
<td>03</td>
<td>2</td>
<td>Estado del sistema de combustible.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bits codificados</td>
</tr>
<tr>
<td>04</td>
<td>1</td>
<td>Valor de carga del motor.</td>
<td>0</td>
<td>100</td>
<td>%</td>
<td>A×100/255</td>
</tr>
<tr>
<td>05</td>
<td>1</td>
<td>T° del refrigerante</td>
<td>-40</td>
<td>215</td>
<td>ºC</td>
<td>A-40</td>
</tr>
<tr>
<td>0A</td>
<td>1</td>
<td>Presión del combustible.</td>
<td>0</td>
<td>765</td>
<td>kPa</td>
<td>A×3</td>
</tr>
<tr>
<td>0C</td>
<td>2</td>
<td>Revoluciones por minuto del motor.</td>
<td>0</td>
<td>16.383</td>
<td>rpm</td>
<td>((A×256)+B)/4</td>
</tr>
<tr>
<td>0D</td>
<td>1</td>
<td>Velocidad del vehículo.</td>
<td>0</td>
<td>255</td>
<td>Km/h</td>
<td>A</td>
</tr>
<tr>
<td>0F</td>
<td>1</td>
<td>Temperatura del aire de entrada.</td>
<td>-40</td>
<td>215</td>
<td>ºC</td>
<td>A-40</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>Posición del acelerador.</td>
<td>0</td>
<td>100</td>
<td>%</td>
<td>A×100/255</td>
</tr>
<tr>
<td>13–1B</td>
<td>2</td>
<td>Voltaje en los distintos sensores de oxígeno.</td>
<td>0</td>
<td>1.275</td>
<td>V</td>
<td>A/200 (B-128)×100/128</td>
</tr>
<tr>
<td>1C</td>
<td>1</td>
<td>Estándar OBD conforme al vehículo.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bits Codificados</td>
</tr>
<tr>
<td>1F</td>
<td>2</td>
<td>Tiempo transcurrido desde el arranque.</td>
<td>0</td>
<td>65.535</td>
<td>Seg</td>
<td>(A×256)+B</td>
</tr>
<tr>
<td>PID</td>
<td>4</td>
<td>PIDs soportados [21-40]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>---</td>
<td>----------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>Distancia recorrida con el indicador MIL encendido.</td>
<td>0</td>
<td>65.535</td>
<td>Km</td>
<td>(A×256)+B</td>
</tr>
<tr>
<td>2C</td>
<td>1</td>
<td>Recirculación de gases de escape o EGR.</td>
<td>0</td>
<td>100</td>
<td>%</td>
<td>A×100/255</td>
</tr>
<tr>
<td>2E</td>
<td>1</td>
<td>Depuración por evaporación.</td>
<td>0</td>
<td>100</td>
<td>%</td>
<td>A×100/255</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>Distancia recorrida desde la última limpieza de los errores.</td>
<td>0</td>
<td>65.535</td>
<td>Km</td>
<td>(A×256)+B</td>
</tr>
<tr>
<td>3C – 3F</td>
<td>2</td>
<td>Temperatura del catalizador.</td>
<td>-40</td>
<td>6.513</td>
<td>ºC</td>
<td>(A×256)+B/10-40</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>PIDs soportados [41-60]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>4E</td>
<td>2</td>
<td>Módulo controlador del voltaje.</td>
<td>0</td>
<td>65.535</td>
<td>V</td>
<td>(A×256)+B</td>
</tr>
<tr>
<td>4D</td>
<td>2</td>
<td>Tiempo en marcha mientras el MIL permanece encendido.</td>
<td>0</td>
<td>65.535</td>
<td>min</td>
<td>(A×256)+B</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>Tipo de combustible.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bits codificados</td>
</tr>
<tr>
<td>5B</td>
<td>1</td>
<td>Nivel de batería híbrida.</td>
<td>0</td>
<td>100</td>
<td>%</td>
<td>A×100/255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PID</th>
<th>20</th>
<th>PIDs soportados [21-40]</th>
<th>-</th>
<th>-</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 – 2B</td>
<td>4</td>
<td>Voltaje equivalente de la sonda lambda.</td>
<td>0</td>
<td>8</td>
<td>V</td>
</tr>
<tr>
<td>24 – 2B</td>
<td>4</td>
<td>Distancia recorrida con el indicador MIL encendido.</td>
<td>0</td>
<td>65.535</td>
<td>Km</td>
</tr>
<tr>
<td>24 – 2B</td>
<td>4</td>
<td>Distancia recorrida desde la última limpieza de los errores.</td>
<td>0</td>
<td>65.535</td>
<td>Km</td>
</tr>
<tr>
<td>24 – 2B</td>
<td>4</td>
<td>Corriente equivalente de la sonda lambda.</td>
<td>-128</td>
<td>128</td>
<td>mA</td>
</tr>
<tr>
<td>34 – 3B</td>
<td>4</td>
<td>Corriente equivalente de la sonda lambda.</td>
<td>-128</td>
<td>128</td>
<td>mA</td>
</tr>
<tr>
<td>3C – 3F</td>
<td>2</td>
<td>Temperatura del catalizador.</td>
<td>-40</td>
<td>6.513</td>
<td>ºC</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>Tipo de combustible.</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5B</td>
<td>1</td>
<td>Nivel de batería híbrida.</td>
<td>0</td>
<td>100</td>
<td>%</td>
</tr>
</tbody>
</table>

**Tabla 3: Principales PIDs estándar OBD-II**

**Modo 02: Acceso al cuadro de los datos congelados**

En el estándar actual del OBD-II, este modo puede ser considerado primordial, ya que es el más importante de todos porque la ECU toma una muestra de todos los valores relacionados con las emisiones, en el momento exacto en el que se produce cualquier fallo. Gracias a esta acción podemos conocer a partir de los datos recogidos, las condiciones exactas en las que ocurrió dicho fallo. Por regla general, todos los datos son registrados en una única tabla, donde existe un cuadro de datos que corresponde al primer fallo detectado aunque esto depende de la ECU. [1]
Para comprender en detalle en qué se basa este modo para su funcionamiento, no es necesario andar dando demasiadas vueltas, ya que acepta los mismos PIDs que el Modo 01, y ni siquiera varía sus significados respectivos ya que los datos son los mismos en ambos modos, con la única salvedad que los datos que registra y controla el modo 01 son leídos en tiempo real y en el modo 02 los datos son los que han sido almacenados del momento en el que ocurrió algún tipo de fallo. [8]

Como se ha dicho en el párrafo anterior estos dos modos son prácticamente iguales y eso conlleva que el formato de las peticiones y respuestas en este modo es tan similar a las del Modo 01 que prácticamente son idénticos. Estos presentan su única diferencia en el indicador de cuadro de datos, el cual, identifica el cuadro de datos congelados para evitar confusiones en caso de existir más de un cuadro.

Todo lo expresado anteriormente se plasma en las siguientes dos tablas. Una de peticiones y la otra de las respuestas:

**Tabla 4: Peticiones estándar OBD-II en Modo 02**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>PARAMETROS ABARCA</th>
<th>VALOR HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Indica que estamos realizando una petición en el Modo 02.</td>
<td>02</td>
</tr>
<tr>
<td>#2</td>
<td>Valor hexadecimal del PID al que se responde.</td>
<td>XX</td>
</tr>
<tr>
<td>#3</td>
<td>Número de Cuadro</td>
<td>XX</td>
</tr>
<tr>
<td>#4</td>
<td>Valor hexadecimal del segundo PID requerido. <em>(Opcional)</em></td>
<td>XX</td>
</tr>
<tr>
<td>#5</td>
<td>Número de Cuadro</td>
<td>XX</td>
</tr>
<tr>
<td>#6</td>
<td>Valor hexadecimal del tercer PID requerido. <em>(Opcional)</em></td>
<td>XX</td>
</tr>
<tr>
<td>#7</td>
<td>Número de Cuadro</td>
<td>XX</td>
</tr>
</tbody>
</table>

**Tabla 5: Respuestas estándar OBD-II en Modo 02**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>PARAMETROS ABARCA</th>
<th>VALOR HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Indica que estamos realizando una respuesta en el Modo 01.</td>
<td>42</td>
</tr>
<tr>
<td>#2</td>
<td>Valor hexadecimal del PID al que se responde.</td>
<td>XX</td>
</tr>
<tr>
<td>#3</td>
<td>Número de Cuadro.</td>
<td>XX</td>
</tr>
<tr>
<td>#4</td>
<td>DATO A</td>
<td>XX</td>
</tr>
<tr>
<td>#5</td>
<td>DATO B</td>
<td>XX</td>
</tr>
<tr>
<td>#6</td>
<td>DATO C</td>
<td>XX</td>
</tr>
<tr>
<td>#7</td>
<td>DATO D</td>
<td>XX</td>
</tr>
</tbody>
</table>

**Modo 03: Códigos de diagnóstico almacenados**

Este modo nos capacita para leer de la memoria de la ECU todos los DTCs almacenados, debido a que permite extraer todos los códigos de diagnóstico de Error o DTCs disponibles.

DTC son las siglas de “Data Trouble Code” y se conocen bajo este nombre a los códigos que se componen de tres dígitos precedidos por un identificador alfanumérico.
El sistema de funcionamiento de las DTCs se puede apreciar en la imagen anterior (Figura 5), ya que cuando la ECU del vehículo reconoce e identifica un problema, se almacena en la memoria un DTC que corresponde a ese fallo, lo cual nos permite determinar la causa principal de un problema. Como podemos ver en la imagen los DTCs de sistemas se clasifican en B, C, P y U.

**B: BODY:** Comprende los sistemas que conforman la parte de carrocería y confort, también algunos sistemas relacionados con el inmovilizador.

**C: CHASIS:** Comprende los sistemas relacionados con el chasis como pueden ser algunos sistemas ABS – AIRBAG y sistemas de diferencial que no estén relacionados con la gestión de la transmisión automática.

**P: POWERTRAIN:** Comprende los códigos relacionado con el motor y la transmisión automática.

**U: NETWORK:** Comprende los problemas relacionados con la transmisión de datos de un módulo a otro, las redes de comunicación se pueden averiar y dejar sistemas completos por fuera del sistema. En ese caso cualquiera de los módulos restantes puede generar un código relacionado con ese sistema. [10]

En las siguientes tablas podremos observar cómo se realiza las peticiones al Modo 03 (Tabla 6) y las respuestas (Tabla 7), las cuales serán más o menos dependiendo del número de fallas o DTCs que se encuentren almacenados. Y en el caso de no existir errores, no se produciría ninguna respuesta a la petición realizada.
Tabla 6: Peticiones estándar OBD-II en Modo 03

<table>
<thead>
<tr>
<th>BYTE</th>
<th>PARAMETROS ABARCADOS</th>
<th>VALOR HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Indica que estamos realizando una petición en el Modo 03.</td>
<td>03</td>
</tr>
</tbody>
</table>

Tabla 7: Respuestas estándar OBD-II en Modo 03

<table>
<thead>
<tr>
<th>BYTE</th>
<th>PARAMETROS ABARCADOS</th>
<th>VALOR HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Indica que estamos realizando una respuesta en el Modo 03.</td>
<td>43</td>
</tr>
<tr>
<td>#2</td>
<td>Cantidad DTCs almacenados.</td>
<td>XX</td>
</tr>
<tr>
<td>#3</td>
<td>High Byte (HB)</td>
<td>XX</td>
</tr>
<tr>
<td>#4</td>
<td>Low Byte (LB)</td>
<td>XX</td>
</tr>
<tr>
<td>#5</td>
<td>No recibe ningún uso.</td>
<td>00</td>
</tr>
<tr>
<td>#6</td>
<td>No recibe ningún uso.</td>
<td>00</td>
</tr>
<tr>
<td>#7</td>
<td>No recibe ningún uso.</td>
<td>00</td>
</tr>
</tbody>
</table>

Todos los DTCs como se han podido ver se encuentran codificados, y esta codificación en High byte (HB) y Low byte (LB). Los High bytes codifican los tres primeros caracteres del DTC y los Low bytes el resto.

<table>
<thead>
<tr>
<th>CARACTER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITS</td>
<td>HB7-HB6</td>
<td>HB5-HB4</td>
<td>HB3-HB0</td>
<td>LB7-LB4</td>
<td>LB3-LB0</td>
</tr>
<tr>
<td>CODIGO</td>
<td>00=P</td>
<td>00 = 0</td>
<td>xxxx valor en Hexadecimal</td>
<td>xxxx valor en Hexadecimal</td>
<td>xxxx valor en Hexadecimal</td>
</tr>
<tr>
<td>01=C</td>
<td>01 = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10=B</td>
<td>10 = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11=U</td>
<td>11 = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tabla 8: Codificación DTC

Modo 04: Borrado de códigos de falla (DTC) y cuadro de almacenados

Entramos en un modo que nos permite borrar todos los códigos almacenados en la PCM (Unidad de Control Electrónico), como son el caso de los DTCs y los cuadros de datos grabados, pero, ésta no es su única función ya que también es el encargado de apagar el indicador luminoso MIL ya que los códigos de error al ser borrados han desaparecido y hasta que no se vuelvan a registrar, este indicador no volverá a lucir. [8]

<table>
<thead>
<tr>
<th>BYTE</th>
<th>PARAMETROS ABARCADOS</th>
<th>VALOR HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Solicita el borrado o reseteo de todos los DTCs almacenados y del cuadro de congelados en el Modo 04.</td>
<td>04</td>
</tr>
</tbody>
</table>

Tabla 9: Peticiones estándar OBD-II en Modo 04
BYTE | PARAMETROS ABARCADOS | VALOR HEXADECIMAL
--- | --- | ---
#1 | Indica que el borrado ha sido realizado correctamente en el Modo 04. | 44

**Tabla 10: Respuestas estándar OBD-II en Modo 04**

Este modo presenta una diferencia muy importante con respecto a los demás. Esta diferencia se da en las respuestas, cuando se da el caso de no poderse realizar el borrado de datos, las mismas cambian totalmente, como se puede observar en la tabla 11.

BYTE | PARAMETROS ABARCADOS | VALOR HEXADECIMAL
--- | --- | ---
#1 | Indicador de aviso de respuesta negativa. | 7F
#2 | Modo de solicitud de la acción pedida. | 04
#3 | Condiciones incorrectas. | 22

**Tabla 11: Respuestas negativas estándar OBD-II en Modo 04**

**Modo 05: Test de los sensores de oxígeno**

Como su propio nombre indica el Modo 05 está destinado a las funciones de control de los sensores de oxígeno, y las realiza devolviendo los resultados de las pruebas realizadas a los sensores de oxígeno para tener la capacidad de determinar el funcionamiento de los mismos y la eficiencia del convertidor catalítico, el cual es el principal encargado del control de las emisiones de CO₂ y otros gases contaminantes. Estos sensores si el vehículo esta frío o apagado, marcarán esta prueba como no preparada para ejecutarse.

Los resultados de estos test se obtienen mediante la devolución de determinados voltajes dependiendo de la mezcla aire y combustible, dando lugar a los valores que se quieren conocer. [8]

- Umbral de voltaje en el sensor de oxígeno (RICO A POBRE).
- Umbral de voltaje en el sensor de oxígeno (POBRE A RICO).
- Voltaje alto/bajo en el sensor para el cambio.
- Tiempo transcurrido del cambio de rico a pobre y de pobre a rico.
- Voltajes máximos y mínimos en el sensor en el momento que la prueba está realizándose.
- Tiempos de transición entre los diferentes sensores del sistema.
Los diversos resultados enumerados anteriormente y que se han podido ver como aparecen en pantalla gracias a la Figura 6, también se muestran de forma gráfica para poder ver con cierta claridad el comportamiento detectado mediante el test.

**Modo 06: Resultado de pruebas de control de otros transductores**

En los sistemas con el estándar OBD-II, este modo permite obtener los resultados de todas las pruebas de a bordo, es decir, nos permite un control de todo el sistema al completo.

Los monitores del sistema identifican los sistemas que pueden ser monitoreados y en el caso de que se pueda, identifican el estado de los mismos. Dicho monitoreado lo realizan mediante una serie de pruebas que se enumeran a continuación:

- Errores relativos al ámbito de la combustión: bujías, arranque, inyectores, etc.
- Sistema de análisis de combustible: sensores de oxígeno, sensores de ciclo cerrado, retroalimentación de combustible y sistema de encendido.
- Catalizador.
• Calentamiento del catalizador.

• Sistema de evaporación.

• Sensores de oxígeno.

• Sistema secundario de aire.

• Refrigerador de aire acondicionado.

• Calentamiento del sensor de oxígeno.

• Recirculación de gases de escape, conocido como EGR.

Cuando estas pruebas detectan errores de algún tipo, los registran, y para poder verlos y corregirlos será necesario acceder al código de falla. También es importante añadir que este modo no es soportado por todos los vehículos lo que puede dar lugar a mensajes de error. [8]

**Modo 07: Mostrar código de diagnóstico pendiente**

El modo 07 permite leer desde la memoria de la ECU todos los DTC pendientes, es decir, todos los cuales no se han borrado de dicha memoria. [8]

**Modo 08: Control de funcionamiento de componentes (C. de a bordo)**

En este modo podemos realizar la denominada prueba de actuadores. Con esta función se puede activar y desactivar los actuadores como bombas de combustible, válvula de ralentí, etc. [8]

**Modo 09: Información del vehículo**

Aquí nos encontramos con un modo especial, ya que este modo no es estrictamente necesario y se considera un modo opcional, ya que hay una amplia gama de vehículos que no disponen de este modo en su sistema. Todo esto se debe a que la única aportación que nos proporciona este modo es la de mostrarnos el número de serie del automóvil y algunas características específicas. [8]
Protocolos para la comunicación OBD II

El estándar OBD-II, es un tipo de estándar que al querer implantarse de forma obligatoria en todo tipo de vehículos, ha tenido que disponer de una gran versatilidad a lo que a compatibilidad respecta, ya que existen una enorme variedad de modelos y marcas de vehículos, al igual que por culpa de esto mismo, también existen incontables software o plataformas de diagnóstico que trabajan con dicho estándar y son capaces de abarcar gran cantidad de modelos automovilísticos ya que cada fabricante usaba su propio sistema de auto diagnóstico a bordo con su propio protocolo de comunicación y un conector único para el sistema de diagnóstico. A raíz de todo esto, se obligó a la implantación de un conector trapezoidal de 16 pines para el sistema de auto diagnóstico OBD-II, para tener la capacidad de abarcar cualquier protocolo de comunicación con un mismo dispositivo tester, y junto con dicho conector nacieron los protocolos OBD homologados, que son aquellos que están aprobados por los distintos organismos tanto europeos como americanos, aunque esto no cierra la puerta a la aparición de nuevos protocolos, sino que todo lo contrario, porque estos protocolos dejan la puerta abierta a la aparición de otros nuevos que se puedan implementar en marcas específicas para su uso exclusivo, con la única condición de que esos nuevos protocolos estén basados y sean compatibles con alguno de los protocolos principales para poder mantener un control sobre ellos por parte de los organismos internacionales.

