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

Simulador de la ECU de un vehículo con protocolo ISO 9141-2

Autor: Jose Pablo Villén Macías
Tutor: Antonio Luque Estepa

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

Sevilla, 2016

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

Simulador de la ECU de un vehículo con protocolo ISO 9141-2

Autor:
Jose Pablo Villén Macías

Tutor:
Antonio Luque Estepa
Profesor titular

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Simulador de la ECU de un vehículo con protocolo ISO 9141-2

Autor: Jose Pablo Villén Macías

Tutor: Antonio Luque Estepa

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

Resumen

La mayoría de los vehículos fabricados en la última década integran una interfaz de diagnóstico llamada OBDII, mediante la cual es posible monitorizar y administrar múltiples parámetros sensoriales del vehículo, así como analizar las distintas averías del mismo.

Con el avance de las tecnologías, se están desarrollando muchas aplicaciones para control y monitorización en tiempo real de diferentes parámetros de los vehículos. Esta infraestructura ha abierto una amplia área de investigación relacionada con la conducción inteligente.

El mayor problema para desarrollar este tipo de aplicaciones es la necesidad de tener un vehículo, para depurar y validar las diferentes aplicaciones. Además, incluso teniendo un vehículo, el problema de probar diferentes configuraciones en diferentes escenario prevalece.

Para solucionar este problema se propone el diseño y fabricación de un simulador de la unidad de control de un vehículo. Se trata de una tarjeta diseñada para Arduino Mega que implementa el estándar OBDII siguiendo el protocolo ISO9141-2.

A lo largo del proyecto se estudiarán los aspectos teóricos necesarios, se diseñará y fabricará la tarjeta que simulará la electrónica del vehículo y se programará el código que se cargará en el Arduino.

Abstract

The majority of vehicles built during the last decade integrate an On Board Diagnostic interface called OBDII, through which it is possible to monitor and manage multiple sensory parameters of the vehicle and analyze its damage.

With technology advancing, a lot of applications for control and monitoring of different parameters in real-time are being developed. This infrastructure has opened a broad research area related to smart driving.

The main problem of the development of these applications is the need to have a vehicle, for debugging and validation different applications. Furthermore, even when we have a vehicle, the challenge of testing different configurations in different scenarios remains.

To fix this problem, it is proposed the design and fabrication of a vehicle's control unit simulator. It is a board designed for Arduino MEGA that implements OBDII standard following ISO 9141-2 protocol.

During the proyect, the required theoretical aspects will be considered, the board that will simulate vehicle electronics will be designed and manufactured and the code to be load in Arduino will be written.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

%	Porcentaje
ºC	Grados centígrados
kPa	KiloPascales
rpm	Revoluciones por minuto
Km/h	Kilómetros por hora
seg.	Segundos
mA	miliAmperios
$\bar{x}$	x invertida
-	No definido
ms	milisegundos
kΩ	KiloOhmios
nº	Número
W	Vatios
etc.	Etcétera
MHz	MegaHercios
V	Voltios
kbps	Kilobit por segundo
0x/$	Hexadecimal
~	aproximadamente
min.	Mínimo
max.	Máximo
in.	inches (pulgadas)
app	Aplicación
V_B/Vbat	Tensión de batería
Vih/voh	Tensión alta de entrada/salida
Vil/Vol	Tensión baja de entrada/salida
Tx	Transmisor
Rx	Receptor

1 INTRODUCCIÓN

En este primer capítulo entenderemos el motivo por el cual se realiza el proyecto, así como el objetivo a llegar en el mismo. Esto conlleva a dividir el trabajo en diferentes tareas específicas que en su conjunto conformarán un simulador de la ECU de cualquier vehículo con protocolo ISO9141-2.

1.1. Motivación

El mundo automovilístico va evolucionando diariamente con nuevos componentes, sensores electrónicos, avances en seguridad o interfaces de comunicación que cada vez hacen a los vehículos más óptimos, seguros e inteligentes. Se está investigando y avanzando cada vez más en las redes vehiculares para las cuales se utilizan dispositivos móviles que analizan y procesan la información obtenida de los vehículos a través de una interfaz OBDII.