A continuación, se enumeran los cinco grandes protocolos homologados tanto por la EPA (Agencia De Protección Al Ambiente) como por la UE (Unión Europea), y que más adelante serán analizados más en profundidad, incluido el protocolo del CAN que es el utilizado en este proyecto. [5] [11]

- **SAE J1850 PWM**: Usado exclusivamente por vehículos fabricados y pertenecientes a Ford. Este protocolo usa señales diferenciales y posee una velocidad de transferencia de hasta 41,6 Kbps y sus alimentaciones al conector de diagnóstico son:

  - 5: Tierra
  - 16: Voltaje de la batería
  - 2: Datos 1
  - 10: Datos 2

- **SAE J1850 VPW**: Este protocolo es utilizado prácticamente en exclusiva por General Motors (GM) en todos sus autos OBD-II, pero también lo utiliza desde el año 2000 Chrysler. Sus alimentaciones al conector de diagnóstico son:

  - 5: Tierra
  - 16: Voltaje de la batería
  - 2: Datos

- **ISO 9141-2**: Protocolo usado por vehículos de la marca Chrysler, en la totalidad de sus autos hasta 1999, y lo continúa utilizando en un buen porcentaje de sus autos, en modelos posteriores al 2000. También lo usan la mayoría de las marcas Asiáticas, como: BMW, Mercedes Benz, VW, Mercedes Benz, VW, Piaggio, Porsche, Seat, Smart, Volvo, etc. Sus alimentaciones al conector de diagnóstico son:
- **ISO 14230-4 (KWP2000):** Es el protocolo estándar de la norma europea EOBD y es muy común en vehículos a partir del año 2003, sobre todo en los Renault, Peugeot, Daewoo y descendientes Opel (vendido por GM). Su principal uso es para la diagnosis off-board de centralitas electrónicas de vehículos y la reprogramación. Sus alimentaciones al conector de diagnóstico son:
  - 5: Tierra
  - 16: Voltaje de la batería
  - 7: Datos
  - 15: Datos 2

- **CAN (ISO 11898/15765):** El protocolo CAN para el bus de diagnóstico comenzó su uso en el año 2003 y de allí a la fecha actual, algunos modelos que los aplican son: Fiesta, Eco Sport, Lobo, Explorer (Ford), Ram, Durango (Chrysler), Vectra, Malibu (GM), Murano, Saab, entre otros. Además desde el año 2008 es obligatorio en los vehículos estadounidenses su implantación. Sus alimentaciones al conector de diagnóstico son:
  - 5: Tierra
  - 16: Voltaje de la batería
  - 6: Datos H
  - 14: Datos L

Los protocolos mencionados no solo se rigen por los organismos y modelos descritos hasta ahora, sino que también se estructuran bajo un modelo concreto. Y ese modelo es el conocido bajo el nombre de modelo OSI que son las siglas de “*Open System Interconnection*”.

El modelo OSI es un producto del esfuerzo de Open Systems Interconnection en la Organización Internacional de Estándares. Es una prescripción de caracterización para conseguir la estandarización de las funciones de un sistema de comunicaciones en términos de abstracción de capas. Esta estandarización se resume en agrupar funciones similares de comunicación en capas lógicas, y cada capa sirve a la capa sobre ella y esa misma capa es servida por la capa inferior a ella.

Las capas del modelo OSI se estructuran en un total de siete, las cuales todas son dependientes de sus anteriores ya que si no se cumplen las condiciones de una capa no se puede pasar a la inmediatamente superior. Estas siete capas de la inferior a la superior son las siguientes: Nivel físico, Nivel de enlace de datos, Nivel de Red, Nivel de transporte, Nivel de sesión, Nivel de presentación y por último Nivel de aplicación. [12]
En la tabla que se muestra a continuación (tabla 12) se resumen la organización de todos los protocolos actuales agrupados por las capas del modelo OSI a las que pertenecen y a los modelos homologados principales.

<table>
<thead>
<tr>
<th>CAPA M. OSI</th>
<th>OBD-II PROTOCOLOS</th>
<th>CAN ISO 15765-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIVEL 1 FÍSICO</td>
<td>SAE J1850 PWM</td>
<td>SAE J1850 VPW</td>
</tr>
<tr>
<td>NIVEL 2 ENLACE</td>
<td>SAE J1850</td>
<td>SAE J1850</td>
</tr>
<tr>
<td>NIVEL 3 RED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NIVEL 4 TRANSPORTE</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NIVEL 5 SESIÓN</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NIVEL 6 PRESENTACIÓN</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Tabla 12: Protocolos estándar OBD-II modelo OSI**
Tras todo esto vamos a analizar en profundidad los cinco protocolos principales, finalizando con el CAN BUS que es el cual nos importa en este proyecto.

**SAE J1850 PWM y VPW**

Las características de estos dos protocolos las vemos resumidas en las siguientes tablas:

<table>
<thead>
<tr>
<th>Característica</th>
<th>Descripción</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS +</td>
<td>Pin 2</td>
</tr>
<tr>
<td>Autobús</td>
<td>Pin 10</td>
</tr>
<tr>
<td>12V</td>
<td>Pin 16</td>
</tr>
<tr>
<td>Tierra</td>
<td>Pines 4, 5</td>
</tr>
</tbody>
</table>

**Estado del autobús:** Activo cuando el BUS + es tirado ALTO, BUS - es tirado BAJO

**Voltaje máximo de la señal:** 5V

**Voltaje mínimo de la señal:** 0V

**Número de bytes:** 12

**Tiempo binario:** '1' bit - 8uS, '0' bit - 16uS, Inicio de la trama - 48uS

**Tabla 13: Características SAE J1850 PWM**

<table>
<thead>
<tr>
<th>Característica</th>
<th>Descripción</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS +</td>
<td>Pin 2</td>
</tr>
<tr>
<td>12V</td>
<td>Pin 16</td>
</tr>
<tr>
<td>Tierra</td>
<td>Pines 4, 5</td>
</tr>
</tbody>
</table>
**Tabla 14: Características SAE J1850 VPW**

Los protocolos J1850 PWM y VPW se pueden describir simultáneamente ya que en términos generales ambos son lo mismo, con la diferencia fundamental que el PWM se utiliza tanto para velocidades de transmisión alta y media, y el VPW se usa para las velocidades bajas.

Ambos protocolos del J1850 al ser en principio prácticamente iguales, utilizan como forma de acceso al medio el modo CSMA/CR. Este modo es un método de acceso múltiple en el que múltiples estaciones tienen acceso al medio de transmisión, y a diferencia de otros CSMA, el CSMA / CR se utiliza en el CAN BUS, para evitar la colisión con el arbitraje. El arbitraje utilizado en CSMA / CR se basa en las prioridades con el que las estaciones transmiten sus datos a través del medio de transmisión. Y dicho arbitraje se lleva a cabo solo cuando varias estaciones intentan al mismo tiempo el acceso al medio de transmisión.

Por un lado, si hablamos del protocolo J1850 PWM, estamos tratando con la versión de velocidad más rápida, ya que con este modelo somos capaces de alcanzar los 41,6 kbit/s, y además su transmisión la realiza en modo diferencial mediante dos cables. Como su propio nombre indica trabaja a partir de modulaciones PWM “Pulse Width Modulation” o lo que es lo mismo codifica por ancho de pulso.

Por otro, cuando se trata del protocolo J1850 VPW, estamos ante la versión de velocidad más lenta de las dos, debido a que este tipo de protocolo trabaja a velocidades de 10,4 kbit/s y utiliza una codificación VPW “Variable Pulse Width”, es decir, de ancho de pulso variable. También añadir que a diferencia del PWM, en este protocolo se transmite mediante un único cable referido a masa.

Por regla general, los bits de interfaz J1850 se encuentran en el conector OBDII que incorporan los vehículos, pero sin embargo debido a la edad del estándar del J1850, no puede residir en muchos modelos de última generación por problemas de compatibilidad. [11] [13]

El SAE J1850 podemos conocerlo mejor si estudiamos su capa física y la de enlace por separado.
**Capa física SAE J1850**

En el SAE J1850 la capa física está optimizada para una velocidad de datos de 10,4 Kbps (VPW) mientras que la otra capa física está optimizada para una velocidad de datos de 41,6 Kbps (PWM). Los parámetros de capa física se especifican, debido a que se detectarán en el medio de red, y no dentro de un módulo particular.

Como se ha comentado antes según el protocolo del SAE J1850 con el que nos encontremos, en la capa física se producirá un tipo de modulación diferente:

- **PWM:**

![Figura 10. Modulación PWM](image)

La modulación de ancho de pulso (PWM) de una señal es una técnica que logra producir el efecto de una señal analógica sobre una carga, a partir de la variación de la frecuencia y ciclo de trabajo de una señal digital. El ciclo de trabajo describe la cantidad de tiempo que la señal está en un estado lógico alto, como un porcentaje del tiempo total que esta toma para completar un ciclo completo. La frecuencia determina que tan rápido se completa un ciclo, y, por lo tanto, que tan rápido se cambia entre los estados lógicos alto y bajo.

En el protocolo SAE J1850 concretamente, tiempo de bit fijo, y al comienzo de cada tiempo de bit se empieza la transmisión el pulso corto o el pulso largo. Y siempre predomina el pulso largo sobre el corto. [13] [14]

- **VPW:**

![Figura 11. Modulación VPW](image)

La modulación de ancho de pulso variable (VPW), funciona como la PWM con la salvedad que va variando el ancho del pulso a medida que pasa el tiempo.

En el protocolo SAE J1850 concretamente, el bus alterna fase entre pasivo y activo para cada bit, provocando que cuando el bus está en estado activo, los pulsos largos dominen a los pulsos cortos, y cuando se da el caso de encontrarnos en estado pasivo, los pulsos cortos dominan a los pulsos largos. [13] [14]
**Capa de enlace SAE J1850**

Tanto en PWM y VPW en esta capa son iguales y no hay ninguna diferencia. Por lo cual su estructura será la misma.

La estructura en la capa de enlace se divide de la siguiente forma:

- **SOF**: Comienzo de la trama.

- **Encabezado**: 3 bits de prioridad, 1 bit de longitud de cabecera, 1 bit de respuesta en la trama (IFR), 1 bit de modo de dirección, 2 bits de tipo de mensaje y por último 2 bytes adicionales para la dirección de destino compuesta por 8 bits y la dirección de origen compuesta por 8 bits.

- **Los datos**: Son un total de siete bytes.

- **CRC**: Verificación de redundancia cíclica.

- **EOD**: Símbolo de fin de datos.

---

**ISO 9141-2**

Las características de este protocolo las vemos resumidas en las siguientes tablas:

<table>
<thead>
<tr>
<th>Característica</th>
<th>Descripción</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Línea K (bidireccional)</strong></td>
<td>Pin 7</td>
</tr>
<tr>
<td><strong>L Line (unidireccional, opcional)</strong></td>
<td>Pin 15</td>
</tr>
<tr>
<td><strong>12V</strong></td>
<td>Pin 16</td>
</tr>
<tr>
<td><strong>Tierra</strong></td>
<td>Pines 4, 5</td>
</tr>
</tbody>
</table>

**Estado del autobús:**

K Línea inactiva ALTO. El bus está activo cuando está bajo LOW.
El protocolo ISO 9141-2 nace a partir del protocolo serie asincrono (RS232), y mantiene infinidad de similitud con su predecesor (RS232). Su uso está muy extendido por Europa y Asia, además del gran uso que le da Chrysler.

El RS232 es una interfaz que designa una norma para el intercambio de datos binarios serie entre un DTE “Data Terminal Equipment”, y un DCE “Data Communication Equipment”. Un claro ejemplo de este tipo de comunicación serie es el que se produce entre una CPU de un ordenador cualquiera y un módem. Ahora bien, cuando tratamos con la ISO 9141-2, estamos hablando de que este estilo de funcionamiento (RS232) pasa a implantarse en los vehículos, encargándose la transmisión constante de los mensajes, mediante los símbolos en encabezados, CRC y datos.

ISO 9141-2 posee una velocidad para la transmisión de datos de 10.4 Kbps y su trama se compone de un máximo de 12 Bytes. La trama a nivel de enlace de este protocolo es la misma que se produce en el protocolo SAE J1850, lo cual hemos podido observar en la (Figura 12). De todas formas, en la siguiente imagen veremos la trama del protocolo ISO 9141-2, pero no sólo a nivel de enlace, sino que se mostrará desde la capa física hasta la capa de aplicación.

**Figura 13. Tramas del protocolo ISO 9141-2**
Los sistemas que operan mediante ISO 9141-2, disponen de dos cables denominados LINEA K y LINEA L, aunque esta última no es estrictamente necesaria, su uso es opcional. Estos dos cables se utilizan básicamente para la comunicación con el exterior, y el flujo de datos en un instante determinado de tiempo solamente puede discurrir en uno de los dos sentidos. A pesar de eso, se conserva la característica fundamental de posibilitar la comunicación de varios implicados a través de un mismo cable. [11]

- **Línea K**: Es un tipo de bus mono-hilo bidireccional, el cual está diseñado tanto para este protocolo ISO 9141, como para el ISO 14230-1.

- **Línea L**: Es un cable muy similar al de la línea K, pero es unidireccional, y se utiliza para el establecimiento de la comunicación con una centralita electrónica y no es soportada por todos los vehículos.

**ISO 14230-4 (KWP2000)**

Las características de este protocolo las vemos resumidas en las siguientes tablas:

<table>
<thead>
<tr>
<th>Característica</th>
<th>Descripción</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Línea K (bidireccional)</strong></td>
<td>Pin 7</td>
</tr>
<tr>
<td><strong>L Line (unidireccional, opcional)</strong></td>
<td>Pin 15</td>
</tr>
<tr>
<td><strong>12V</strong></td>
<td>Pin 16</td>
</tr>
<tr>
<td><strong>Tierra</strong></td>
<td>Pines 4, 5</td>
</tr>
</tbody>
</table>

**Estado del autobús:**
Activó cuando se conduce BAJO.

**Voltaje máximo de la señal:**
+ 12V

**Voltaje mínimo de la señal:**
0V

**Número de bytes:**
Datos: 255

**Tiempo binario:**
UART: 10400bps, 8-N-1

*Tabla 16: Características ISO 14230-4 (KWP2000)*
Este es el protocolo de palabras clave 2000, otro método de comunicación en serie asincróno al igual que el protocolo ISO 9141-2, y al igual que éste, también se ejecuta hasta una velocidad de transmisión de datos de 10,4 kbps. Además es capaz de contener hasta 255 Bytes en los campos de datos que transmite. En este estándar también se utilizan los dos cables Línea K y Línea L, de los cuales se ha hablado en el apartado anterior.

Centrándonos más en lo que sería el KWP 2000, éstas son las siglas de Key Word Protocol 2000. Con este método se consiguió la estandarización de la comunicación entre el tester del taller y las centralitas. Toda esta comunicación transcurre mediante el tan conocido y famoso principio de pregunta-respuesta, en el cual, el tester manda a la centralita una consulta definida por el protocolo (En el caso que estamos viendo el 14230-4), a la cual la ECU lanza una respuesta inmediata. Esta respuesta puede desde contener algún dato solicitado por el tester hasta tratarse de un simple mensaje de acuse de recibo. [11]

**CAN (11898/15765)**
Las características de este protocolo las vemos resumidas en las siguientes tablas:

<table>
<thead>
<tr>
<th>Característica</th>
<th>Descripción</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUDE ALTO (CAN H)</strong></td>
<td>Pin 6</td>
</tr>
<tr>
<td><strong>CAN LOW (CAN L)</strong></td>
<td>Pin 14</td>
</tr>
<tr>
<td><strong>12V</strong></td>
<td>Pin 16</td>
</tr>
<tr>
<td><strong>Tierra</strong></td>
<td>Pines 4, 5</td>
</tr>
<tr>
<td><strong>Estado del autobús:</strong></td>
<td>Activo cuando CANH tiró ALTO, CANL tiró BAJO. Inactivo cuando las señales están flotando.</td>
</tr>
<tr>
<td><strong>CANH Voltaje de la señal:</strong></td>
<td>+ 3.5V</td>
</tr>
<tr>
<td><strong>CANL Voltaje de la señal:</strong></td>
<td>+ 1.5V</td>
</tr>
<tr>
<td><strong>Voltaje máximo de la señal:</strong></td>
<td>CANH = + 4,5V, CANL = + 2,25V</td>
</tr>
<tr>
<td><strong>Voltaje mínimo de la señal:</strong></td>
<td>CANH = + 2.75V, CANL = + 0.5V</td>
</tr>
</tbody>
</table>
En este apartado, al tratarse del protocolo que se utiliza en el diseño del simulador del proyecto, será más extenso para poder entrar más en profundidad en el funcionamiento de un sistema que se rija por cualquier protocolo CAN BUS. De hecho, estamos ante un protocolo con un amplio abanico de variantes ya que goza desde hace muchos años de gran popularidad, lo que da lugar a mejoras continuas de este tipo de sistemas.

El bus CAN “Controller Area Network”, aparece de la necesidad en los años ochenta de encontrar una forma de interconectar los distintos dispositivos de un automóvil de la manera más rápida, sencilla y reduciendo de manera muy significativa las conexiones. Todos estos condicionantes acabaron dando lugar a que la empresa Robert Bosch GmbH comenzará el desarrollo de un sistema y un estándar que pudiera cumplir con todos esos condicionantes, para finalmente lograr desarrollar el bus CAN, que posteriormente se estandarizó gracias a la aparición de la norma ISO 11898-1.

Los automóviles modernos llegan a tener hasta 50 unidades de control electrónico (ECU), para infinidad de subsistemas. De los cuales, los principales son la unidad de control de motor, transmisión, airbags, frenos antibloqueo, control de crucero, ventanas, etc. Algunos de estos sistemas son independientes, pero en otros la comunicación entre ellos es de vital importancia. Es aquí donde entra en escena el CAN bus, ya que puede comunicar la unidad de control de motor y la transmisión, conectar puertas con cierres de seguridad, etc. [11]

**Tabla 17: Características CAN (11898/15765)**

<table>
<thead>
<tr>
<th>Número de bytes:</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiempo binario:</td>
<td>250kbit / seg o 500kbit / seg</td>
</tr>
</tbody>
</table>

Figura 14. Sistema CAN bus en un vehículo
El CAN-Bus de datos representa un modo de transmitir los datos entre las unidades de control (como se comenta en el párrafo anterior), por lo cual, cuanto mayor es la cantidad de información que recibe cada unidad de control acerca del estado operativo del sistema global, mejor se puede ajustar al conjunto sus funciones específicas. Es por esto que los sistemas que funcionan a partir del protocolo CAN se engloban en dos sistemas globales:

- **Área de la tracción:**
  - Unidad de control del motor.
  - Unidad de control para cambio automático.
  - Unidad de control ABS.

- **Área de confort:**
  - Unidad de control central.
  - Unidades de control de puertas.

**Características y ventajas del CAN**

Las características y ventajas que presenta un CAN bus son por un lado muy similares a la del resto de protocolos, pero cuenta con el añadido favorable de que todas tienen una complejidad bastante más reducida, y una amplia versatilidad:

- Si el protocolo de datos necesita ampliación con información suplementaria solamente se necesitan modificaciones en el software.

- Presenta un bajo porcentaje de errores mediante una verificación continua de la información transmitida, de parte de las unidades de control, y mediante protecciones adicionales en los protocolos de datos.

- Economiza el coste monetario y minimiza la complejidad del cableado, en el ámbito del sector automovilístico.

- Menos sensores y cables de señales, debido al uso múltiple de una misma señal de sensores.

- Presenta una transmisión de datos muy rápida entre las unidades de control, aunque la velocidad depende de la distancia, que es un máximo de 1000 metros.

- El espacio disponible es mucho más amplio, mediante unidades de control más pequeñas y conectores más compactos para las unidades de control.

- El CAN-Bus de datos está normalizado a nivel mundial gracias a la norma ISO 11898-1. Provocando que las unidades de control de diferentes fabricantes puedan intercambiar datos entre sí.
- Todo CAN es un sistema multimaestro donde puede haber más de un “Máster” al mismo tiempo y sobre la misma red, o lo que es lo mismo, todos los nodos de los que dispone, poseen la capacidad de transmitir, permitiendo la construcción de sistemas inteligentes y redundantes.

- Es multicast, y permite que todos los nodos puedan acceder al bus de forma simultánea con sincronización de tiempos.

**Topología en una red CAN**

Cualquier red CAN posee lo que podríamos denominar topología base, que utiliza el método de eslabón Maestro (MÁSTER) y eslabón Esclavo, a partir de la cual derivan todas las mejoras que se quieran implantar.

![Figura 15. Topología de una red CAN](image)

**Controlador CAN:** Es el receptor del microprocesador, en la unidad de control, de los datos que han de ser transmitidos. Su misión es acondicionar los datos y pasarlos al transceptor CAN. Así mismo recibe los datos procedentes del transceptor CAN, los acondiciona y los manda al microprocesador en la unidad de control.

**Transceptor CAN:** Hace las veces de transmisor y receptor. Su misión es transformar los datos del controlador CAN en señales eléctricas y transmitirlas sobre los cables del CAN-Bus. De la misma forma, recibe los datos y los transforma para el controlador CAN.

**Cables del bus de datos:** Es bidireccional y sirven para la transmisión de los datos. Se denominan con las designaciones CAN-High “señales de nivel lógico alto” y CAN-Low “señales de nivel lógico bajo”.

**Elemento final del bus de datos:** Estos elementos no son más que una resistencia, para evitar que los datos transmitidos sean devueltos en forma de eco de los extremos de los cables y que pueda provocar la falsificación de los datos obtenidos. [4] [16]
**Capas OSI en protocolo CAN: CAPA FÍSICA**

Esta capa es la encargada de definir los aspectos del medio físico para la transmisión de datos entre nodos de una red CAN, lo que implica que es la responsable de la transferencia de bits. Además, debe ser capaz de representar los estados recesivo y dominante.

La capa de nivel físico en este protocolo puede ser dividida en tres subcapas: **Physical Signaling Layer (PSL)**, **Physical Medium Attachment (PMA)** y **Medium Dependent Interface (MDI)**.

- **Physical Signaling Layer (PSL)**: Subcapa mediante la cual se realiza la sincronización de los bits y la temporización de estos mismos.

- **Physical Medium Attachment (PMA)**: Subcapa a partir de la cual se transforman los niveles lógicos de transmisión y recepción a los que vienen indicados por el protocolo CAN bus. La subcapa PMA comprende una función PMA Reset y cinco funciones operativas simultáneas y asíncronas. Las funciones de operación PMA son PHY Control, PMA Transmit, PMA Receive, Link Monitor y Clock Recovery. Todas las funciones de operación se inician inmediatamente después de finalizar con éxito la función PMA Reset.

- **Medium Dependent Interface (MDI)**: Esta es la denominada subcapa de interfaz y dependiente del medio que indica el conector y el medio de transmisión.

El protocolo CAN en esta capa emplea par trenzado como medio de transmisión, junto con la codificación de bit del tipo NRZ con bit stuffing cada 5 bits para evitar la desincronización de bit. Mediante este método, no aparecen flancos 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. Además el NRZ es un tipo de codificación muy eficiente y es bastante utilizado, ya que requiere poco ancho de banda para transmitir. [16]

Los distintos bus en la capa física pueden ser distintos en lo que a velocidad de bits respecta. Esto se debe a la codificación NRZ mencionada anteriormente, que tienen el siguiente comportamiento.

![NRZ-L-Code](image)

*Figuras 16 y 17. Ejemplo comportamiento NRZ*
Como hemos visto en las dos imágenes (Figuras 16 y 17), la codificación NRZ la podemos dividir en dos tipos de tramos, **Dominante** y **Recesivo**. Sabiendo esto es más sencillo explicar cuando estamos ante un CAN de Alta velocidad y un CAN de Baja velocidad.

La diferencia del tipo de velocidad reside que cuando nos encontramos ante una codificación que se rige bajo la norma ISO 11898-2 tendremos un CAN de Alta velocidad que tendrá entre las señales CAN-H y CAN-L una diferencia de entre 0 y 2 voltios que indicará el nivel lógico de un bit. Y tendrá el siguiente comportamiento:

![Figura 18. Codificación en NRZ a alta velocidad](image)

Y cuando nos encontramos ante una codificación que se rige bajo la norma ISO 11898-3 tendremos un CAN de Baja velocidad que tendrá entre las señales CAN-H y CAN-L una diferencia de entre 5 y 2 voltios que indicará el nivel lógico de un bit. Y tendrá el siguiente comportamiento:

![Figura 19. Codificación en NRZ a baja velocidad](image)
Existen varios estándares para la capa física de este protocolo, y se resumen en la siguiente tabla (Tabla 18):

<table>
<thead>
<tr>
<th>PROTOCOLOS CAPA FISICA DEL CAN</th>
<th>Detalles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO 11898-1</td>
<td>Capa de enlace de datos CAN y señalización física.</td>
</tr>
<tr>
<td>ISO 11898-2</td>
<td>Unidad de acceso a medios de alta velocidad CAN (utiliza un esquema de señalización balanceado de dos hilos. Aplicaciones automotrices y redes de control industrial).</td>
</tr>
<tr>
<td>ISO 11898-3</td>
<td>Interfaz de baja velocidad, tolerante a fallos y dependiente del medio.</td>
</tr>
<tr>
<td>ISO 11898-4</td>
<td>Comunicación activada por el tiempo de CAN (estándar que define la comunicación activada por tiempo en CAN (TTCAN) Está basada en el protocolo de capa de enlace de datos CAN que proporciona un reloj de sistema para la programación de mensajes).</td>
</tr>
<tr>
<td>ISO 11898-5</td>
<td>Unidad con modo de baja potencia.</td>
</tr>
<tr>
<td>ISO 11992-1</td>
<td>Tolerancia a fallas CAN para la comunicación de camión / remolque.</td>
</tr>
<tr>
<td>ISO 11783-2</td>
<td>Velocidad de 250 kbit/s. Estándar agrícola que utiliza cuatro cables trenzados no blindados; Dos para el CAN y dos para la alimentación del circuito de polarización de terminación (TBC) y tierra. Este auto-bus se utiliza en tractores agrícolas y está diseñado para proporcionar interconectividad con cualquier implementación que se adhiera a la norma.</td>
</tr>
<tr>
<td>SAE J1939-11</td>
<td>Velocidad de 250 kbit/s. Este estándar utiliza un par trenzado de dos hilos, y tiene un escudo alrededor del par. SAE 1939 es ampliamente utilizado en equipos agrícolas y de construcción.</td>
</tr>
<tr>
<td>SAE J2411</td>
<td>Single-wire CAN (SWC).</td>
</tr>
</tbody>
</table>

Tabla 18. Protocolos capa física del CAN

**Capas OSI en protocolo CAN: CAPA de Enlace**

El protocolo de acceso al medio utilizado para la capa de enlace es el CSMA/CD + AMP. Consta de un gran número de bits enlazados, y su cantidad de bits depende del tamaño del campo de datos. Mediante este método o protocolo los nodos realizan una escucha del medio, de forma que conocen en todo momento cuándo un mensaje está siendo transmitido, evitando de esta manera, mandar una trama de datos cuando la red está ocupada. En el caso de encontrarnos ante una red multimaestro, esta metodología de acceso al medio es muy importante ya que todo nodo activo tiene los derechos para controlar la red y acaparar los recursos. [16]

La trama de la capa de enlace se puede ver seccionada en sus diversos campos a continuación:
Figura 20. Trama capa de enlace del CAN

**Campo de inicio del mensaje (comienzo del datagrama):**

Marca el comienzo del protocolo de enlace de los datos. Mediante el cable CAN-High se transmite un bit con aproximadamente 5 voltios y por el cable CAN-Low se transmite un bit con aproximadamente 0 voltios.

El bit RTR es el encargado de indicar si el mensaje contiene datos, es decir RTR=0, o si tenemos una trama remota sin datos, lo que significa que RTR=1. Toda trama de datos siempre tiene una prioridad más alta que una trama remota. Y la trama remota se emplea para la solicitud de datos a otras unidades de mando, ya sea por la necesidad o por un chequeo.

**Campo de estado:**

Su función exclusiva es definir la prioridad del protocolo. Lo que implica que, si hay dos unidades de control intentando transmitir simultáneamente su protocolo de datos, se concede la preferencia al protocolo que posea una prioridad superior.

**Campo de control:**

Campo con gran relevancia ya que especifica la cantidad de información que está contenida en el campo de datos. Esto da lugar a que cada receptor pueda revisar si ha recibido la información completa. Los cuatro bits que componen el campo DLC indican el número de bytes contenidos en el campo de datos.

La diferencia entre una trama estándar y una trama extendida en el campo de control, es que la primera dispone 11 bits y la segunda aumenta hasta los 29 bits.

- **Trama Estándar:** Tiene un identificador de 11 bits. Mediante este identificador se establece la prioridad de los diferentes nodos de la red.

- **Trama Extendida:** Tiene un identificador que se extiende hasta 29 bits, e incluye dos bits adicionales, denominados SSR (Recesivo) y RB1 (Dominante). El SSR se encuentra al final de los 11 primeros bits del identificador, y el RB1 se sitúa antes que el bit RB0. Esta tipología de tramas no se utiliza en redes que requieren de un desempeño muy elevado.
Ambas tramas tienen la capacidad de coexistir de forma eventual, gracias a existencia de dos versiones de CAN.

**Campo de datos:**
Es un campo muy simple, ya que simplemente transmite la información para las demás unidades de control. Puede contener entre 0 y 8 bytes (de 0 a 64 bit), y la información es transmitida a lo largo de la línea CAN-Bus.

**Campo de aseguramiento (CRC):**
Se encarga de la detección de fallos durante la transmisión del mensaje. Posee una longitud de 16 bits y se delimita con un bit final en estado recesivo, lo que implica que en realidad tiene 15 bits más el bit en estado recesivo.

**Campo de confirmación (ACK):**
El campo de confirmación, conocido bajo las siglas ACK, está compuesto por dos bits que siempre se trasmiten como recesivos “bit 1”. Todas las unidades de mando que reciben el mismo CRC (campo inmediatamente anterior a este) modifican el primer bit del campo ACK por uno dominante “bit 0”, de forma que la unidad de mando que está todavía trasmitiendo reconoce que al menos alguna unidad de mando ha recibido un mensaje escrito de forma correcta. Si esto no se produjese, la unidad de mando transmisora interpretaría que el mensaje presenta algún tipo de error.

**Campo de final de mensaje (fin del datagrama) (EOF):**
Serie de 7 bits recesivos que indican el final de la trama, y por esto mismo es la última oportunidad que existe para localizar y dar el aviso de un error para corregirlo y proceder a una repetición del mensaje. [2]

![Figura 21. Ejemplo trama](image)

La capa de enlace de un protocolo como el CAN no solo se queda en la estructura descrita hasta ahora, sino que también podemos dividir esta capa en dos subcapas, las cuales se reparten las diversas funciones que se efectúan a nivel de enlace. Estas subcapas son las siguientes:

- **Subcapa MAC (Medium Access Channel):** Es el núcleo del protocolo CAN. Presenta los mensajes recibidos a la subcapa LLC y acepta los mensajes para transmitirlos a esa subcapa. También es la responsable del mecanismo de arbitraje de acceso al medio. Utiliza la técnica de acceso al medio denominada como CSMA/CD+CR “Carrier Sense Multiple Access/Collision Detection + Collision Resolution”.

- **Subcapa LLC (Logical Link Control):** 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:

- **Servicio de transmisión de datos sin reconocimiento**: Su función es la de aportar al usuario LLC los medios para poder 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 remoto sin reconocimiento**: Su función es la de aportar al usuario LLC los medios necesarios para solicitar que un nodo remoto transmita sus LDSU sin establecer una conexión de enlace de datos.

**Capas OSI en protocolo CAN: CAPAS Intermedias y la de Aplicación**

Por último, en lo que respecta a las capas de un protocolo CAN pasamos a la capa de aplicación, saltándonos las capas intermedias entre la de enlace y esta, ya que el protocolo CAN afecta únicamente a la capa física y de enlace, y ligeramente a la de aplicación. Por lo cual el resto no se ven afectados por este protocolo.

Si queremos mencionar algo relacionado con las capas intermedias cabe destacar el protocolo ISO 15765-2, conocido como ISO-TP, el cual corresponde a los niveles 3 de red y 4 de transporte. Y su función es la segmentación de los mensajes largos en varias tramas CAN.

Hablando ya de la capa de aplicación, se dispone de un amplio abanico de estándares que definien la capa de aplicación; algunos son muy específicos lo que provoca que estén estrictamente vinculados a sus campos de aplicación.

El HLP “**Higher Layer Protocols**” más destacado en lo que a capa de aplicación respecta es el CAN Open que es muy utilizado no solo en el sector automovilístico, sino también en el ámbito del equipamiento médico, transporte público, etc. [16]

Otros HLP muy utilizados y que merecen la pena mencionar para saber de su existencia son las siguientes:

- **CAL**
- **Device Net**
- **SDS (Smart Distributed System)**
- **OSEK**
- **CAN Kingdom**
**Temporización en un CAN-Bus**

Las señales eléctricas del bus sufren las alteraciones propias de cualquier distorsión que se produce, por causas de las características físicas de la línea, los retardos de propagación, y los retardos producidos en los propios controladores, además de por las posibles fuentes externas de interferencia electromagnética.

Cada controlador CAN depende de un oscilador distinto, lo que provoca diferencias de frecuencia que pueden dar lugar a desfases en el muestreo de tramas de los distintos nodos. Por todo esto, se ha tomado finalmente la medida de que los controladores sigan un proceso de muestreo y resincronización orientado a evitar los desajustes provocados por las circunstancias adversas.

Todo esto se aborda en parte gracias a la codificación NRZ de la que se ha hablado en el apartado de la capa física del CAN. Pero aparte de la codificación también es necesario utilizar una frecuencia de reloj, ya que una señal de reloj como tal no existe la posibilidad de incluirla. Debido a este problema con la señal de reloj, la que denominaremos frecuencia de reloj será múltiplo de la frecuencia nominal de bit. Esta frecuencia se obtiene como cociente de la frecuencia del oscilador. La velocidad de comunicación a la que queda configurado el controlador, para un oscilador de cierta frecuencia, queda definida por el valor del divisor de frecuencia (BRP), que también es configurable, y por el número de ‘TQ’ por bit. Este TQ (quantum de tiempo) es una unidad de tiempo fija, derivada del periodo del oscilador. [1] [16]

La unidad de tiempo fija está definida por la siguiente expresión:

\[
TQ = \frac{2 \times BRP}{F_{osc}}
\]

**Resincronismo en un CAN-Bus**

La resincronización consiste en el acortamiento o alargamiento del tiempo de bit para que el punto de muestreo se desplace con relación al último flanco recesivo-dominante detectado. La decisión de esta acción es propiedad del controlador tras detectar el flanco y evaluar el error.

El tiempo de bit se estructura en los siguientes cuatro segmentos:

![Diagrama de resincronismo](image)

**Figura 22. Resincronismo**
• **Segmento de sincronización (Sync):** Su función consiste en la espera de que se produzca cualquier cambio de recesivo a dominante en el bus. Con una duración de 1 TQ.

• **Segmento de propagación (Prop. Segment):** Encargado de la compensación del retardo de tiempo de la red. Este retardo es el que se ha generado por la suma del provocado por la línea y el que se produce por la electrónica de la interfaz de los controladores. El segmento puede configurarse de 1 a 8 TQ.

• **Segmento de fase 1 (Phase Segment 1):** Encargado de detectar y reajustar los desfases entre nodos. Puede configurarse con una duración de 1 a 8 TQ, pero existe la posibilidad de alargarlo en operaciones de resincronización.

• **Segmento de fase 2 (Phase Segment 2):** Retrasa la siguiente transición de datos a transmitir. Su valor máximo puede ser igual al segmento de fase 1 más grande o al tiempo de procesado de la información. Nos encontramos ante un segmento con la capacidad de ser programable desde 1TQ hasta 8 TQ. Aunque se puede recortar las operaciones de resincronización. [1] [16]
Desarrollo del Proyecto: Hardware

A partir de este apartado se explicará todo lo relacionado para llevar a cabo el diseño del simulador de la ECU que se ha fabricado para este proyecto. En este primer gran apartado se analizará todo lo relacionado con el hardware, en el siguiente todo lo que tenga relación con el software y por último lo que abarca a las mediciones realizadas.

En este punto se estudiará todo lo relativo a los componentes que forman la parte de hardware del simulador, tanto los componentes para el montaje del simulador y toda su circuitería, el dispositivo de medición y el dispositivo que recibirá y mostrará la información proporcionada por la ECU diseñada.

Para el simulador se utiliza un circuito compuesto por resistencias, potenciómetros y un LED controlado por un pulsador. Todo este circuito está conectado a una placa CAN-BUS Shield que está a su vez comunicada mediante una interfaz SPI con una placa de Arduino UNO que es en la cual se implementa el código o programa que permitirá el funcionamiento de todo el sistema de dicha ECU.

Y para la medición y muestra de resultados se utilizará un ELM 327, que es un conector OBD-II que recogerá la información en tiempo real desde el controlador CAN de la placa CAN-BUS Shield. Dicha información será enviada a través de una toma USB a un ordenador, que con el programa correspondiente (ScanMaster) mostrará en pantalla los resultados.
Sistema Arduino

En primer lugar, hay que explicar qué es y para qué se utiliza Arduino. Es una plataforma de código abierto utilizada para la construcción de proyectos electrónicos. Arduino consta de una placa de circuito programable física (a menudo denominada microcontrolador) y una pieza de software, o IDE (Integrated Development Environment) que se ejecuta en la computadora, utilizada para escribir y cargar el código de la computadora en la placa física.

La plataforma Arduino se ha vuelto muy popular entre la gente que acaba de comenzar con la electrónica, y por una buena razón. A diferencia de la mayoría de las placas de circuitos programables anteriores, el Arduino no necesita una pieza separada de hardware (llamada programador) para cargar el nuevo código en la placa, simplemente puede usar un cable USB. Además, el IDE de Arduino utiliza una versión simplificada de C++, por lo que es más fácil aprender a programar. Por último, Arduino proporciona un factor de forma estándar que rompe las funciones del microcontrolador en un paquete más accesible.

El hardware y software de Arduino es diseñado por artistas, diseñadores, aficionados, hackers, novatos y cualquier persona interesada en crear objetos o entornos interactivos. Arduino puede interactuar con botones, LEDs, motores, altavoces, unidades GPS, cámaras, Internet e incluso con teléfonos inteligentes (Smartphones) o con telescopios y vehículos. Esta flexibilidad combinada con el hecho de que el software Arduino es totalmente gratuito, las placas de hardware son bastante baratas, y tanto el software como el hardware son fáciles de aprender, ha conseguido atraer una gran comunidad de usuarios que han contribuido con código e instrucciones para llegar a obtener una gran variedad de proyectos basados en Arduino.

Arduino también dispone de otra gran ventaja con respecto a otros tipos de plataformas microcontroladoras y lenguajes, que junto a lo anteriormente mencionado es una de las grandes causantes de haber seleccionado esta plataforma para utilizarla en este proyecto, y es que es Multiplataforma. Este software tiene la capacidad de ser compatible para prácticamente todos los sistemas operativos, como pueden ser Windows, Linux o Macintosh OS X.

También es importante destacar que el catalogo de librerías necesarias para un proyecto de estas características, es mucho más amplio en Arduino que en otras plataformas muy utilizadas como puede ser el caso de Raspberry Pi. Y esto ha permitido que se pueda escoger la librería más adecuada a las necesidades que se buscaban cubrir.