Sin embargo, las posibilidades y tiempo de investigación necesario para el desarrollo de nuevas propuestas y soluciones para la integración de dispositivos móviles y vehículos se ven limitados por la necesidad de tener a mano un vehículo con un intérprete OBDII conectado para realizar las pruebas de funcionamiento. Además, las soluciones disponibles en la actualidad no cumplen las expectativas requeridas, ya sean porque son muy básicas para probar diferentes parámetros y escenarios, o porque son costosas y poco flexibles.

Debido a estos inconvenientes, es necesario desarrollar un simulador flexible, configurable y fácil de utilizar, en el que se pueda realizar pruebas de manera fácil y rápida y sea compatible con dispositivos móviles.

1.2. Objetivos

Se establece como objetivo general el diseño y fabricación de un simulador de una ECU que sea capaz de comunicarse con un equipo de diagnóstico OBDII (para su posterior visualización en un software de Android), realizar correctamente la inicialización, peticiones y respuestas establecidas por el protocolo a implementar, simular distintos parámetros del vehículo en tiempo real, la activación de una serie de códigos de falla que puedan aparecer mediante la conducción así como el testigo “Check Engine”.

Con el fin de cumplir el objetivo general se ha dividido el trabajo en los siguientes objetivos específicos

- Estudio detallado de los aspectos teóricos que se tendrán presentes en la implementación del simulador.

    - Estudio del estándar OBD y posteriores versiones, así como su conector implicado e indicador MIL.
    - Estudio del intérprete ELM327, puente entre lenguaje OBD y RS232.
    - Estudio de los protocolos OBD vigentes hoy día haciendo hincapié en el ISO9141-2 .
    - Estudio de los diferentes modos de medición de los que dispone el sistema OBD.
    - Estudio de los simuladores existentes.

- Diseño y fabricación de un simulador capaz de comunicarse con un equipo de diagnóstico mediante el protocolo ISO9141-2, enviar y recibir tramas según el estándar, simular parámetros del vehículo en tiempo real, códigos de falla y luz indicadora de falla (MIL).

    - Selección del hardware a usar: microcontrolador que contenga el programa del simulador, componentes electrónicos para simulación e implementación del protocolo, equipo de diagnóstico y equipo visualizador de resultados.
    - Diseño e implementación de la interfaz OBD-Arduino, consiguiendo una línea bidireccional requerida por el protocolo.
    - Diseño e implementación del circuito simulador en placa de pruebas.
    - Diseño del circuito final en un software de diseño de PCB.
    - Fabricación de la placa PCB.
    - Implementación del programa del simulador, teniendo en cuenta todos los aspectos anteriores y las limitaciones hardware.

- Comprobación del correcto funcionamiento del simulador
    - Inicialización del protocolo.
    - Simulación y visualización de distintos parámetros PID en tiempo real, manualmente o por serie.
    - Simulación y visualización de distintos códigos de falla DTC, así como su borrado si procede.
    - Visualización del estado del vehículo mediante el indicador MIL.

2 ESTADO DEL ARTE

A lo largo de este capítulo se explicarán los conceptos teóricos necesarios para la realización del simulador. Veremos cómo ha evolucionado la electrónica en el ámbito automovilístico y los diferentes estándares del diagnóstico a bordo que se han ido creando a raíz de esta evolución.

También entenderemos qué es el indicador MIL del vehículo, el conector estandarizado que se halla en el mismo, la comunicación serie RS-232 para mandar los datos a una herramienta de diagnóstico, así como el micro que traduce esta comunicación a lenguaje OBD para que sea entendido por el coche en cuestión.

A continuación se explicarán los diferentes protocolos existentes haciendo especial hincapié en el ISO 9141-2 (ya que es el que implementaremos) así como los distintos modos de medición del protocolo.

Por último, se verán simuladores comerciales similares al implementado en este proyecto.

2.1. Historia de la electrónica en el automóvil

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

Iniciando el siglo XX, las innovaciones mecánicas siguieron sin afectar al diseño básico de los motores, suponiendo tan solo la adición de elementos orientados a la optimización de los mismos. Es a finales de los 70 cuando se empieza a incorporar la electrónica a los automóviles. Se añadieron los primeros sensores a los motores para verificar su correcto funcionamiento y también se añadieron unidades de control del motor que manejaban dichos sensores. El objetivo inicial de estos elementos electrónicos era el control de las emisiones de gases contaminantes y facilitar la diagnosis de averías.