En la actualidad existen bastantes modelos de placas Arduino, buscando cubrir las distintas necesidades que tienen los usuarios. Algunas de estas placas son más potentes, otras pueden conectarse a una red Ethernet, y otros simplemente cuentan con una mayor cantidad de pines de un tipo u otro (analógicos, digitales, etc).

A continuación, en la siguiente tabla (Tabla 23) se resumen brevemente los diferentes modelos de placas Arduino que existen en el mercado, con sus características básicas y para qué su utiliza cada una. De esta manera se puede llegar a comprender con mayor facilidad hasta dónde es capaz de llegar una plataforma de software abierto y hardware de bajo coste, como es el caso de esta misma.
<table>
<thead>
<tr>
<th>TIPO DE ARDUINO</th>
<th>PLACA</th>
<th>CARACTERISTICAS</th>
</tr>
</thead>
</table>
| **UNO**        | ![Arduino Uno](image) | - Microcontrolador ATmega328.  
- Voltaje de entrada 7-12V.  
- 14 pines digitales de I/O (6 salidas PWM).  
- 6 entradas análogas.  
- 32k de memoria Flash.  
- Reloj de 16MHz de velocidad. |
| **MEGA**       | ![Arduino MEGA](image) | - Microcontrolador ATmega2560.  
- Voltaje de entrada de –7-12V.  
- 54 pines digitales de Entrada/Salida (14 de ellos son salidas PWM).  
- 16 entradas análogas.  
- 256k de memoria flash.  
- Velocidad del reloj de 16Mhz. |
| **DUE**        | ![Arduino DUE](image) | - Microcontrolador: AT91SAM3X8E.  
- Voltaje de operación: 3.3V.  
- Voltaje recomendado de entrada (pin Vin): 7-12V.  
- Pines de entrada y salida digitales: 54 pines I/O, de los cuales 12 proveen salida PWM.  
- Pines de entrada análogos: 12.  
- Pines de salida análogos: 2.  
- Corriente de salida total en los pines I/O: 130mA.  
- Corriente DC máxima en el pin de 3.3V: 800mA.  
- Corriente DC máxima en el pin de 5V: 800mA.  
- Memoria Flash: 512 KB toda disponible para aplicaciones del usuario.  
- SRAM: 96 KB (en dos bancos de: 64KB y 32KB).  
- Velocidad de reloj: 84 MHz |
| ETHERNET | **Microcontrolador:** ATmega328.  
**Voltaje de operación:** 5V.  
**Voltaje de salida de conexión (recomendada):** 7-12V.  
**Voltaje de salida de conexión (límites):** 6-20V.  
**Pines de entrada/salida digitales:** 14 (4 de los cuales proveen salida de PWM).  
**Pines Arduino reservados:**  
- 10 a 13 usados para SPI.  
- 4 usados para tarjeta SD.  
- 2 de interrupción W 5100 (cuando está en puente).  
**Pines de salida análogos:** 6.  
**Corriente DC por cada pin de entrada/salida:** 40mA.  
**Corriente DC para pines de 3.3V:** 50mA.  
**Memoria flash:** 32Kb (ATmega328) de los cuales 0.5Kb son usados por el gestor de arranque.  
**SRAM:** 2Kb (ATmega328).  
**EEPROM:** 1Kb (ATmega328).  
**Velocidad de reloj:** 16MHz.  
**Controlador Ethernet integrado W 5100 TCP/IP.**  
**Tarjeta Micro SD, con transistores de voltaje activos.** |
| MEGA ADK | **El ADK se basa en el Mega 2560.**  
**Como en el Mega 2560 y Uno, cuenta con un Atmega8U2 programado como un convertidor USB a serie.** |
| LEONARDO | **Microcontrolador Atmega32u4.**  
**Voltaje de entrada:** 7-12V.  
**Voltaje de trabajo:** 5V.  
**Corriente por pin I/O:** 40mA.  
**20 pines digitales I/O.**  
**7 canales PWM.**  
**12 ADC.**  
**16MHz de velocidad de reloj.**  
**Memoria Flash:** 32 KB (ATmega32u4) de los cuales 4 KB son usados por el bootloader.  
**Memoria SRAM:** 2.5 KB (ATmega32u4).  
**Memoria EEPROM:** 1KB. |
<table>
<thead>
<tr>
<th><strong>PRO</strong></th>
<th><strong>LILYPAD</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• ATmega328 funcionando a 8 MHz resonador externo</td>
<td>• Construida con un microcontrolador ATmega 328, que a su vez posee un gestor de arranque Arduino (bootloader).</td>
</tr>
<tr>
<td>• tablero de baja tensión no necesita circuitos de interfaz con dispositivos de 3.3V populares y módulos (GPS, acelerómetros, sensores, etc.)</td>
<td>• Cuenta con un número mínimo de componentes para mantener un tamaño reducido.</td>
</tr>
<tr>
<td>• conexión USB fuera de bordo</td>
<td>• Funciona desde 2 a 5V.</td>
</tr>
<tr>
<td>• regulador de 3.3V</td>
<td>• Presenta un reseteo automático para una programación aún más fácil.</td>
</tr>
<tr>
<td>• Salida máxima de 150 mA</td>
<td>• 50 mm de diámetro externo.</td>
</tr>
<tr>
<td>• Durante protegida actual</td>
<td>• PCB con 0.8mm de espesor.</td>
</tr>
<tr>
<td>• polaridad inversa protegida</td>
<td><strong>PRO Mini</strong></td>
</tr>
<tr>
<td>• DC de entrada 3.3V hasta 12V</td>
<td>• Microcontrolador ATmega328 funcionando a 16MHz con cristal externo (0.5% tolerancia)</td>
</tr>
<tr>
<td>• fusible reajustable evita daños a bordo en caso de corto</td>
<td>• Esta tarjeta no posee conector USB.</td>
</tr>
<tr>
<td>• La energía del interruptor de selección actúa como interruptor de encendido / apagado.</td>
<td>• Soporta auto-reset para carga de Sketch con FTDI.</td>
</tr>
</tbody>
</table>

**Tabla 19. Modelos Arduino**
Tras conocer de forma un poco genérica todos los modelos de Arduino, ahora se explica en profundidad la placa Arduino UNO que como bien se ha dicho anteriormente es la que se ha escogido para este proyecto.

**Arduino UNO**

El modelo Arduino UNO se puede considerar el modelo base de todas las placas Arduino, ya que es la que tiene el manejo más simple y además el resto de placas derivan sus características a partir de esta.

Las placas Arduino UNO poseen 6 PINES Analógicos que funcionan como entradas, y un total de 14 PINES Digitales, los cuales pueden utilizarse tanto para salidas como para entradas. Dentro de los pines digitales, 6 de ellos tienen la capacidad de utilizarse como salidas PWM. Estas placas como vemos en la figura anterior (Figura 23), también cuentan con un cristal de cuarzo de 16 MHz, cabecera ICSP, un botón de reinicio, una conexión USB, un conector para fuente de alimentación externa.

Las funciones de los pines de Arduino UNO son:

- **GND**: Hay varios pines GND en el Arduino, cualquiera de los cuales se puede utilizar para conectar a tierra el circuito.

- **5V & 3.3V**: El pin 5V suministra 5 voltios de la energía, y el pin 3.3V suministra 3.3 voltios de energía. La mayoría de los componentes simples utilizados con el Arduino funcionan con 5 o 3.3 voltios.

- **Analógicos**: Los pines analógicos son del A0 a A5. Estos pines pueden leer la señal de un sensor analógico (como un sensor de temperatura) y convertirlo en un valor digital que podemos leer.
- **Digital**: Los pines digitales son del 0 al 13. Estos pines se pueden utilizar tanto para la entrada digital (como indicar si se pulsa un botón) como para la salida digital (como encender un LED).

- **PWM**: Son los pines digitales 3, 5, 6, 9, 10 y 11. Dichos pines actúan como digitales normales, pero también pueden usarse para modulación de Ancho de Pulso (PWM). Estos pines son capaces de simular la salida analógica (como el desvanecimiento de un LED de entrada y salida).

- **AREF**: Soportes para referencia analógica. A veces se utilizan para establecer una tensión de referencia externa (entre 0 y 5 voltios) como el límite superior para los pines de entrada analógica.

- **SPI**: Los pines 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). Soportan la comunicación SPI haciendo uso de la librería SPY.

- **LED**: Es el pin 13, y en él hay un LED empotrado conectado al pin digital 13. Cuando el pin está a valor HIGH, el LED se enciende, y cuando el pin está en valor LOW, se apaga.

- **SERIALES TX y RX**: TX es para transmitir y RX para recibir. Estas marcas aparecen para indicar los pines responsables de la comunicación en serie. TX y RX son los pines 1 y 0 respectivamente.

- **Interrupciones externas**: Son los pines 2 y 3. Y se pueden configurados para disparar una interrupción.

El **Arduino UNO** para funcionar no necesita grades requisitos únicamente es necesario conectarlo a un ordenador con un cable USB o con un adaptador AC-DC a una toma de corriente o batería.

Estas placas soportan unos voltajes límites de entre 6 y 20 V, aunque el fabricante recomienda su utilización entre los 7 y los 12 V para trabajar con cierto margen de seguridad. Además, también hay que resaltar que el voltaje con el que opera la placa, es decir, el que la placa suministra a un circuito a través del pin correspondiente, es de 5 V.

La base del funcionamiento de **Arduino UNO** es su microcontrolador, ya que por muchas atribuciones que le podamos dar, no deja de ser una placa microcontroladora. Y el microcontrolador que sustenta el funcionamiento de la placa es el **ATmega328P**.

![Figura 24. ATmega328P](image-url)
El microcontrolador ATmega328P es un AVR fabricado por la empresa ATMEL, la cual es una diseñadora y fabricante estadounidense de semiconductores y destaca en el ámbito de la microelectrónica y la programación.

El ATmega328P posee una arquitectura RISC avanzada:

- 131 instrucciones de gran alcance.
- Ejecución de ciclo de reloj más simple.
- 32 x 8 registros de trabajo de propósito general.
- Operación completamente estática.
- Hasta 20 MIPS de Rendimiento a 20MHz.
- Multiplicador On-chip de 2 ciclos.

También posee segmentos de memoria no volátiles de alta resistencia:

- 32KBytes de Memoria de programa Flash auto-programable en el sistema.
- Hasta 1KBytes EEPROM.
- 2KBytes SRAM interno.
- Ciclos de escritura / borrado: 10.000 Flash / 100.000 EEPROM.
- Retención de datos: 20 años a 85 °C / 100 años a 25 °C.
- Sección de código de inicio opcional con bits de bloqueo independientes.

Otras características resaltables que tiene el ATmega328P son: Interfaz serie SPI maestro / esclavo, dispone de un temporizador de vigilancia programable con un oscilador de chip integrado, comparador Analógico On-chip, interrupción y activación del cambio de pin, tiene restablecimiento de encendido y detección de apagado programable, y por último, posee seis modos de reposo (Inactivo, reducción de ruido ADC, ahorro de energía, apagado, espera y espera extendida).

**Placa CAN-BUS Shield para Arduino**

CAN-Bus es un bus industrial común, como se ha visto anteriormente, gracias a la capacidad de viajar grandes distancias, una velocidad media de comunicación que es óptima para la mayor parte de los dispositivos, y haber demostrado una alta fiabilidad. Y como se dijo al comienzo de este proyecto, se encuentra comúnmente en las máquinas y herramientas modernas, tales como un autobús de diagnóstico del automóvil (que es el caso que nos atañe).

Como en este proyecto se utiliza como plataforma de microcontrolador un Arduino UNO, se necesitaba una placa CAN-Bus compatible con esta plataforma. Finalmente se ha optado por una placa CAN BUS Shield V1.2.

![Figura 26. CAN BUS Shield V1.2](image)

Las shields, como es el caso de la placa CAN BUS Shield V1.2, son placas de circuitos modulares que se montan unas encima de otras para dar funcionalidad extra a un Arduino.

Estas placas shields se pueden comunicar con el Arduino ya sea por los pines digitales o los analógicos o bien por algún bus como el SPI, I2C o puerto serie, también se pueden usar otros pines como interrupción. Además, estas shields se alimentan generalmente a través del Arduino mediante los pines de 5V y GND.
Otras características que posee la placa utilizada son: Alcanza velocidades de hasta 1 Mbps en la implementación CAN 2.0B, soporta tramas de 11 y 29 bits, tiene una interfaz SPI de hasta 10MHz, posee indicadores LED y tiene dos buffers de recepción con almacenamiento de mensajes con prioridad.

Las placas CAN BUS Shield V1.2 se comercializan sin soldar las patillas, lo que implica que se deben soldar por el comprador (procedimiento que se ha tenido que realizar en este proyecto mediante una soldadura con estaño).

![Patillas CAN BUS Shield V1.2](image)

Figura 27. Patillas CAN BUS Shield V1.2

Al seleccionar esta placa para la conexión CAN, estamos ante un escudo CAN-BUS que utiliza el controlador de bus **CAN MCP2515** con interfaz SPI y el transceptor **CAN MCP2551**. Con todo esto y un cable convertidor OBD-II junto con la biblioteca OBD-II importada, está listo para construir un dispositivo de diagnóstico a bordo.

En la siguiente figura (Figura 28) se puede ver los distintos componentes que formas el CAN BUS Shield V1.2:

![Esquema CAN BUS Shield V1.2](image)

Figura 28. Esquema CAN BUS Shield V1.2

1. **Interfaz DB9**: Para poder conectar a la interfaz OBDII a través de un cable DBG-OBD.
2. **V_OBD**: Obtiene potencia de interfaz OBDII.
3. **Indicador Led:**
   - **PWR:** Potencia.
   - **TX:** Parpadea cuando se envían los datos.
   - **RX:** Parpadea cuando hay recepción de datos.
   - **INT:** Interrupción de datos.

4. **Terminales CANH y CAN L**
5. **Arduino UNO pin de salida**
6. **Conector Serial Grove**
7. **Conector I2C Grove**
8. **Pines del ICSP**
9. **IC - MCP2551:** Transceptor CAN de alta velocidad.
10. **IC - MCP2515:** Controlador CAN autónomo con interfaz SPI.

Ahora que ya se comprende mejor la estructura de este dispositivo, es el momento de explicar brevemente el funcionamiento tanto del transceptor como del controlador para finalizar la explicación del dispositivo:

- **MCP2551:** Es un dispositivo CAN de alta velocidad, tolerante a fallos que sirve como interfaz entre un controlador de protocolo CAN y el bus físico. Este tipo de dispositivo MCP2551 proporciona una capacidad de transmisión y recepción diferencial para el controlador de protocolo CAN y es totalmente compatible con la norma ISO-11898, incluidos los requisitos de 24V, y funciona a velocidades de hasta 1Mb/s. Normalmente, cada nodo en un sistema CAN debe tener un dispositivo para convertir las señales digitales generadas por un controlador CAN en señales adecuadas para la transmisión a través del cableado del bus. También proporciona un búfer entre el controlador CAN y los picos de alta tensión que pueden generarse en el bus CAN por fuentes externas (EMI, ESD, transitorios eléctricos, etc.).

- **MCP2515:** Es un controlador autónomo CAN. Es capaz de transmitir y recibir tramas de datos en formato estándar y extendido y tramas remotas. El MCP2515 cuenta con dos máscaras de aceptación y seis filtros de aceptación que se utilizan para descartar mensajes no deseados, lo que reduce la sobrecarga del microcontrolador anfitrión. El MCP2515 se conecta con los microcontroladores mediante una interfaz de periféricos serie (SPI) de estándar industrial.
Dispositivo de diagnóstico

Los dispositivos descritos a continuación dentro de este apartado son los que se han utilizado para recoger los datos y analizarlos.

Hay que mencionar que los dispositivos usados son los que más se adaptan a las características buscadas sin generar un elevado coste.

ELM 327

El dispositivo ELM327 se trata de un conector OBD-II cuya función es realizar una diagnosis completa del vehículo y visionar todo tipo de errores registrados en la centralita. El OBD es un programa instalado en las unidades de mando del motor. Por lo cual, su función es vigilar continuamente los componentes que intervienen en las emisiones de escape, y en el momento en que se produce un fallo, el OBD lo detecta, se carga en la memoria y avisa al conductor mediante un testigo luminoso.

Generalmente la ubicación del conector OBD-II en nuestro vehículo, suele encontrarse en la zona de los pies del conductor, consola central e incluso debajo del asiento del copiloto. Pero en el caso de este proyecto al no disponer de una toma hembra de un conector OBD-II, solo se realizará la conexión de los pines necesarios a través de cables individuales.

![ELM 327](image)

Figura 31. ELM 327

El ELM 327 mediante un testigo luminoso, dispone de un código de señales luminosas explicado en el aparatado Funcionamiento OBD-II. Y mediante este código se avisa al conductor de un posible fallo en el sistema del vehículo. Este dispositivo que se ha elegido es el más utilizado por todo tipo de usuarios ya que todos los protocolos que se han descrito con anterioridad son soportados por el ELM 327, facilitando así el trabajo de los usuarios al no tener problemas de compatibilidad.

Existen varios modelos de OBD ELM 27, pero los más utilizados son dos:

- **ELM 327 con cable USB**: Es el que he seleccionado para utilizar, ya que es el más simple. El ELM 327 recoge la información y la transmite a partir del cable y el puerto USB hasta el dispositivo dotado con el programa de diagnosis necesario (en nuestro caso el ScanMaster) para poder ver los datos y realizar los test correspondientes.
- **ELM 327 Bluetooth:** Este modelo es más útil para dispositivos Android portátiles como Tablet y Smartphones. Aunque con algunos programas de diagnosis tiene problemas de compatibilidad y reconocimiento de protocolos.

El mapa de pines del OBD-II es el mismo que el de cualquier otro tipo de dispositivo OBD, pero lo se vuelve a mostrar para recordarlo:

![Mapa pines ELM 327](image)

**Figura 32. Mapa pines ELM 327**

**Ordenador SONY VAIO**

Los resultados obtenidos a partir del ELM 327 se mostrarán gracias al programa de diagnosis ScanMaster, que se instalará en un ordenador Sony Vaio.

El ordenador Sony Vaio que se ha utilizado es del modelo VPCEC1Z1E, y su procesador es un Intel Core i5 520M. El sistema operativo que utiliza el dispositivo es el Windows 7 Premium Home, lo cual es importante porque el programa para mostrar los resultados de los test no es compatible con todos los sistemas operativos, solo lo es con Windows 7, Windows 8 y Macintosh. Además la importancia del sistema operativo no se reduce solo a la compatibilidad con el programa, sino que también a la facilidad o dificultad de la instalación de los drivers del conector OBD (ELM 327) que se explicará en la parte de software.

![Ordenador Sony Vaio](image)

**Figura 33. Ordenador Sony Vaio**
Circuito y Montaje

En este punto se redacta la circuitería y las conexiones que se han realizado para la fabricación de la parte de hardware del simulador de la ECU que se quiere conseguir.

El circuito que se monta en la placa base es un circuito bastante simple, compuesto por un pulsador, dos potenciómetros, resistencias y un diodo LED. Todo este montaje viene a simular un sistema de sensores como los que se puede encontrar en cualquier tipo de vehículo. En la imagen que se ve a continuación se puede observar el circuito montado en la placa (lo que vendría a ser el sistema de sensores), a falta solo de las conexiones que irán a la placa CAN-Bus Shield. El circuito completo se verá más adelante.

Cada componente del circuito de sensores representa un elemento dentro de un vehículo.