A partir de la década de los 80 la mayor parte de las innovaciones provienen principalmente de la incorporación de la electrónica, y no de la incorporación de mejoras mecánicas. Se añadieron gran cantidad de sensores y se mejoraron las unidades de control del motor. Hoy en día, un automóvil puede incorporar más de 200 sensores y más de una unidad de control. Hay unidades de control para el motor, aire acondicionado o bolsa de aire, entre otros.

Las primeras unidades de control eran Módulos de Control de Motor o ECM (Engine Control Module), con el tiempo estas ECMs se hicieron más complejas y pasaron a convertirse en Unidades de Control Electrónico o ECU (Electronic Control Unit), estas ECUs son las conocidas como centralitas o UCEs (siglas en español, Unidad de Control Electrónico).

Actualmente los sensores se encargan de la medición de temperaturas, presiones, rotaciones, volúmenes, y gran cantidad de parámetros de funcionamiento automotor. La información que captan los sensores es enviada y almacenada en las centralitas.
Toda esta información permite que el propio automóvil monitoree su estado. En realidad, los sensores se limitan a detectar una serie de valores que envían a las centralitas y una vez allí son comparados con los valores óptimos que están almacenados en las memorias.
Cuando se encuentra un valor incorrecto, la centralita notifica un fallo avisando al conductor de alguna forma (indicadores luminosos, sonidos, etc.) y los fallos quedan almacenados para su posterior verificación por el personal autorizado.

A los automóviles nuevos fabricados hoy en día se les exige, por ley, disponer de una interfaz por medio de la cual se pueda obtener información o bien un diagnóstico del automóvil por medio de un equipo de prueba. Factores tales como la mejora de seguridad y la comodidad, son cada vez más complejos en los vehículos modernos. Como los fabricantes suben cada vez más las categorías e incluso los costes de los servicios y pocas veces disponemos de personal calificado en el taller para la realización de pruebas, los sistemas de diagnóstico de coches se han convertido en esenciales para garantizar rapidez, seguridad y un chequeo económico en los servicios y reparaciones de automóviles. [1]

2.2. Sistema de diagnóstico a bordo (OBD)

Para combatir los problemas de polución en Los Ángeles, el estado de California exigió sistemas de control de emisiones de gases en los modelos de automóvil posteriores a 1966. El Gobierno Federal de los Estados Unidos extendió estos controles a toda la nación en 1968. El Congreso aprobó el Clean Air Act (Acta Anti polución) en 1970 y creo la Agencia de Protección Medioambiental o EPA (Environmental Protection Agency). La EPA inicio el desarrollo de una serie de estándares en la emisión de gases y unos requerimientos para el mantenimiento de los vehículos con el fin de reducir la contaminación y ampliar su vida útil.

Para cumplir estos estándares se implementaron sistemas de encendido y alimentación controlada de combustible con sensores que median las prestaciones del motor y ajustaban los sistemas para conseguir una mínima polución. Estos sensores también permitían una cierta ayuda en la reparación.

2.2.1 Estándar OBD

En abril de 1985 un organismo estatal de California, el CARB (California Air Resources Board), aprobó una regulación para un sistema de diagnóstico a bordo u OBD (On-Board Diagnostic) [2]. Esta regulación que se aplica a los automóviles vendidos en el estado de California a partir de 1988, especifica que el Modulo de Control de Motor o ECM (Engine Control Module) debe monitorizar ciertos componentes del vehículo relacionados con las emisiones de gases para asegurar un correcto funcionamiento, y que se ilumine una lámpara Indicadora de Fallo o MIL (Malfunction Indicator Lamp) en el cuadro de instrumentos cuando se detecte un problema. El sistema OBD también aporta un sistema de Códigos de Error de Diagnóstico o DTC (Diagnostic Trouble Codes) y unas tablas de errores en los manuales de reparación para ayudar a los técnicos (mecánicos) a determinar las causas más probables de avería en el motor y problemas en las emisiones.

Los objetivos básicos de esta regulación son fundamentalmente dos:

- Reforzar el cumplimiento de las normativas de la regulación de la emisión de gases alertando al conductor cuando se presenta un fallo.

- Ayudar a los técnicos de reparación de automóviles en la identificación y reparación de fallos en el sistema de control de emisiones.

La autodiagnosis OBD se aplica a sistemas que se consideran responsables de un incremento en las emisiones de gases de escape en caso de avería y principalmente el sistema OBD chequea estos componentes:

- Los sensores principales del motor
- El sistema de medición del combustible
- Función de Recirculación de Gases de Escape o EGR (Exhaust Gas Recirculation)

2.2.2 Estándar OBDII

Los Sistemas de Diagnóstico a Bordo u OBD se encuentran en la mayoría de automóviles y vehículos ligeros actuales. Durante la década de los 70 y principio de los 80 se introdujeron componentes electrónicos para cumplir los estándares de emisión de gases de la EPA, posteriormente la implantación de sistemas OBD para controlar funciones del motor y diagnosticar problemas supuso una mayor complejidad en la electrónica integrada en los vehículos. A través de los años los sistemas OBD se han hecho más sofisticados y así el OBD-II es un nuevo estándar introducido a mediados de los 90 que aporta un control casi completo de chequeo del motor y también monitoriza partes del chasis y otros dispositivos del vehículo.

Inicialmente hubo varios estándares y cada fabricante tenía sus propios sistemas y códigos. En 1988 la Sociedad de Ingenieros de Automoción o SAE (Society of Automotive Engineers) definió un conector estándar OBD de 16 pines y un conjunto de códigos de diagnóstico, adoptando la EPA la mayoría de estándares y recomendaciones SAE sobre aplicaciones OBD. Posteriormente, con OBD-II, un conjunto más amplio de estándares y sistemas también definidos por la SAE y adoptado por la EPA y el CARB es aprobado para su implementación el 1 de enero de 1996.

Los componentes del sistema OBD-II son: la ECU (Engine Control Unit) conocida como la computadora del automóvil, los transductores encargados de enviar los datos hacia la ECU, la luz indicadora de fallas ubicada en el tablero (MIL, Malfunction Indicator Light) y el conector de diagnóstico (DLC, Data Link Connector) que sirve de interfaz entre la ECU y los dispositivos de diagnóstico automotriz.

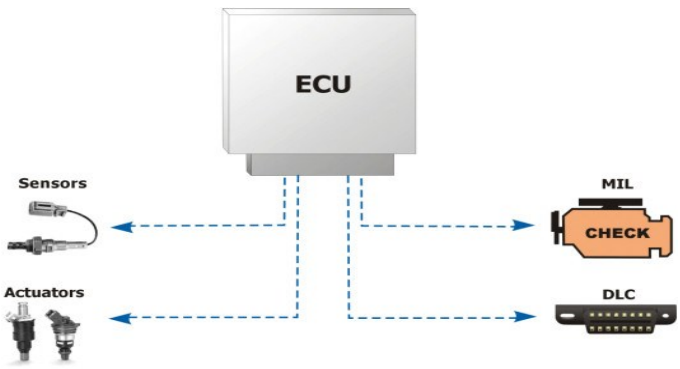


Fig. 1 – Esquema del sistema OBD2 del vehículo

2.2.3 EOBD (European On Board Diagnostic)

El EOBD [3] es un conjunto de normas parecida a la OBD II que ha sido implantada en Europa a partir del año 2000. Una de las características innovadoras es el registro del tiempo de kilometraje desde la aparición de un defecto hasta su diagnóstico. La normativa Europea obliga a los fabricantes a instalar sistemas de diagnosis compatibles con los americanos, con conectores e interfaces estandarizados. Los fabricantes también estarán obligados a publicar detalles de las partes importantes de sus sistemas de diagnóstico, de los cuales hasta ahora han sido propietarios. Las directrices de la Unión Europea se aplican a motores de explosión (motores de gasolina) registrados en el 2000 y posteriores y a motores Diésel registrados en 2003 y posteriores.