Por un lado están los dos potenciómetros en los cuales cada uno tiene atribuciones totalmente distintas debido a que uno de ellos representa el nivel de carga que hay en cada momento, en lo que vendría a ser la batería del vehículo. Este nivel de batería depende de la posición en la que se encuentra el potenciómetro, variando dicho nivel a un valor mayor o menor a medida que se modifica la posición del potenciómetro hacia un lado u otro. El segundo potenciómetro, por su parte, vendría a representar el acelerador del automóvil, ya que variando la posición de un lado a otro la aceleración aumentará o se reducirá progresivamente. El correcto funcionamiento del sensor que controla la aceleración es esencial ya que a partir de esta se obtendrán las revoluciones por minuto (rpm) del supuesto motor y la supuesta velocidad a la que se desplazaría el automóvil, aplicando las formulas correspondiente en el apartado del código donde se desempeña esta función.

Por el otro lado encontramos el pulsador de cuatro pines y el diodo LED, estos dos elementos están directamente relacionados dentro del circuito debido a que se podría interpretar como que el pulsador imparte órdenes al LED. El pulsador en el circuito montado tiene la función de simular que se está produciendo algún tipo de error en el sistema, y esto se consigue mediante un sistema de interrupciones en el cual cada vez que se pulse dicho pulsador se encenderá el diodo LED que ejerce de indicador luminoso MIL (Indicador de avería). Lo que ocurre cuando el LED luce, es que dicho LED se encuentra en estado HIGH (se encuentra en Alto), y cuando el
ELM 327 detecta este estado a través de su conexión con la placa CAN-Bus Shield, lo interpreta como que la ECU le está avisando que existe un error.

Una vez explicados los distintos componentes, ya se puede ver en la siguiente imagen como queda el circuito completo, es decir, conectándolo con la placa shield.

A continuación en la siguiente figura (Figura 36) se muestra el esquema de conexión de la imagen anterior (Figura 35), donde se observa con claridad como las patillas centrales de los potenciómetros se conectan a los pines A0 y A1, que actúan como entradas en este caso. El LED se conecta al pin digital 13 que es un pin especialmente diseñado para la conexión de diodos LED. El pulsador está conectado al pin digital 2, el cual en el Arduino UNO está habilitado para realizar interrupciones junto con el pin 3. Y por último, mencionar también que la placa introduce 5V al circuito por el pin 5V y también proporciona la tierra al mismo por el pin GND.
El circuito al estar conectado a la placa CAN-Bus Shield V1.2 también lo está al Arduino UNO, debido a que ambas placas están conectadas entre sí gracias a que los pines de la placa shield están adaptados para poderla superponer sobre una placa de Arduino UNO. Por este hecho la alimentación que recibe el circuito es enviada a través del Arduino UNO (aunque si es necesario se le puede introducir alimentación externa), que se alimenta por el puerto USB del portátil gracias a un cable tipo A/B que además se utiliza para cargar las instrucciones del código en la placa.

Figura 37. Esquema simulador ECU 2

Para tomar los datos y realizar el análisis del simulador de la ECU que ha sido diseñada es necesario conectar el pin 6 del ELM 327 con el puerto CAN H del CAN-Bus Shield V1.2 y el pin 14 con el puerto CAN L, ya que será esta conexión la que se encargue de la transmisión de todos los datos que registren. Además no hay que olvidar que es necesario conectar el pin 4 o el pin 5 a tierra, porque de no ser así la interfaz notificará al programa de diagnosis que no se detecta ninguna ECU conectada.

Toda la información recogida junto con los códigos de fallas por el conector OBD son transmitidos al ordenador por el puerto USB mediante el cable que conecta dicho puerto con la interfaz del ELM 327. Si toda la conexión se encuentra en el estado descrito cuando se ejecute el programa de diagnosis, este mostrará que detecta tanto el dispositivo de recogida de datos, como un protocolo que sea compatible con la interfaz del programa.
Desarrollo del Proyecto: Software

En esta parte se va a tratar toda la parte relacionada directamente con el software que es una parte esencial para el funcionamiento de dispositivos de estas características.

Con este apartado vamos a conocer cómo es el funcionamiento del entorno de Arduino (IDE), cómo se realiza su programación y cómo funcionan sus funciones y librerías, principalmente y ante todo aquellas a las que se les da uso en el en código de funcionamiento del simulador ECU. También gracias a este apartado se podrá conocer el funcionamiento del programa de diagnosis ScanMaster, para saber todas las herramientas que nos puede aportar independientemente de que la necesitemos o no.

En principio el objetivo que se busca en este aparto, es comprender cómo la ECU realiza sus funciones y cómo tiene la capacidad de interpretar y responder a las diversas peticiones que se realizan a través del conector OBD (ELM 327). Y a esto hay que sumarle que hay que entender cómo el dispositivo es capaz de reconocer distintos tipo de datos, porque no hay que olvidar en este proyecto al igual que en otros muchos tipos de desarrollos mediante la plataforma Arduino, cuenta con entradas y salidas tanto digitales como analógicas.

Lo más importante de este apartado y probablemente de todo el proyecto, debido a que es la base del funcionamiento de todo el sistema diseñado, va a ser la explicación de las funciones que se han construido para este proyecto. Se explicará su código y se mostraran los diagramas de flujo para entender mejor su comportamiento.
Entorno de programación de Arduino

El entorno de desarrollo integrado también llamado IDE “Integrated Development Environment”, es un programa informático que se compone de un amplio conjunto de herramientas de programación. Puede dedicarse en exclusiva a un solo lenguaje de programación o bien puede utilizarse para varios lenguajes, gracias a la alta versatilidad que presenta.

Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación, lo que viene siendo un editor de código, un compilador, un depurador y un constructor de interfaz gráfica. Es destacable que Arduino incorpore las herramientas para cargar cualquier clase de programa ya compilado en la memoria flash del hardware.

El entorno de Arduino puede descargar de la página web oficial de Arduino de forma totalmente gratuita. En la web se dispone de versiones para Windows y para MAC así como las fuentes necesarias para compilarlas en LINUX.

Por defecto los drivers de las placas Arduino se descargan automáticamente en Windows, pero puede ocurrir que esto no ocurra tal y como me ocurrió al conectar la placa por primera vez. Por lo cual se explica brevemente cómo he realizado la instalación manual de los drivers, a continuación:

1. Para instalar los drivers de forma manual de un Arduino UNO, en primer lugar hay que conectar la placa y acto seguido acceder al panel de control y desde ahí se abre el administrador de dispositivos.
2. Seleccionar el puerto, click derecho sobre él y seleccionar la opción Updat drivers.

3. Avanzar en las ventanas de instalación.
4. Y finalmente ya debería aparecer instalados los drivers y reconocer la placa.

![Figuras 39, 40, 41 y 42. Instalación drivers Arduino](image)

Para comenzar a trabajar en la IDE de Arduino hay que configurar las comunicaciones entre la placa Arduino y el ordenador, abriendo en menú “Tools” y seleccionando “Serial Port”, una vez en este punto hay que seleccionar el puerto serie al que está conectada la placa. Si la instalación de los drivers ha sido manual el puerto que aparecerá será el que se le haya asignado previamente.

El entorno desarrollo de Arduino tiene la apariencia de la siguiente figura (Figura 43) y siempre que se abra un nuevo proyecto desde cero la ventana será la misma.

![Figura 43. Entorno Arduino](image)
ACCESOS RÁPIDOS EN EL IDE

<table>
<thead>
<tr>
<th>Verifica que el programa está bien escrito y puede funcionar.</th>
<th>Abre un programa.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carga el programa a la placa de Arduino tras compilarlo.</td>
<td>Guarda el programa en la dirección del disco duro del ordenador.</td>
</tr>
<tr>
<td>Crea un programa nuevo.</td>
<td>Monitor Serial.</td>
</tr>
</tbody>
</table>

Tabla 20. Accesos rápidos en el entorno de Arduino

Cuando se carga un código o programa en la placa a través del puerto USB, durante la duración de la misma se encienden los LED indicadores de que se están enviando y recibiendo información por el puerto serie TX/RX. Si la carga se desarrolla sin inconvenientes al finalizar la interfaz muestra el mensaje de confirmación “Done uploading”.

Programación en Arduino

Los programas escritos con Arduino Software (IDE) se denominan bocetos. Estos bocetos se escriben en el editor de texto y se guardan con la extensión de archivo .ino. El editor tiene funciones para cortar/pegar y para buscar/reemplazar texto. El área de mensajes da retroalimentación al guardar y exportar, además de mostrar errores. La consola muestra el texto generado por el software Arduino (IDE).

Dentro del entorno de Arduino, se emplea un lenguaje propio desarrollado a partir de otros lenguajes de alto nivel, principalmente basado en C y C++.

Permite gestionar bocetos con más de un archivo (cada uno de los cuales aparece en su propia pestaña). Estos pueden ser archivos de código Arduino normales (sin extensión visible), archivos C (extensión .c), archivos C ++ (.cpp) o archivos de encabezado (.h). Es por todo esto que el uso de multitud de librerías está tan extendido y popularizado en Arduino. [9]

Las librerías más habituales que suelen utilizarse en Arduino son aquellas que utilizan el lenguaje de programación C y C++, ya que este lenguaje es la base mayoritaria de el lenguaje de Arduino. Estas bibliotecas proporcionan funcionalidad adicional para su uso en sketches o bocetos, por ejemplo, trabajar con hardware o manipular datos. Para utilizar una biblioteca en un boceto hay que seleccionarla dentro del menú en Importar biblioteca. Esto insertará una o más instrucciones #include en la parte superior del boceto y compilará la biblioteca con su boceto. Debido a que las bibliotecas se suben al tablero con su boceto correspondiente, aumentan la cantidad de espacio que ocupa. Si un boceto ya no necesita una biblioteca, simplemente se elimina la instrucción #include desde la parte superior de su código.

La base de la estructura de la que se compone cualquier programa desarrollado bajo el lenguaje de Arduino posee un nivel de simplicidad que podríamos considerar muy elevado, lo que facilita el desarrollo del programa a diseñar.

La estructura base para el correcto funcionamiento de los códigos Arduino se encuentra dividido en dos partes, a partir de las cuales el resto de la programación se desarrolla dando lugar a una estructura de trabajo solida, cómoda y sobre todo fácil de comprender fragmento por fragmento. Estas dos partes son: Setup y Loop.
• **Función SETUP:** Constituye la preparación del programa. Es la primera función en ejecutarse dentro de un programa en Arduino. Y básicamente es el lugar donde se testean las funciones que llevará a cabo el microcontrolador. Es en este punto donde se establecen algunos criterios que requieren una ejecución única. Dentro de esta función se incluye la declaración de variables y se ejecuta una única vez después de cada encendido o reinicio de la placa Arduino.

• **Función LOOP:** Es la encargada de la ejecución. La función loop en Arduino es la que se ejecuta un número infinito de veces. Al encenderse el Arduino se ejecuta el código del Setup y luego se entra al Loop, el cual se repite de forma indefinida hasta que se apague o se reinicie el microcontrolador. Esta función puede ser considerada la más importante dentro de un código.

También es importante mencionar que en cualquier programa de código abierto como es el caso de Arduino se pueden encontrar funciones diseñadas por usuarios. Esta funciones suelen estar diseñadas para realizar tareas repetitivas e incluso con el objetivo de reducir el tamaño de un programa. Las funciones se declaran asociadas a un tipo de valor que recibe el nombre de “*type*”. Este valor type es el que se devuelve por parte de la función. Si dicha función no devuelve ningún valor, en tal caso se debe colocar delante la palabra “*void*” que viene a identificarse como una función vacía.

**Principales funciones**

Arduino a pesar de ser un lenguaje de código abierto no solo dispone de funciones diseñadas por los usuarios de la plataforma, sino que dispone de una serie de funciones de gran relevancia y que prácticamente su uso es obligatorio en cualquier programa, ya que estas funciones se encargan de la comunicación con el hardware u otras partes del código, asignan tanto salidas como entradas y otras muchas acciones que son básicas para el correcto funcionamiento de un programa.

A continuación se explican el funcionamiento de cada una de estas funciones, debido a que necesitamos entenderlas para poder comprender cada línea del código. Añadir que entre paréntesis se indicara las funciones que se utilizan y las que no en el código del simulador que se ha diseñado y se encuentra en el **ANEXO I** de este documento:

- **PinMode(pin, mode):** [*Utilizada en el código del simulador*]. Utilizada en la función Setup() para la configuración de los pines, asignándolos con entradas (Input) o como salidas (Output).

- **AnalogWrite(pin, value):** [*Utilizada en el código del simulador*]. Escribe un valor pseudo-analógico mediante modulación PWM en un pin de salida de este tipo de modulación.

- **AnalogRead(pin):** [*Utilizada en el código del simulador*]. Lee valores de los pines analógicos asignados, con una resolución de 10bits. Su valor resultante es un número entero que se encuentra entre 0 y 1023.
- **DigitalWrite**(pin, **value**): [Utilizada en el código del simulador]. Introduce niveles HIGH y niveles LOW en el pin digital asignado.

- **DigitalRead**(pin): [Utilizada en el código del simulador]. Lee el valor de un pin digital asignado previamente y devuelve un valor HIGH o LOW.

- **Delay**(ms): [Utilizada en el código del simulador]. Pausa el programa en el tiempo fijado en milisegundos.

- **Min**(x,y): [No utilizada en el código del simulador]. Devuelve el mínimo de sus parámetros.

- **Max**(x,y): [No utilizada en el código del simulador]. Devuelve el máximo de sus parámetros.

- **Millis()**: [Utilizada en el código del simulador]. Devuelve los milisegundos de ejecución que lleva ejecutándose la placa de Arduino, actuando como un valor long unsigned.


- **RandomSeed**(seed): [Utilizada en el código del simulador]. Identifica un valor como punto de inicio para la función random().

- **Serial.println**(data): [Utilizada en el código del simulador]. Imprime información en el Monitor serial del IDE de Arduino.

- **Serial.begin**(rate): [Utilizada en el código del simulador]. Abre el puerto serie y especifica la velocidad de transmisión, que normalmente suele ser 9600.

- **Serial.read()**: [Utilizada en el código del simulador]. Lee y captura un carácter desde el puerto serie.

- **Serial.available()**: [No utilizada en el código del simulador]. Devuelve el número de caracteres disponibles para leer desde el puerto serie.
Software del simulador

La programación de todo el programa de simulación se ha realizado enfocada en todo momento a que el simulador diseñado tenga el comportamiento lo más parecido posible a la ECU de un vehículo real, pero siempre teniendo en cuenta las restricciones que existen con el material disponible para utilizar y la cantidad de componentes que es capaz de soportar una placa de Arduino UNO, porque un exceso de componentes provocaría que el circuito no estuviera suficientemente alimentado para un correcto funcionamiento, ya que la placa de Arduino no sería capaz de responder a tantos dispositivos conectados entre sí. Aunque siempre existe la posibilidad de alimentar de forma externa para no sufrir este problema, pero como el objetivo es comprobar el funcionamiento del sistema CAN-Bus con Arduino mediante un circuito simple, no es necesario introducirle dicha alimentación.

El código realizado que se encuentra en el ANEXO I, sustenta su funcionamiento en las librerías que se le han implementado. Una librería es la mcp_can.h para poder habilitar la comunicación CAN con el Arduino, cuyo fabricante es Seed, y la otra es la librería SPI.h de Arduino para que pueda existir comunicación entre la placa CAN-Bus y la del propio Arduino. Ambas librerías junto a la estructura principal del código se explican en este apartado.

La actividad que realiza el programa de simulación de la ECU es la de establecer comunicación con el conector OBD-II, y acto seguido responder a la peticiones que realiza el dispositivo. Tiene la capacidad de tanto recibir como de responder las peticiones de los modos 01, 02, 03 y 04 del estándar OBD el cual dispone de hasta 9 modos, también almacena DTCs de error al accionar el pulsador del circuito que simula un fallo del sistema el cual provoca a su vez la activación del LED que ejerce de indicador MIL y entonces el simulador comienza a contar la teórica distancia que se recorre con el indicador encendido. Además, también es capaz de enviar datos en tiempo real como la aceleración o el nivel de batería (controlados por los potenciómetros) al ser capaz de soportar las peticiones del modo 01.

Para ello la estructura base del código se basa en la que se basa todo programa de Arduino, la función SETUP y la función LOOP. Y dentro de estas funciones se realizan llamadas a otras funciones que se crean a lo largo del programa con el apoyo de las librerías.

Cada una de las funciones utilizadas habilita nuevas aplicaciones, pero es el funcionamiento en conjunto lo que permite que el sistema actúe como se desea. Y gracias a esto, al simulador se le ha podido habilitar de las funciones deseadas.

Para comprender bien el programa que ha sido desarrollado para el buen funcionamiento del proyecto, en primer lugar se explica brevemente las dos librerías genéricas utilizadas, y justo después se explican las funciones que ha desarrollado a lo largo del código para crear el programa del simulador de la ECU.

Librería genérica SPI.h

Esta librería es perteneciente a Arduino, y fue lanzada por la plataforma para poder comunicar las placas Arduino UNO con otros circuitos integrados, como lo son las placas Shield del modelo que se utiliza para el simulador. SPI realiza la comunicación entre placas a través de los siguientes pines:
• **SCLK**: Señal de reloj del bus. Es el encargado de marcar la velocidad a la que se transmiten los bits.

• **SS**: Habilita el circuito integrado hacia el que se envían los datos. Es un tipo de señal cuyo uso no es estrictamente necesario.

• **MISO**: Señal de entrada hacia el dispositivo, por aquí se reciben los datos desde un circuito integrado a otro.

• **MOSI**: Transmisión de datos entre circuitos.

La siguiente tabla muestra los pines de las líneas SPI en las diferentes tarjetas de Arduino:

<table>
<thead>
<tr>
<th>Tarjeta Arduino / Genuino</th>
<th>MOSI</th>
<th>MISO</th>
<th>SCK</th>
<th>SS esclavo</th>
<th>SS maestro</th>
<th>Nivel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uno o Due</td>
<td>11 o ICSP-4</td>
<td>12 o ICSP-1</td>
<td>13 o ICSP-3</td>
<td>10</td>
<td>-</td>
<td>5V</td>
</tr>
<tr>
<td>Mega1280 o Mega2560</td>
<td>51 o ICSP-4</td>
<td>50 o ICSP-1</td>
<td>52 o ICSP-3</td>
<td>53</td>
<td>-</td>
<td>5V</td>
</tr>
<tr>
<td>Leonardo</td>
<td>ICSP-4</td>
<td>ICSP-1</td>
<td>ICSP-3</td>
<td>-</td>
<td>4, 10, 52</td>
<td>3,3V</td>
</tr>
<tr>
<td>Debedo</td>
<td>ICSP-4</td>
<td>ICSP-1</td>
<td>ICSP-3</td>
<td>-</td>
<td>-</td>
<td>3,3V</td>
</tr>
<tr>
<td>Cero</td>
<td>ICSP-4</td>
<td>ICSP-1</td>
<td>ICSP-3</td>
<td>-</td>
<td>-</td>
<td>3,3V</td>
</tr>
<tr>
<td>101</td>
<td>11 o ICSP-4</td>
<td>12 o ICSP-1</td>
<td>13 o ICSP-3</td>
<td>10</td>
<td>10</td>
<td>3,3V</td>
</tr>
<tr>
<td>MKR1000</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>3,3V</td>
</tr>
</tbody>
</table>

**Tabla 21. Pines de las líneas SPI**

**Librería genérica mcp_can.h**

Esta librería es la que habilita todo el funcionamiento del sistema CAN-Bus y está especialmente adaptada para la placa que se ha utilizado en este proyecto la CAN-BUS Shield V1.2. Ha sido diseñada por Seed, fabricante el cual la ha desarrollado para código abierto de tal manera que los usuarios interesados puedan aportar mejoras a la librería.