Hoy en día los fabricantes, estando obligados a instalar estos puertos de diagnóstico, han ampliado sus funciones para poder controlar y gestionar muchos más aspectos cotidianos del vehículo. A través de dicho puerto, se puede leer cualquier código de error que haya registrado la centralita, activar o desactivar funciones del vehículo, solicitar a la centralita del vehículo que realice testeos en todos los sistemas: cuadro de mandos, ABS, inyección, encendido, etc., reduciendo así los tiempos de taller para la búsqueda de un problema. Además de varias utilidades más que se pueden suponer y no están confirmadas (ej.: reprogramación de la centralita para aumento de potencia).

Control en los motores de gasolina

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

Control en los motores diésel

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

2.2.4 Luz indicadora de fallas (MIL)

El testigo MIL es utilizado por el sistema OBD-II y se enciende cuando los transductores del motor detectan un problema en el automóvil. Su propósito es alertar al conductor de la necesidad de realizar un mantenimiento del mismo.

La luz tiene dos etapas: estable (lo que indica un fallo menor, como un casquete de gas suelto o del sensor de oxígeno) e intermitente (que indica un fallo grave y podría dañar el convertidor catalítico si no se corrige durante un período prolongado). Cuando el testigo MIL se ilumina, la unidad de control del motor almacena un código de error relacionado con el mal funcionamiento, que se puede recuperar con la herramienta de análisis de diagnóstico.

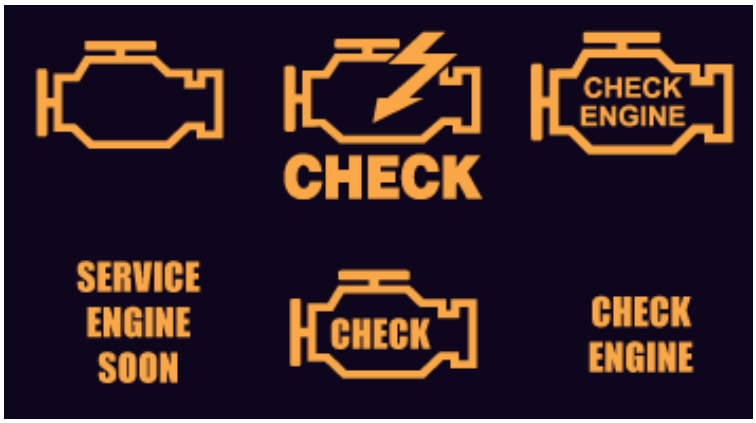


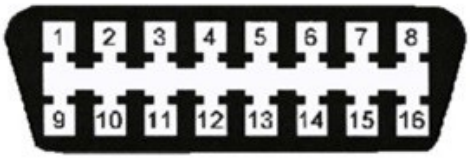
Fig. 2 – Variantes del testigo MIL

2.2.5 Conector OBD

El conector del sistema OBDII tiene que cumplir una serie de especificaciones según la normativa ISO 15031-3:2016 [4]. La normativa estipula que el conector para diagnóstico de OBDII o EOBD, debe de estar situado en el compartimento de los pasajeros, cerca del asiento del conductor. Esto es lo contrario a los sistemas anteriores donde el conector estaba en el compartimento motor. El conector estará situado detrás del cenicero o debajo del panel de instrumentos o en la consola central detrás de una tapa que lo cubre.

Fig. 3 – Conector OBD en el vehículo

El estándar SAE J1962 [5] define el conector físico usado para la interfaz OBD2. El sistema utiliza un conector de 16 pines, aunque no todos están ocupados.



1 – Sin uso	9 – Sin uso
2 - J1850 Bus positivo	10 - J1850 Bus negativo
3 – Sin uso	11 – Sin uso
4 - Tierra del Vehículo	12 – Sin uso
5 – Tierra de la Señal	13 – Tierra de la señal
6 - CAN High	14 - CAN Low
7 - ISO 9141-2 - Línea K	15 - ISO 9141-2 - Línea L
8 – Sin uso	16 - Batería - positivo

Fig. 4 - Pinout del conector OBD

2.2.6 Interfaz RS232

También se conoce como Electronic Industries Alliance RS-232C, y es una interfaz que designa una norma para el intercambio serie de datos binarios entre un DTE (Data Terminal Equipment, Equipo terminal de datos) y un DCE (Data Communication Equipment, Equipo de Comunicación de datos), aunque existen otras situaciones en las que también se utiliza la interfaz RS-232, una de ellas el uso automotriz.

El RS-232 [6] consiste en un conector tipo DB-25 (de 25 terminales), aunque es normal encontrar la versión de 9 terminales (DE-9), más barato e incluso más extendido para cierto tipo de periféricos (como el ratón serie del PC). En el uso automotriz no es necesario usar un conector de 25 terminales, basta con un DB-9 dado que solo usamos pocas líneas de comunicación de las 16 del conector J1962 del automóvil.

Las señales con las que trabaja este puerto serie son digitales, de +12V (0 lógico) y -12V (1 lógico), para la entrada y salida de datos, y a la inversa en las señales de control. El estado de reposo en la entrada y salida de datos es -12V. Dependiendo de la velocidad de transmisión empleada, es posible tener cables de hasta 15 metros.

Fig. 5 – Conectores DB-25 y DE-9

2.2.7 ELM327 Intérprete de OBD2 a RS232

Casi a todos los automóviles nuevos producidos hoy en día se les exige, por ley, disponer de una interfaz de la cual el equipo de prueba pueda obtener una información de diagnóstico. La transferencia de datos en estas interfaces sigue varios estándares, de los cuales ninguno es directamente compatible con PCs o PDAs.

El ELM 327 [7] fue diseñado para actuar como puente entre los puertos de OBD y los puertos del estándar RS232. Es una versión mejorada a las interfaces ELM320, ELM322 y ELM323 agregando 7 protocolos CAN de ellos. El resultado es un circuito integrado que puede captar y convertir la mayoría de los protocolos comunes usados hoy en día. Entre otras mejoras también encontramos la opción de alta velocidad de RS232, monitoreo del voltaje en la batería o características a gusto del usuario a través de parámetros programables, por mencionar algunas.

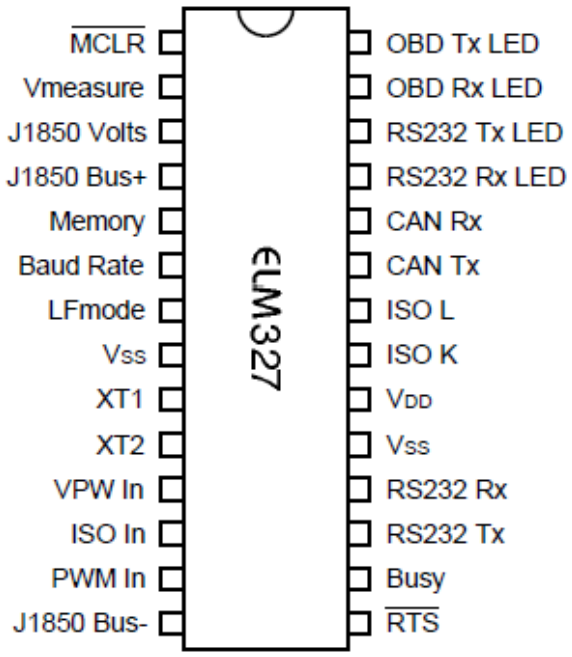


Fig. 6 – Pinout ELM327

Características

- Soporta 12 protocolos
- Busca automáticamente un protocolo
- Completamente configurable con comandos AT
- Tasa de transferencia de RS232 de 500Kbps
- Voltaje de entrada de la batería monitoreada.
- Bajo consumo del diseño CMOS.