La librería mcp_can.h dispone de las siguientes funciones que se describen a continuación brevemente:

• **mcp_can**: Se encarga de establecer el pin CS para la comunicación serie.

• **begin**: Inicia el bus CAN a la velocidad especificada como parámetro de entrada.

• **init_Mask**: Establece una máscara para todos los identificadores CAN (ID).

• **init_Filt**: Establece los filtros para nuestros identificadores CAN (ID).
- **sendMsgBuf**: Se encarga de enviar 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**: Lectura de los mensajes CAN recibidos.

- **readMsgBufID**: Lectura de los mensajes CAN recibidos de una dirección ID específica.

- **checkReceive**: Realiza la comprobación de la recepción de algún mensaje CAN.

- **checkError**: Comprobación de errores.

- **getCanId**: Almacena la dirección ID, cuando se produce la detección de un mensaje recibido.

- **RemoteRequest**: Activación de una bandera si se ha recibido una trama de petición remota.

- **ExtendedFrame**: Activación de una bandera si se ha recibido una trama CAN Extended.

**Funciones importantes diseñadas para el proyecto: F. SETUP**

La función Setup es el lugar donde se han asignado las distintas variables y los pines de entradas y salidas del programa de simulación, y también es desde donde se enviarán las notificaciones de errores. Además esta función es la encargada de establecer la comunicación de las placas Arduino UNO y CAN-Bus Shield con el ordenador a través del dispositivo ELM 327.

![Figura 44. Diagrama de flujo función setup](image)
El siguiente fragmento del código que se encuentra en el ANEXO I, es el relativo a la función que se ha creado para la F. SETUP:

```c
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( digitalPinToInterrupt(pulsador), InterrupDTC1, RISING); // 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;
  }
}
```

Como se puede apreciar en el código cuando la comunicación ha sido iniciada el método que se utiliza para generar un error mediante el pulsador es el de las interrupciones. Gracias al cual mediante el comando `attachInterrupt( digitalPinToInterrupt(pulsador), InterrupDTC1, RISING)` podemos llamar a la función `InterrupDTC1` (la función que ha sido diseñada para encargarse de los DTC, y que se explica más adelante) cuando el pulsador es accionado, e introducimos un flanco de subida para provocar que el indicador luminoso LED MIL se encienda.
Funciones importantes diseñadas para el proyecto: F. LOOP

Es la función principal del programa debido a que es la que contiene el programa en sí, es decir, esta función es donde se ejecuta el programa, y además se repite constantemente.

En esta función se envían a los distintos modos las peticiones que son realizadas por el dispositivo OBD-II (ELM 327), esto provoca la respuesta de las funciones encargadas del funcionamiento de los modos. Además la función se encarga de comprobar si se ha producido la interrupción del pulsado (como si fuese un error). Si la interrupción del pulsador se produce el programa activara el indicador LED y realizara la llamada de la función encargada de realizar el cálculo de los kilómetros recorridos con el LED iluminado.

![Diagrama de flujo función loop](image-url)

Figura 45. Diagrama de flujo función loop
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 que responder
                break;
            case 3: // Si estamos trabajando en el modo 03 -> DTCs
                ReplyMode03(); // Llamamos a la función que responder
                break;
            case 4: // Si estamos trabajando en el modo 04 se 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 con MIL encendido
CalMeters();
}
else{
digitalWrite(LedMIL, LOW);
}

En el código de la función vemos cómo en primer lugar se verifica mediante un IF si llega algún tipo de información o trama desde el CAN.

Si se da el caso de existir mensaje, el mismo es leído y mediante el switch (MODE) se averigua en cuál de los casos nos encontramos, es decir, si es una trama del Modo 01 (case 1), Modo 03 (case 3) o del Modo 04 (case 4).

Cuando se detecta que se está en el case 3 y existe error, el MIL se ilumina (DTCflag == 1) y se produce la llamada de la función CalMeters() para monitorear la distancia recorrida con el LED encendido.
Funciones importantes diseñadas para el proyecto:

F. ReadMessage

Con esta función se consiguen leer y almacenar en una tabla de datos todos los mensajes relevantes que envía el dispositivo OBD-II (ELM 327), es decir, los del modo 01. Esta función es muy dependiente de la librería mcp_can.h ya que para realizar la acción que le corresponde necesita utilizar las funciones “readMsgBuf” y “getCanId” de esa librería.

El siguiente fragmento del código que se encuentra en el ANEXO I, es el relativo a la función que se ha creado para la F. ReadMessage:

```c
// Salidas: Message. Función para leer los datos OBD-II dentro de la trama CAN, devuelve la cadena leída

// Si es 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
```

Figura 46. Diagrama de flujo función ReadMessage
// 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);
    // Guarda el modo y el PID requerido
    if (buf[0] != 0){ // Comprobamos mmsg no vacio
        MODE = buf[1];
        Serial.print("Modo: ");
        Serial.println(MODE);
        // Si trabaja en modo 01 guardamos el PID
        if (MODE == 1){
            PID = buf[2];
            Serial.print("PID: ");
            Serial.println(PID);
        }
    }

    return Message;
Con este código se almacena el mensaje del modo y su PID, a la vez que se imprime el indicador CAN Id.

Mediante un bucle **FOR** se realiza la construcción e impresión del mensaje. Para posteriormente almacenarlo en el caso de que sea procedente del Modo 01, o devolverlo en el caso de no serlo.

**Funciones importantes diseñadas para el proyecto:**

**F.ReplyMode01**

Ésta es la función que se encarga en exclusiva del modo 01. Envía respuestas a las peticiones que realiza del dispositivo OBD-II, las cuales se acompañan con indicadores PID para que la ECU identifique con qué parámetro está relacionado la petición.

En la función el mensaje que se va a utilizar como respuesta, se realiza de una forma u otra dependiendo del indicador PID. Y para enviar el mensaje se recurre a la función “**sendMsgBuf**” de la librería mcp_can.h.

![Diagrama de flujo función ReplyMode01](image-url)
El siguiente fragmento del código que se encuentra en el ANEXO I, es el relativo a la función que se ha creado para la F. ReplyMode01:

```c
// 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};  // Inicializa 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 encendida y DTCs 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 refrigerante
    OBDIImsg[3] = random(65, 70); // valor aleatorio
    // Construimos la respuesta
    CAN.sendMsgBuf(0xE8, 0, 8, OBDIImsg);
    break;

case 12: // PID 0C hex
    // Calculo de las RPM ((A*256)+B)/4
    A = ReadRPM();
    // 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
// Valor entre 0 y 65535s formula: (A*256)+B
A = millis()/1000;
// 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
// 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 útil 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); // 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
}

// Construimos la respuesta
En este fragmento correspondiente a la función del Modo 01 lo principal es la estructuración de los PIDs, ya que dependiendo del caso en el que nos encontremos habrá unos PIDs u otros, y esto afectará a la respuesta que se construye en la parte final del código de la función.

Aquí entran en acción dependiendo del caso en el que nos encontremos, otras funciones como millis() (Para medir el tiempo en ms), ReadBatteryLvl() (Lee nivel de batería), etc.

**Funciones importantes diseñadas para el proyecto:**

**F.ReplyMode03**

Realiza la misma actividad que la función ReplyMode01, con la diferencia de que se encarga del modo 03 y por tanto no envía mensajes de respuesta, sino que envía los códigos de error registrados, y si no se registra ninguno envía un mensaje predeterminado por defecto.
El siguiente fragmento del código que se encuentra en el **ANEXO I**, es el relativo a la función que se ha creado para la F. `ReplyMode03`:

```c
// 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
    }
    // Si no hay errores se envía el mensaje inicial
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImsg);
}
```

En esta función se utiliza un **IF** para identificar cuando se produce un error y qué clase de error. En caso de no haber errores inicializamos a cero.

**Resto de funciones diseñadas para el proyecto**

Además de las funciones que se han explicado anteriormente con sus diagramas de flujo, también hay otras funciones en el programa de simulación que aunque no tienen el mismo nivel de relevancia, también son imprescindibles porque sin ellas no se podría realizar el código de forma correcta.

Las funciones diseñadas de las que se habla y también se encuentran dentro del código del **ANEXO I** son las siguientes:

- **InterruptDTC1**: Detecta los errores simulados con la activación un pulsador, lo cual provoca la activación de la bandera de error y comienza a registrarse el tiempo y kilómetros que se están recorriendo.
// void InterrupDTC1()
// Función que atiende a la interrupción 1
void InterrupDTC1() {
    DTCflag = 1;
    TimeMIL = millis();
}

- **ReadBatteryLvl**: Nos permite la lectura del nivel de batería a partir de la posición del potenciómetro número dos.

    // Salidas: Level. Función para leer el nivel de batería según la posición del potenciómetro 2
    int ReadBatteryLvl(){
        // 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;
    }

- **ReadSpeed**: Calcula la velocidad actual dependiendo de la posición en la que se encuentre nuestro potenciómetro número uno.

    //Salidas: Speed. Función para leer la velocidad según la posición del potenciómetro1.
    int ReadSpeed(){
        // Lee la entrada 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;
    }

- **ReadThrottle**: Calcula la aceleración a partir de la posición del potenciómetro número uno.

    // Salidas: TPosition. Función para leer la posición del acelerador según la posición del potenciómetro 1
    int ReadThrottle(){
        // Lee 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;
    }

- **ReadRPM**: Calcula las revoluciones por minuto (rpm) dependiendo de la posición del potenciómetro número uno.
// Salidas: RPM. Función para leer las rpm según la posición del potenciómetro 1
int ReadRPM(){
    // Lee 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;
}

- **CalMeters**: Calcula la velocidad (en m/s) para conocer la distancia recorrida cuando el indicador MIL está encendido.

// Salidas: ninguna. Calcula los metros recorridos por segundo con MIL encendido los metros se van almacenando en la variable global MetersMIL
void CalMeters(){
    // Si se acaba de encender el indicador MIL comienza la cuenta
    if(Tans == 0){
        Tans = millis();
    }
    else{
        // Sigue contando metros con el indicador encendido y pasamos los km/h a metros/segundo y se suman cada segundo transcurrido.
        int minCicle = (millis() - Tans);
        if(minCicle >= 1000){ // Si ha transcurrido 1 segundo
            // Calculamos los metros recorridos por segundo y sumamos
            MetersMIL = MetersMIL + (ReadSpeed() * 10 / 36);
            Tans = millis();
        }
    }
}

---

**Software de representación y diagnóstico**

Actualmente hay una gran cantidad de programas para la visualización e interpretación de los datos de diagnóstico. Para ordenador el más conocido son el ScanTool y el ScanMaster, mientras que para dispositivos portátiles como los smartphone existen infinidad de apps muy conocidas, tal es el caso de Torque Pro, OBD Car Tracker, OBD Fusion, OBD Car Doctor, OBD Link y OBD eZWay.

En este proyecto el diagnóstico se realiza a través de un ordenador Sony Vaio. A partir de las características de este dispositivo y teniendo en cuenta que su sistema operativo es Windows 7 se decidió utilizar el software de diagnóstico ScanMaster.

ScanMaster es un programa muy idóneo para este proyecto debido a que soporta el protocolo CAN, el cual muchos software no soportan. Además su interfaz gráfica es muy útil porque aun a pesar de estar diseñado para el uso casero, dicha interfaz se asemeja mucho a las que se utilizan en el ámbito profesional.
Los requisitos mínimos que pide el programa ScanMaster son:

- Microsoft Windows 98/ME/2000/XP con Microsoft InternetExplorer 5.01.
- Pentium 200MHz o Superior.
- 64MB de RAM o Superior.
- Unidad de CD-ROM o DVD-ROM.
- Un Puerto COM disponible. [6]

Las características de las que dispone el software ScanMaster son las siguientes:

1. El programa detecta automáticamente el Protocolo correspondiente al Vehículo que está conectado.

2. Soporta identificadores de parámetros (PIDs) genéricos SAE J1979 (Modo $01) de $00 a $4E.

3. Puede representar con gráficos o indicadores toda la información o solamente la información seleccionada, respecto de las mediciones soportadas por el controlador OBD-II del vehículo.

4. Ver la condición del sistema del vehículo cuando una emisión relacionada con un código de error se ha obtenido en el modo FreezeFrame.

5. Leer 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.

6. Soporta más de 4200 descripciones de códigos de error genéricos de SAE J2012 y más de 3600 códigos OBD-II mejorados.

7. Elimina toda la información proveniente de diagnósticos realizados. (Modo $04)

8. Revisar los resultados de los test de sensores de oxígeno realizados por el modulo de control del tren de fuerza del vehículo en modo “Sensor de Oxígeno” (Modo $05).

9. 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”.

10. Presenta diagnóstico de monitoreo continuo realizado mientras el vehículo ha estado en funcionamiento. Esto incluye reporte de fallos que no han sido detectados aun a través del modo “Códigos de Error Pendientes”.


12. Unidades de Medidas Métricas e inglesas. [6]
En la siguiente imagen podemos ver la interfaz de ScanMaster, en la cual se puede apreciar que todas las herramientas se encuentran a la vista.

**Figura 49. Inicio ScanMaster**

Hay varias opciones (Ventanas) que se pueden utilizar para hacer un análisis a un automóvil con este software, y a continuación se procede a su descripción:

**Figura 50. Ventanas ScanMaster**

- **INICIO**: En esta ventana es donde se comprueba si la ECU a través del ELM 327 está conectada. Si la ECU aparece conectada, en esta ventana ya aparecerían los primeros datos de la ECU, como pueden ser el protocolo o el nivel de batería.

**Figura 51. Inicio ScanMaster**
• **INFORMACIÓN DEL VEHÍCULO:** Aquí se muestra información específica del vehículo, si es proporcionada. Los datos que se pueden observar son: VIN, ID de Calibración, Número de Verificación de Calibración, Seguimiento del Desempeño en Funcionamiento.

![Figura 52. Información del vehículo ScanMaster](image)

• **ESTADO DEL SISTEMA:** Se puede ver el estado del indicador luminoso y la cantidad de DTCs.

![Figura 53. Estado del sistema ScanMaster](image)

• **CODIGOS DE ERROR:** En esta venta se registran los DTCs. Y se pueden filtrar por DTCs estándar (Generic) y por DTCs del fabricante (Manufacturer).

![Figura 54. Códigos de error ScanMaster](image)
• **DATOS ESTATICOS:** Para ver los datos congelados.

![Figura 55. Datos estáticos ScanMaster](image)

• **OXYGEN SENSOR:** Es donde se muestran los resultados de los test de oxígeno. En esta venta con los resultados obtenidos de los test, el programa realiza una representación grafica de los niveles.

![Figura 56. Oxygen sensor ScanMaster](image)

• **RESULTADO DE LOS TEST:** La función de esta ventana es el acceso a los resultados de los test de a bordo, es decir, los test que realiza el vehículo por su propia cuenta y los almacena.

![Figura 57. Resultado de los Test ScanMaster](image)
• **DATOS EN TIEMPO REAL CON MEDIDORES**: Se puede seguir la información en tiempo real. Como máximo muestra cuatro valores simultáneos.

![Imagen de medidores ScanMaster](image1)

**Figura 58. Datos en tiempo real con medidores ScanMaster**

• **DATOS EN TIEMPO REAL EN TABLAS**: Se puede seguir la información en tiempo real de todos los parámetros.

![Imagen de tablas ScanMaster](image2)

**Figura 59. Datos en tiempo real en tablas ScanMaster**

• **DATOS EN TIEMPO REAL EN GRAFICOS**: Representa gráficamente los resultados en tiempo real.

![Imagen de gráficos ScanMaster](image3)

**Figura 60. Datos en tiempo real en gráficos ScanMaster**
• CONFIGURACIÓN DEL PID: Desde aquí se pueden modificar los parámetros que se muestran en las ventanas de datos en tiempo real.

Figura 61. Configuración del PID ScanMaster

• POWER: Se utiliza para realizar test gráficos personalizados combinando las representaciones gráficas de la aceleración, las revoluciones por minuto (PRM) y la velocidad.

Figura 62. Power ScanMaster
Pruebas y Mediciones

En este apartado se realizan las pruebas pertinentes conectando el ELM 327 al simulador de la ECU y al ordenador, para realizar el análisis de diagnóstico con el programa ScanMaster.

En las pruebas que se realizan no se utilizan todas las opciones de las que dispone el programa de diagnóstico, ya que al tratarse de un simulador sólo va a realizar las funciones que se le han programado. Por lo tanto las ventanas como la de el test de oxígeno no se van a utilizar. Pero por el contrario todas las relacionadas con el nivel de batería, velocidad, aceleración, kilómetros recorridos con el identificador MIL encendido y los registros en tiempo real sí se utilizan.

Figura 63. Simulador de ECU
Nivel de batería

El nivel de batería se controla mediante el potenciómetro número uno (cable rojo). Según la posición en la que se encuentre en cada momento el nivel de carga de la batería cambiará.

Figura 64. Control de batería

Dicho nivel de batería lo podemos ver en la ventana de inicio al conectar la ECU.

Figura 65. Datos ScanMaster
En la imagen anterior se apreciaba que el nivel de batería era de 0.3 V, lo que se podría considerar prácticamente descargada. Ahora bien si volvemos a variar la posición del potenciómetro en el sentido contrario aumentaremos el nivel de batería.

Como se puede observar ahora el nivel de batería marca 13.3 V, lo que nos muestra la diferencia que hay entre tener muy cerrado el potenciómetro o muy abierto.

Hay que añadir también que el nivel máximo que se hubiera podido alcanzar si se gira por completo el potenciómetro habría sido 15 V que era la alimentación que se le estaba introduciendo al circuito.

**Velocidad, aceleración y RPM**

La velocidad y las RPM son dos factores que en el programa del simulador (código en el ANEXO I) implementado sobre la placa de Arduino UNO han sido “diseñadas” para que sean dependientes de la aceleración.

La aceleración se controla a través del potenciómetro número dos (cable azul), el cual ejerce como una especie de acelerador. Al girarlo en el sentido correcto acelera y si se gira al contrario reduce la velocidad como cuando en un automóvil se levanta el pie del acelerador.
La aceleración en el programa viene expresada en porcentaje bajo el nombre de valor de carga, ya que el programa interpreta la posición del potenciómetro como el porcentaje de la exigencia que se le pide al supuesto motor.

Con el manejo del potenciómetro también se controla tanto la velocidad como las revoluciones por minuto (RPM), ya que como se ha dicho anteriormente ambas son dependientes de la aceleración.
La interfaz de ScanMaster nos permite ver los resultados en tiempo real todos juntos en la misma pantalla lo cual facilita en gran medida a la hora de querer comparar los valores.

Figura 69. Monitoreado ScanMaster

La interfaz también da la posibilidad de seguir los datos en tiempo real de forma gráfica para poder ir controlando su desarrollo a lo largo del tiempo.

Figura 70. Monitoreado gráfico ScanMaster
Indicador luminoso MIL
El simulador está programado de tal manera que cuando se acciona el pulsador, se interpreta como un fallo, o lo que es lo mismo un DTC. Lo que provoca que el diodo LED que ejerce como indicador luminoso MIL se encienda.

Figura 71. Activación indicador MIL

Cuando el indicador MIL se ilumina manda un aviso para que se activen las funciones encargadas de controlar cuanto tiempo está encendido. De esta manera junto a la velocidad el software calcula los kilómetros recorridos con el indicador encendido. Acto seguido el dispositivo OBD-II recibe el mensaje, permitiendo de esta manera que el programa de diagnóstico en uso pueda ir actualizando la información de la distancia recorrida bajo esta circunstancia de error.

Figura 72. Distancia recorrida con el MIL encendido
Como detalle final sobre las pruebas, hay que añadir que todos los parámetros pueden ser seguidos a la vez en tiempo real en una tabla que funciona igual que los marcadores o las gráficas, con la diferencia de que no hay un límite de ver cuatro valores simultáneos sino que pueden verse todos a la vez.

Figura 73. Registro global en tiempo real

Podemos apreciar en la imagen como se realiza un barrido constante de los parámetros que varían en cada momento. También es importante matizar que algunos parámetros como la presión barométrica son aleatorios ya que el simulador no está preparado para generarlos, y aparecen por defecto.
Conclusiones y mejoras

Conclusiones
Tras todo el diseño y montaje del simulador de la ECU, y tras haber desarrollado las funciones necesarias, he podido llegar a la conclusión de que es posible la fabricación de una ECU similar a la de los automóviles para estudiar y comprobar el modo de funcionamiento del CAN BUS a través de Arduino simulando las medidas típicas con un circuito sencillo, lo cual genera un coste muy reducido. Esto es debido a que los componentes necesarios para realizarla de forma simple son de lo más común, ya que estos componentes son resistencias, diodos LED y potenciómetros. Incluso las placas de Arduino UNO y CAN-Bus Shield v1.2 que serían los componentes más complejos y con mayor coste, siguen siendo componentes bastante sencillos de comprender su manejo y con mucho material disponible para su aprendizaje.

Las facilidades que ha aportado Arduino para realizar el programa de simulación, son muchas. En primer lugar, el único coste que genera es la placa ya que todo lo relacionado con Arduino es gratuito. En segundo lugar, Arduino al ser un lenguaje de código abierto dispone de varias librerías útiles para realizar este simulador, y además también hay bastantes funciones listas para implementar, que han sido desarrolladas por usuarios. Y por último, es importante mencionar la enorme compatibilidad de la que dispone Arduino para trabajar con protocolos CAN.

Todo esto abre la puerta a la posibilidad de que cualquier persona un poco interesada en la electrónica pueda fabricarse un sistema de diagnóstico y de órdenes para su vehículo o para cualquier otro dispositivo en el que se pueda implementar.

Finalmente, se ha conseguido que el simulador fuera capaz de soportar los modos más importantes en el estándar OBD-II, el modo 01, 03 y 04. Además se ha conseguido simular la aceleración, los controles de velocidad, parámetros de energía y una pequeña representación de cómo actúan los errores. Por lo cual, los objetivos buscados al comienzo se han conseguido realizar satisfactoriamente.

Mejoras
Este proyecto se puede utilizar como base para futuras mejoras debido a que utiliza el modelo más simple de Arduino UNO. Si se utilizase otra placa con mayor potencia y atribuciones como es el caso de Arduino MEGA, se podría conseguir implementar todos los modos del protocolo OBD-II.

Algunas mejoras que se le pueden incluir es el aumento de sensores para poder simular mayor número de fenómenos, pero hay que prestar atención de la alimentación de la placa porque si hay demasiados sensores no proporcionará suficiente suministro de energía obligando a reducir sensores o a aplicar algún tipo de alimentación externa.

Otra mejora sería buscar la forma de proporcionar conectividad inalámbrica al OBD-II, para poder entablar comunicación sin necesidad de conectar cables, lo que facilitaría la conexión con dispositivos móviles y tablets.
Bibliografía

1. Dogan Ibrahim. ADVANCE PIC MICROCONTROLLER PROJECTS IN C. From USB to ZIGBEE with PIC 18F Series

https://www.mecanicoautomotriz.org/1437-manual-sistema-can-bus-datos-bosch-diseno-funcionamiento


5. Pines OBD-II.  
http://automotrizenvideo.com/configuracion-de-pines-en-el-conector-de-obdii/  
http://obd2-elm327.es/protocolos-conexion-obd2  
https://equipoautomotrizjavaz.com/equipo-de-diagnostico/scanners-multimarcas/alimentacion-de-5-protocolos-de-obd2.pdf


7. ELM327 USB INTERFACE SCANNER OBD CAN-BUS TOOL ELM 327 V 1.5A  

8. Modos y PIDs.  
http://copro.com.ar/PIDs_OBD-II.html  
https://es.wikibooks.org/wiki/El_OBDII_Completo/Protocolos/lista_SAE_ISO  
http://obd2-elm327.es/modos-prueba-obd2-elm327  
https://es.wikibooks.org/wiki/El_OBDII_Completo/Los_PIDs/Modos_PID

http://www.prometec.net/interrupciones/  
https://www.luisllamas.es/que-son-y-como-usar-interrupciones-en-arduino/  
https://learn.sparkfun.com/tutorials/what-is-an-arduino  
http://arduino.cl/que-es-arduino/

10. Códigos de falla o DTC  
http://www.cise.com/portal/notas-tecnicas/item/228-acerca-de-los-c%C3%B3digos-de-falla-o-dtc.html

11. Protocolos.  
https://learn.sparkfun.com/tutorials/getting-started-with-obd-ii  
http://www.oneminuteinfo.com/2014/02/can-bus-obd-ii-explained-for-dummies.html  
http://pinoutguide.com/CarElectronics/obd_can_bus_pinout.shtml  
https://es.slideshare.net/fer technicians/funciones-obd2  
http://www.onboarddiagnostics.com/page03.htm#PWM
12. Modelo OSI y TCP/IP

13. SAE J1850 Description
   http://www.interfacebus.com/Automotive_SAE_J1850_Bus.html

14. J1850 Class B Data Communications Network Interface
   http://standards.sae.org/j1850_201510/

15. CSMA/CR (carrier sense multiple access/collision resolution)

16. Can Bus
   http://wiki.seeed.cc/CAN-BUS_Shield_V1.2/
   http://wiki.seeed.cc/How_to_install_Arduino_Library/
   https://aprendiendoarduino.wordpress.com/2015/03/23/shields-para-arduino/
   http://saber.patagoniatec.com/can-bus-shield/

17. Atmega

18. ELM Electronics “ELM327DSF – OBD to RS232 Interpreter”
   www.elmelectronics.com
PARTE II: Planos. Pliego de condiciones. Presupuesto Anexo I.
Planos

Circuito del simulador

Circuito del simulador con ELM327
Diagramas de flujo

**Función Setup**

1. Se definen los parámetros y se establece comunicación. Y se define Interrup1 DIS1 0
2. Inicio
3. Notificación de detección de error
4. Si: Inicio correcto del sistema CAN-BUS
5. Salir de SETUP

**Función Loop**

1. Si mensaje sistema correcto
2. Llamada de funciones (Modo 1)
3. Llamada función Modo 1
4. NO: MODO 01
5. MODO 03
6. Llamada función Modo 3
7. MODO 04
8. Comienzo el conteo de los kilómetros recorrido con el LCD extendido
9. APAGAR INDICADOR LUMINOSO MIL
10. Si: Detección de error
11. Encender indicador luminoso MIL y llamar función Calibrar
Función ReadMessage

Función ReplyMode01

Función ReplyMode03
Pliego de condiciones

El pliego de condiciones es el lugar donde podremos encontrar las hojas de características y todas las especificaciones técnicas de todos los componentes utilizados a lo largo del proyecto. Gracias a este apartado se puede tener de manera rápida la información técnica deseada sin la necesidad de buscar por todo el proyecto.

**Arduino 1.8.3**

Software de vital importancia en el proyecto y

Es un software de fuente abierta Arduino (IDE) hace que sea fácil escribir código y subirlo a la placa. Se ejecuta en Windows, Mac OS X y Linux. El entorno está escrito en Java y basado en el procesamiento y otros software de código abierto.

Este software se puede utilizar con cualquier tarjeta Arduino.

**ScanMaster**

En el proyecto se utiliza este programa como interfaz para la visualización y el diagnóstico de la información enviada por el OBD-II.

Soporta los siguientes protocolos:

1. SAE J1850 PWM (41.6 Kbaud)
2. SAE J1850 VPW (10.4 Kbaud)
3. ISO 9141-2 (5 baud init, 10.4 Kbaud)
4. ISO 14230-4 KWP (5 baud init, 10.4 Kbaud)
5. ISO 14230-4 KWP (fast init, 10.4 Kbaud)
6. ISO 15765-4 CAN (11 bit ID, 500 Kbaud)
7. ISO 15765-4 CAN (29 bit ID, 500 Kbaud)
8. ISO 15765-4 CAN (11 bit ID, 250 Kbaud)
9. ISO 15765-4 CAN (29 bit ID, 250 Kbaud) [6]

Requisitos mínimos:

- Microsoft Windows 98/ME/2000/XP con Microsoft InternetExplorer 5.01.
- Pentium 200MHz o Superior.
- 64MB de RAM o Superior.
Unidad de CD-ROM o DVD-ROM.

Un Puerto COM disponible. [6]

El programa detecta automáticamente el Protocolo correspondiente al Vehículo que está conectado.

Soporta identificadores de parámetros (PIDs) genéricos SAE J1979 (Modo $01) de $00 a $4E.

Puede representar con gráficos o indicadores toda la información o solamente la información seleccionada, respecto de las mediciones soportadas por el controlador OBD-II del vehículo.

Ver la condición del sistema del vehículo cuando una emisión relacionada con un código de error se ha obtenido en el modo FreezeFrame.

Leer Códigos de Error DTC (y sus descripciones estándar según SAE) que provocan que la luz del “Check Engine” este 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.

Elimina toda la información proveniente de diagnósticos realizados. (Modo $04)

Revisar los resultados de los test de sensores de oxígeno realizados por el modulo 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. [6]

**Arduino UNO**

Es la placa utilizada para implementar el código, y es la que hace que funcione el simulador, porque sobre ella es donde se carga el código.
El Arduino Uno es una placa de microcontroladores basado en el ATmega328. Tiene 14 pines digitales de entrada / salida (de los cuales 6 se pueden utilizar como salidas PWM), 6 entradas analógicas, un resonador cerámico de 16 MHz, una conexión USB, una toma de alimentación, una cabecera ICSP y un botón de reinicio. Contiene todo lo necesario para soportar el microcontrolador. Simplemente conéctelo a un ordenador con un cable USB o conéctelo con un adaptador de CA a CC o batería para empezar.

<table>
<thead>
<tr>
<th>Microcontroller</th>
<th>ATmega328</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Voltage</td>
<td>5V</td>
</tr>
<tr>
<td>Input Voltage (recommended)</td>
<td>7-9V</td>
</tr>
<tr>
<td>Input Voltage (limits)</td>
<td>5-20V</td>
</tr>
<tr>
<td>Digital I/O Pins</td>
<td>14 (of which 6 provide PWM output)</td>
</tr>
<tr>
<td>Analog Input Pins</td>
<td>6</td>
</tr>
<tr>
<td>DC Current per I/O Pin</td>
<td>40 mA</td>
</tr>
<tr>
<td>DC Current for 3.3V Pin</td>
<td>50 mA</td>
</tr>
<tr>
<td>Flash Memory</td>
<td>32 KB (ATmega328) (0.5 KB used by bootloader)</td>
</tr>
<tr>
<td>SRAM</td>
<td>2 KB (ATmega328)</td>
</tr>
<tr>
<td>EEPROM</td>
<td>1 KB (ATmega328)</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>16 MHz</td>
</tr>
</tbody>
</table>

**CAN-BUS Shield V1.2**

Esta shield permite a una placa Arduino comunicarse mediante el bus CAN. Utiliza el controlador CAN MCP2515 de Microchip. La comunicación se realiza mediante un conector sub-D de 9 pines para utilizar con un cable OBDII (no incluido) lo que la hace perfecta para aplicación en automóviles.

Características:

- Bus CAN v2.0B (1 Mb/s)
- Interfaz SPI de 10 MHz
- Standard and extened data and remote frames
- Conexión CAN mediante conector sub-D de 9 pines
- Conexión estándar para cable OBDII
- Protección de polaridad
- Conector para GPS tipo EM406
- Zócalo para tarjetas de memoria Micro SD
- Conector para pantalla LCD serial
- Botón de RESET
- Joystick integrado y dos diodos LED indicadores integrados.

**Pulsador de cuatro pines**

**Potenciómetro**

**PIHER**

**PT-10 10 mm Carbon Potentiometer**

**FEATURES**
- Carbon resistive element
- IPS4 protection according to IEC 60529
- Polyester substrate
- Also upon request:
  - Wiper positioned at 50% or fully clockwise.
  - Supplied in magazine for automatic insertion.
  - Long life model for low cost control potentiometer applications
  - Self-extinguishable plastic UL 94V-0
  - Cut track option
  - Special tapers
  - Mechanical detents
  - Low torque version
  - Special switch option
  - 5% Linearity and 100K cycles mechanical life

**MECHANICAL SPECIFICATIONS**
- Mechanical rotation angle: 350° ± 5°
- Electrical rotation angle: 220° ± 20°
- Torque: 0.4 to 2 Nm (0.6 to 2.7 in-lb)
- Stop torque: > 5 Nm (>7 in-lb)
- Life*: Up to 10K cycles

* Others check availability

**ELECTRICAL SPECIFICATIONS**
- Range of values*: 1000 ≤ Rn ≤ 5 M (Decad: 1.0 – 2.0 – 2.2 – 2.5 – 4.7 – 5.0)
- Tolerance*: 1000 ≤ Rn ≤ 1MΩ … ±20%  
  1MΩ < Rn ≤ 5MΩ … ±30%  
  5MΩ < Rn ≤ 50MΩ … ±50%  
  100MΩ ≤ Rn ≤ 1GΩ … ±75%  
  1GΩ ≤ Rn ≤ 10GΩ … ±100%
- Max. Voltage: 200 VDC (lin) 100 VDC (no lin)
- Nominal Power: 50°C (122°F) (see power rating curve)
  0.15 W (lin) 0.07 W (no lin)
- Taper (Log & Alog. only Rn > 1K): Lin / Log / Alog.
- Residual resistance: < 0.5 % (1K min.)
- Equivalent Noise Resistance: ≤ 3% (0.0 min.)
- Operating temperature*: -25°C to 70°C (-13°F to 158°F)

* Others check availability

**Up to 80°C depending on application.**
LED

**ELM 327**

Empleado para recoger la información procedente del simulador de la ECU y llevarla hasta el ordenador para su visualización.
Presupuesto

En este aparato se hará un desglose de los costes que han generado los componentes de proyecto, para finalmente conocer coste total amortizado, incluyendo lo que sería la mano de obra.

En primer lugar vemos el inventario del coste que tiene la compra de cada uno de los componentes (Se utilizarán los precios actuales):

<table>
<thead>
<tr>
<th>MATERIAL</th>
<th>Precio (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software ScanMaste</td>
<td>185</td>
</tr>
<tr>
<td>Software Arduino</td>
<td>0</td>
</tr>
<tr>
<td>Elm 327 con puerto USB</td>
<td>10,99</td>
</tr>
<tr>
<td>Arduino Uno atmega 328</td>
<td>21,72</td>
</tr>
<tr>
<td>Shield Can-Bus para Arduino</td>
<td>43,67</td>
</tr>
<tr>
<td>Placa protoboard</td>
<td>4,95</td>
</tr>
<tr>
<td>Ordenador Sony Vaio SVE1512R1EW</td>
<td>500</td>
</tr>
<tr>
<td>Windows 7 Home Premium</td>
<td>35,3</td>
</tr>
<tr>
<td>LED</td>
<td>0,21</td>
</tr>
<tr>
<td>Resistencia 10kohm (2Unidades)</td>
<td>0,5</td>
</tr>
<tr>
<td>Pulsador</td>
<td>0,34</td>
</tr>
<tr>
<td>Potenciómetros (2 Unidades)</td>
<td>0,6</td>
</tr>
</tbody>
</table>

**PRECIOS TOTAL: 803,28 €**

Una vez que tenemos los costes del precio de cada producto, pasamos a calcular las amortizaciones de cada uno de ellos con la siguiente fórmula:

$$Amortización = \frac{Valor \ de \ producto}{Tiempo \ de \ vida}$$

La amortización en los software y en el sistema operativo no se aplicara ya que no tienen un periodo de vida útil estipulado. Tampoco se aplicara a los componentes del circuito ya que su precio es casi nulo.

<table>
<thead>
<tr>
<th>MATERIAL</th>
<th>Precio (€)</th>
<th>Tiempo de vida (Años)</th>
<th>Amortización</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software ScanMaste</td>
<td>185</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Arduino</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elm 327 con puerto USB</td>
<td>10,99</td>
<td>10</td>
<td>1,099</td>
</tr>
<tr>
<td>Arduino Uno atmega 328</td>
<td>21,72</td>
<td>10</td>
<td>2,172</td>
</tr>
<tr>
<td>Shield Can-Bus para Arduino</td>
<td>43,67</td>
<td>10</td>
<td>4,367</td>
</tr>
<tr>
<td>Placa protoboard</td>
<td>4,95</td>
<td>8</td>
<td>0,61875</td>
</tr>
<tr>
<td>Ordenador Sony Vaio SVE1512R1EW</td>
<td>500</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Windows 7 Home Premium</td>
<td>35,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LED</td>
<td>0,21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resistencia 10kohm (2Unidades)</td>
<td>0,5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulsador</td>
<td>0,34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Potenciómetros (2 Unidades)</td>
<td>0,6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finalmente se obtendrá el coste total del proyecto sumando todos los costes y la mano de obra, la cual costará 28800 € ya que es un proyecto de 8 meses de duración y la mano de obra la tomaremos como 15 € a la hora, trabajando un periodo diario de 8 horas (como una jornada laboral).

Los dispositivos a los cuales se les han aplicado amortización, su coste no será el original de la compra, sino el coste amortizado.

\[
\text{Coste amortizan} = \frac{\text{Número de meses de uso}}{\text{Amortización}} \cdot \text{Coste original del equipo} \cdot \% \text{ de Uso}
\]

<table>
<thead>
<tr>
<th>MATERIAL</th>
<th>Precio (€)</th>
<th>Tiempo de vida (Años)</th>
<th>Amortización</th>
<th>Coste amortizado (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software ScanMaste</td>
<td>185</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Arduino</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elm 327 con puerto USB</td>
<td>10,99</td>
<td>10</td>
<td>1,099</td>
<td>20,8</td>
</tr>
<tr>
<td>Arduino Uno atmega 328</td>
<td>21,72</td>
<td>10</td>
<td>2,172</td>
<td>20,8</td>
</tr>
<tr>
<td>Shield Can-Bus para Arduino</td>
<td>43,67</td>
<td>10</td>
<td>4,367</td>
<td>20,8</td>
</tr>
<tr>
<td>Placa protoboard</td>
<td>4,95</td>
<td>8</td>
<td>0,61875</td>
<td>16,64</td>
</tr>
<tr>
<td>Ordenador Sony Vaio</td>
<td>500</td>
<td>10</td>
<td>50</td>
<td>20,8</td>
</tr>
<tr>
<td>Windows 7 Home Premium</td>
<td>35,3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LED</td>
<td>0,21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resistencia 10kohm (2Und)</td>
<td>0,5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulsador</td>
<td>0,34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Potenciómetros (2 Und)</td>
<td>0,6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Mano de obra (€)               | 28800      |
| Coste total Proyecto (€)       | 29121,79   |
ANEXO I

Código del proyecto

```c
#include <mcp_can.h>
#include <mcp_can_dfs.h>
#include <SPI.h>

// Constantes
const int SPl_CS_PIN = 9; // Pin CS
const int LedMIL = 13; // Led indicador MIL pin 13
const char SpeedPin = A0; // Entrada de velocidad analóg (potenciómetro 1)
const char BatteryPin = A1; // Entrada de batería analóg (potenciómetro 2)
const int pulsador = 2; // Pulsador pin 2

// Variables
unsigned long canId = 0x000; // Identificador CAN ID
unsigned char len = 0; // Longitud de datos recibidos
unsigned char buf[8]; // Almacenamiento del mensaje OBD-II
String BuildMessage=""; // Cadena para imprimir el mensaje recibido
int MODE = 0; // Modo del OBD-II
int PID = 0; // IDS del Modo 01
int MILflag = 0; // Bandera (Apagada hasta que sea 1)
volatile int DTCflag = 0; // DTC en memoria
unsigned long TimeMIL = 0; // Crono de tiempo MIL encendido
float MetersMIL = 0; // Distancia con MIL encendido
unsigned long Tans = 0; // Contar un segundo

MCP_CAN CAN(SPl_CS_PIN); // Definir el pin CS

// Salidas: Speed. Función para leer la velocidad según la posición del potenciómetro 1.
int ReadSpeed() {
    // Lee la entrada 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;
}

// Salidas: RPM. Función para leer las rpm según la posición del potenciómetro 1
int ReadRPM() {
    // Lee 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;
}
```
// Salidas: TPosition. Función para leer la posición del acelerador según la posición del potenciómetro 1
int ReadThrottle(){
  // Lee 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;
}

// Salidas: Level. Función para leer el nivel de batería según la posición del potenciómetro 2
int ReadBatteryPin(){
  // Lee la entrada analógica del PIN A1
  int sensorValue = analogRead(BatteryPin);
  // Convierte el valor leído entre 0-1023 en un valor entre 0-255
  int Level = sensorValue * (255.0 / 1023.0);
  return Level;
}

// Salidas: ninguna. Calcula los metros recorridos por segundo con MIL encendido
// los metros se van almacenando en la variable global MetersMIL
void CalcMeters(){
  // Si se acaba de encender el indicador MIL comienza la cuenta
  if(Tans == 0){
    Tans = millis();
  }
  else{
    // sigue contando metros con el indicador encendido
    // Transformamos los km/h a metros/segundo y se suman cada segundo transcurrido
    int minCicle = (millis() - Tans);  // Transcurridos 1 segundo
   if (minCicle >= 1000){ // Si ha transcurrido 1 segundo
      // Calculamos los metros recorridos por segundo y sumamos
      MetersMIL += (ReadSpeed() * 10 / 36)
      // Construye el mensaje y lo imprime
      Tans = millis();
    }
  }
}

// Salidas: Message. Función para leer los datos GBD-II dentro de la tram CAN, devuelve la cadena leída
// Si es solicitud válida guarda el Modo y el PID requerido
String ReadMessage(){
  String Message="";
  CAN.readMsgBuf(len, buf); // Almacenamos el mensaje GBD-II en buf y su longitud en len
  // Imprime la dirección Id
canId = CAN.getCanId(); // Identificador del mensaje
  Serial.print("\n"); Serial.print(canId); Serial.print("\n");
  // Construye el mensaje y lo imprime
  for(int i = 0; i<len; i++)
  {
    if(i == len-1){
      Message = Message + buf[i];
      Message = Message + ";\n";
    }
  }
  return Message;
}
Serial.println(Message);
// Guarda el modo y el PID requerido
if (buf[0] != 0) { // Comprobamos msg no vacío
  MODE = buf[1];
  Serial.print("(Modo: ");Serial.println(MODE);
// Si trabaja en modo 01 guardamos el PID
if (MODE == 1) {
  PID = buf[2];
  Serial.print("(PID: ");Serial.println(PID);
}
return Message;
}

// Función para responder a una petición OBD-II en Modo 1
void ReplyModo1() {
  // Mensaje OBD-II predefinido para responder en el modo 1
  // [len datos, 01+40hex, PID, DatoA, DatoB, DatoC, DatoD]
  unsigned char OBDIImg[8] = {4, 65, PID, 0, 0, 0, 0}; // Inicializa len a 4
  // Datos = 55 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 útil en este caso es 6
    OBDIImg[0] = 6;
    // PIDs sugeridos del 01-20 hex
    OBDIImg[3] = 0x88;
    OBDIImg[4] = 0x18;
    OBDIImg[5] = 0x80;
    OBDIImg[6] = 0x13;
    CAN.sendMsg Buf(0x7E8, 0, 0, OBDIImg); // Respuesta
    break;
  case 1: // PID 01 hex
    // MIL encendido el DTCFlag = 1
    if (DTCFlag == 1) {
      OBDIImg[3] = 128; // Luz MIL encendida y DTCs detectados
    } // MIL apagado, 0 DTCs almacenados, ningún test posible
    // DatoA, DatoB, DatoC, DatoD = 0;
    CAN.sendMsg Buf(0x7E8, 0, 0, OBDIImg); // Respuesta
    break;
  case 5: // PID 05 hex
    // Temperatura del refrigerante
    OBDIImg[3] = random(55,70); // Valor aleatorio
    // Construimos la respuesta
    CAN.sendMsg Buf(0x7E8, 0, 0, OBDIImg);
    break;
  case 12: // PID 0C hex
    // Cálculo de las RPM ((A*256)+B)/4
    A = ReadRFM();
    OBDIImg[3] = (A*4)/256;
    // Construimos la respuesta
    CAN.sendMsg Buf(0x7E8, 0, 0, OBDIImg);
    break;
  }
}
case 13: // PID 0B hex
// Calculo de la velocidad
OBDIImag[3] = ReadSpeed();
// Construimos la respuesta
CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
b Wan:break;

// PID 11 hex
// Calculo de la posicion del pedal del acelerador
OBDIImag[3] = ReadThrottle();
// Construimos la respuesta
CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
b Wan:break;

// PID 1C hex
// Estender OBD-II utilizado
OBDIImag[3] = 3; // El valor 3 indica compatible con OBD y OBD-II
// Construimos la respuesta
CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
b Wan:break;

// PID 19 hex
// Tiempo en segundos desde el arranque
// La funcion millis() mide el tiempo en ms
// Valor entre 0 y 65535; formula: (A*256)+B
A = millis()/1000;
OBDIImag[4] = A - (A/256);
// Construimos la respuesta
CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
b Wan:break;

// PID 20 hex
// PIDo soportados del 21-40 hcx
OBDIImag[0] = 6;
// PIDo soportados del 21-40 hcx
OBDIImag[3] = Ox80;
OBDIImag[4] = OxC2;
OBDIImag[5] = Ox00;
OBDIImag[6] = Ox01;
CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
b Wan:break;

// PID 21 hex
// km recorridos desde que el indicador MIL esta encendida
A = MetersMIL/1000;
// Entre 0 y 65535; formula: (A*256)+B
OBDIImag[4] = A - (A/256);
// Construimos la respuesta
CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
b Wan:break;

// PID 2F hex
// Nivel de la bateria
OBDIImag[3] = ReadBatteryLvl();
// Construimos la respuesta
CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
b Wan:break;
case 64: // PID 40 hex
    // La longitud util en este caso es 6
    OBDIImag[0] = 6;
    // PIDs soportados del 41-60 hex
    OBDIImag[3] = 0x04;
    OBDIImag[4] = 0x08;
    OBDIImag[5] = 0x80;
    OBDIImag[6] = 0x00;
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
    break;

    case 70: // PID 46 hex
    // Temperatura ambiente Λ-40
    OBDIImag[3] = random(70, 75); // valor aleatorio
    // Construimos la respuesta
    CAN.sendMsgBuf(0x7E8, 0, 5, OBDIImag);
    break;

    case 77: // PID 4D hex
    // Tiempo desde que el indicador MIL esta encendido
    // valor entre 0 y 65535 de formula: (A*256)+B
    if (TimeMIL != 0){
        A = (millis() - TimeMIL)/60000; // Tiempo en minutos
    }
    // Construimos la respuesta
    CAN.sendMsgBuf(0x7E8, 0, 8, OBDIImag);
    break;

    case 81: // PID 51 hex
    // Indica el tipo de fuel: electrico 8
    OBDIImag[3] = 8;
    // Construimos la respuesta
    CAN.sendMsgBuf(0x7E8, 0, 6, OBDIImag);
    break;
}

// 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, 0x00, 0x16, 0x16, 0x16, 0x16, 0x16, 0x16, 0x16, 0x16]
    unsigned char OBDIImag[8] = {6, 67, 0, 0, 0, 0, 0, 0}; // inicializamos a 0 no hay DTCs
    // Si se ha detectado un sobrecalentamiento del motor
    if (DICflag == 1){
        OBDIImag[2] = 1; // Número de DTCs = 1
        OBDIImag[3] = 2; // P0217 Sobrecalentamiento del motor HIGH
    }
    // Si no hay errores se envía el mensaje inicial
    CAN.sendMsgBuf(0x7E8, 0, 0, OBDIImag);
}

// void InterruptDTC1()
// Función que atiende a la interrupción 1
void InterruptDTC1()
{
    DICflag = 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(digitalPinToInterrupt(pulsador), InterruptPC1, RISING); // Clancos de subida
  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");
  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 qué modo estamos trabajando
    switch (MODE) {
      case 1: // Si estamos trabajando en el modo 01 -> datos actuales
        ReplyMode01(); // Llamamos a la función que responder
        break;
      case 3: // Si estamos trabajando en el modo 03 -> DTCs
        ReplyMode03(); // Llamamos a la función que responder
        break;
      case 4: // Si estamos trabajando en el modo 04 se 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 con MIL encendido
    CalMeters();
  }
  else{
    digitalWrite(LedMIL, LOW);
  }
}
### PID de los modos OBD

<table>
<thead>
<tr>
<th>PID</th>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>List of PIDs supported (range 01 to 32)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>Status since the last clearing of fault codes</td>
</tr>
<tr>
<td>02</td>
<td>02</td>
<td>Fault code that caused the recording of &quot;freeze frame&quot; data</td>
</tr>
<tr>
<td>03</td>
<td>03</td>
<td>Fuel system status</td>
</tr>
<tr>
<td>04</td>
<td>04</td>
<td>Engine load calculated in %</td>
</tr>
<tr>
<td>05</td>
<td>05</td>
<td>Temperature of the engine coolant in °C</td>
</tr>
<tr>
<td>06</td>
<td>06</td>
<td>Short-term fuel % trim bank 1</td>
</tr>
<tr>
<td>07</td>
<td>07</td>
<td>Long-term fuel % trim bank 1</td>
</tr>
<tr>
<td>08</td>
<td>08</td>
<td>Short-term fuel % trim bank 2</td>
</tr>
<tr>
<td>09</td>
<td>09</td>
<td>Long-term fuel % trim bank 2</td>
</tr>
<tr>
<td>0A</td>
<td>10</td>
<td>Fuel pressure in kPa</td>
</tr>
<tr>
<td>0B</td>
<td>11</td>
<td>Intake manifold absolute pressure in kPa</td>
</tr>
<tr>
<td>0C</td>
<td>12</td>
<td>Engine speed in rpm</td>
</tr>
<tr>
<td>0D</td>
<td>13</td>
<td>Vehicle speed in kph</td>
</tr>
<tr>
<td>0E</td>
<td>14</td>
<td>Timing advance on cylinder 1 in degrees</td>
</tr>
<tr>
<td>0F</td>
<td>15</td>
<td>Intake air temperature in °C</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>Air flow measured by the flowmeter in g/s</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td>Throttle position in %</td>
</tr>
<tr>
<td>12</td>
<td>18</td>
<td>Status of the secondary intake circuit</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
<td>O2 sensor positions bank/sensor</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>Oxygen sensor volts bank 1 sensor 1/td&gt;</td>
</tr>
<tr>
<td>15</td>
<td>21</td>
<td>Oxygen sensor volts bank 1 sensor 2</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
<td>Oxygen sensor volts bank 1 sensor 3</td>
</tr>
<tr>
<td>17</td>
<td>23</td>
<td>Oxygen sensor volts bank 1 sensor 4</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>Oxygen sensor volts bank 2 sensor 1</td>
</tr>
<tr>
<td>19</td>
<td>25</td>
<td>Oxygen sensor volts bank 2 sensor 2</td>
</tr>
<tr>
<td>1A</td>
<td>26</td>
<td>Oxygen sensor volts bank 2 sensor 3</td>
</tr>
<tr>
<td>1B</td>
<td>27</td>
<td>Oxygen sensor volts bank 2 sensor 4</td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
<td>OBD computer specification</td>
</tr>
<tr>
<td>1D</td>
<td>29</td>
<td>O2 sensor positions bank/sensor</td>
</tr>
<tr>
<td>1E</td>
<td>30</td>
<td>Auxiliary input status</td>
</tr>
<tr>
<td>1F</td>
<td>31</td>
<td>Run time since engine start</td>
</tr>
<tr>
<td>20</td>
<td>32</td>
<td>List of PIDs supported (range 33 to 64)</td>
</tr>
<tr>
<td>21</td>
<td>33</td>
<td>Distance travelled with MIL on in kms</td>
</tr>
<tr>
<td>22</td>
<td>34</td>
<td>Relative fuel rail pressure in kPa</td>
</tr>
<tr>
<td>23</td>
<td>35</td>
<td>Fuel rail pressure in kPa</td>
</tr>
<tr>
<td>24</td>
<td>36</td>
<td>O2 sensor (extended range) bank 1, sensor 1 (lambda and volts)</td>
</tr>
<tr>
<td>25</td>
<td>37</td>
<td>O2 sensor (extended range) bank 1, sensor 2 (lambda and volts)</td>
</tr>
<tr>
<td>26</td>
<td>38</td>
<td>O2 sensor (extended range) bank 1, sensor 3 (lambda and volts)</td>
</tr>
<tr>
<td>27</td>
<td>39</td>
<td>O2 sensor (extended range) bank 1, sensor 4 (lambda and volts)</td>
</tr>
<tr>
<td>28</td>
<td>40</td>
<td>O2 sensor (extended range) bank 2, sensor 1 (lambda and volts)</td>
</tr>
<tr>
<td>29</td>
<td>41</td>
<td>O2 sensor (extended range) bank 2, sensor 2 (lambda and volts)</td>
</tr>
<tr>
<td>PID</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>2A</td>
<td>O2 sensor (extended range) bank 2, sensor 3 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>2B</td>
<td>O2 sensor (extended range) bank 2, sensor 4 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>EGR in %</td>
<td></td>
</tr>
<tr>
<td>2D</td>
<td>EGR error in %</td>
<td></td>
</tr>
<tr>
<td>2E</td>
<td>Evaporation purge in %</td>
<td></td>
</tr>
<tr>
<td>2F</td>
<td>Fuel level in %</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Number of warning(s) since faults (DTC) were cleared</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Distance since faults (DTC) were cleared.</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Evaporation system vapour pressure in Pa</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Barometric pressure in kPa</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>O2 sensor (extended range) bank 1, sensor 1 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>O2 sensor (extended range) bank 1, sensor 2 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>O2 sensor (extended range) bank 1, sensor 3 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>O2 sensor (extended range) bank 1, sensor 4 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>O2 sensor (extended range) bank 2, sensor 1 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>O2 sensor (extended range) bank 2, sensor 2 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>O2 sensor (extended range) bank 2, sensor 3 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>3B</td>
<td>O2 sensor (extended range) bank 2, sensor 4 (lambda and volts)</td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>Catalyst temperature in °C bank 1, sensor 1</td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>Catalyst temperature in °C bank 2, sensor 1</td>
<td></td>
</tr>
<tr>
<td>3E</td>
<td>Catalyst temperature in °C bank 1, sensor 2</td>
<td></td>
</tr>
<tr>
<td>3F</td>
<td>Catalyst temperature in °C bank 2, sensor 1</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>List of PIDs supported (range 65 to 96)</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>Monitor status this drive cycle</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Control module voltage in V</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Absolute engine load</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Equivalent fuel/air mixture request</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>Relative throttle position in %</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>Ambient air temperature in °C</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Absolute throttle position B in %</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Absolute throttle position C in %</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>Accelerator pedal position D in %</td>
<td></td>
</tr>
<tr>
<td>4A</td>
<td>Accelerator pedal position E in %</td>
<td></td>
</tr>
<tr>
<td>4B</td>
<td>Accelerator pedal position F in %</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>Commanded throttle actuator in %</td>
<td></td>
</tr>
<tr>
<td>4D</td>
<td>Engine run time since MIL on in min</td>
<td></td>
</tr>
<tr>
<td>4E</td>
<td>Engine run time since faults cleared in min</td>
<td></td>
</tr>
<tr>
<td>4F</td>
<td>External test equipment no. 1 configuration information</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>External test equipment no. 2 configuration information</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>Fuel type used by the vehicle</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>Ethanol fuel %</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>Absolute evaporation system vapour pressure in kPa</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Evaporation system vapour pressure in Pa</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Short-term O2 sensor trim bank 1 and 3</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Long-term O2 sensor trim bank 1 and 3</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>Short-term O2 sensor trim bank 2 and 4</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Long-term O2 sensor trim bank 2 and 4</td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>Absolute fuel rail pressure in kPa</td>
<td></td>
</tr>
<tr>
<td>5A</td>
<td>Relative accelerator pedal position in %</td>
<td></td>
</tr>
<tr>
<td>5B</td>
<td>Battery unit remaining life (hybrid) in %</td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>Engine oil temperature in °C</td>
<td></td>
</tr>
<tr>
<td>5D</td>
<td>Fuel injection timing in °C</td>
<td></td>
</tr>
<tr>
<td>5E</td>
<td>Fuel consumption in litre/hr</td>
<td></td>
</tr>
<tr>
<td>5F</td>
<td>Fuel consumption in litre/hr</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>List of PIDs supported (range 97 to 128)</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Driver demand: torque percentage (%)</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Final engine torque percentage (%)</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Engine torque reference in Nm</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Engine torque data in %</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>Auxiliary inputs / outputs</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>Flowmeter sensor</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>Engine water temperature in °C</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>Air temperature sensor in °C</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>Commanded EGR and EGR error</td>
<td></td>
</tr>
<tr>
<td>6A</td>
<td>Commanded Diesel intake air flow control and relative intake air flow position</td>
<td></td>
</tr>
<tr>
<td>6B</td>
<td>Recirculation gas temperature in °C</td>
<td></td>
</tr>
<tr>
<td>6C</td>
<td>Commanded throttle actuator control and relative throttle position</td>
<td></td>
</tr>
<tr>
<td>6D</td>
<td>Fuel pressure control system</td>
<td></td>
</tr>
<tr>
<td>6E</td>
<td>Injection pressure control system</td>
<td></td>
</tr>
<tr>
<td>6F</td>
<td>Turbocharger compressor inlet pressure in kPa</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>Boost pressure control in kPa</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>Variable Geometry turbo (VGT) control</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>Wastegate control</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>Exhaust pressure in kPa</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>Turbocharger RPM</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>Turbocharger A temperature in °C</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>Turbocharger B temperature in °C</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>Charge air cooler temperature in °C</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>Exhaust Gas temperature (EGT) Bank 1</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>Exhaust Gas temperature (EGT) Bank 2</td>
<td></td>
</tr>
<tr>
<td>7A</td>
<td>Diesel particulate filter (DPF) bank 1</td>
<td></td>
</tr>
<tr>
<td>7B</td>
<td>Diesel particulate filter (DPF) bank 2</td>
<td></td>
</tr>
<tr>
<td>7C</td>
<td>Diesel Particulate filter (DPF) temperature</td>
<td></td>
</tr>
<tr>
<td>7D</td>
<td>NOx NTE control area status</td>
<td></td>
</tr>
<tr>
<td>7E</td>
<td>PM NTE control area status</td>
<td></td>
</tr>
<tr>
<td>7F</td>
<td>Engine run time</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>List of PIDs supported (range 129 to 160)</td>
<td></td>
</tr>
</tbody>
</table>