Aplicaciones

- Lector de diagnóstico de códigos de falla.
- Herramienta de escaneo automotriz
- Conocimiento de herramientas auxiliares

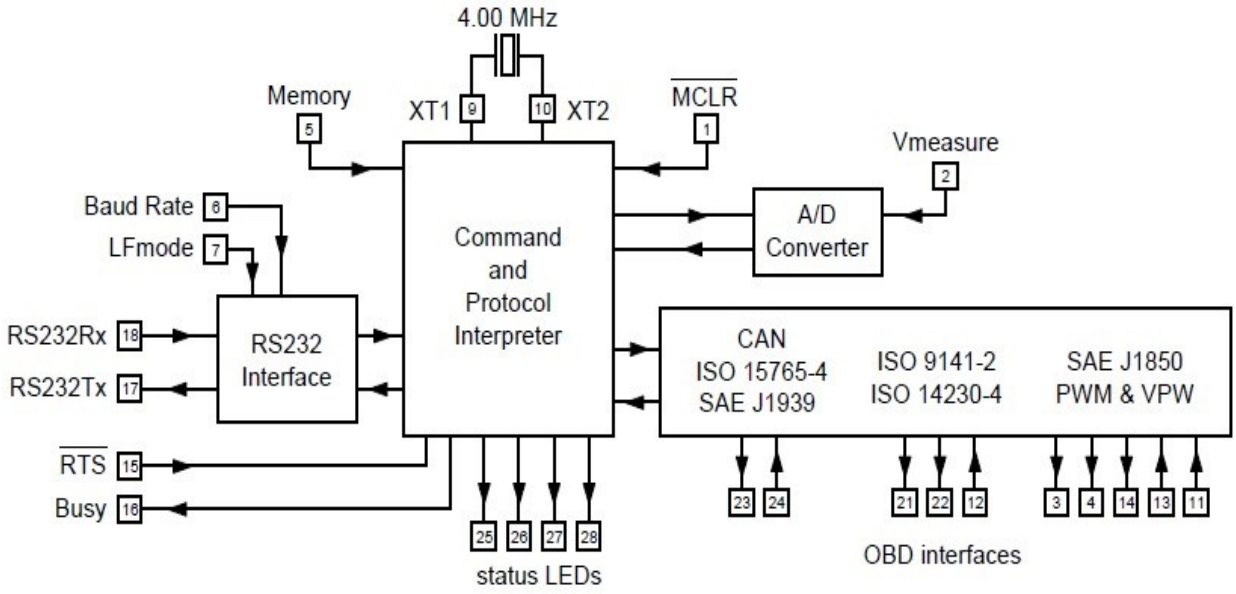


Fig. 7 – Diagrama de bloque del integrado ELM327

2.3. Protocolos de comunicación

Al comienzo de la etapa automotriz cada fabricante usaba su propio sistema de auto diagnostico a bordo (OBD) con su protocolo de comunicación y un conector único para el sistema de diagnóstico. Esto hacía que los técnicos tuvieran que adquirir distintos equipos que cubrieran los diferentes protocolos y contar con los conectores para dichas marcas.

Como solución al problema, la EPA (Agencia De Protección Al Ambiente) estableció una norma que obligaba a todos los vehículos vendidos en USA a partir de 1996 contar con un conector trapezoidal de 16 pines para el sistema de auto diagnostico conocido hoy como OBD2 (siendo los vehículos del 95 hacia atrás conocidos como OBD1) para abarcar cualquier protocolo de comunicación con un mismo dispositivo tester.

En la Tabla 1 se muestran los distintos protocolos, así como su modo de comunicación y vehículos que lo soportan.

Protocolo	Modo	Marcas
SAE J1850 VPW [8]	Modulación por ancho de pulso variable	General Motors
SAE J1850 PWM	Modulación por ancho de pulso	Ford, Lincoln y Mercury
ISO 9141-2 [9] ISO 14230-4 (KWP2000) [10]	Comunicación serie	Chrysler, Jeep, Dodge, Nissan, Volvo, Mitsubishi y mayoría de vehículos Europeos y Asiáticos
ISO 15765-4 (CAN BUS) [11]	Red de área de controlador	BMW y vehículos americanos a partir de 2008

Tabla 1 – Protocolos OBD2

A continuación, en la Tabla 2 se muestra las diferentes capas del modelo de referencia OSI (Open System Interconnection).

Nivel OSI	PROTOCOLO OBD2			
	SAE J1850 VPW/PWM	ISO 9141	ISO 14230-4 (KWP2000)	ISO 15765-4 (CAN BUS)
Aplicación (Nivel 7)	SAE J1979/ ISO 15031-5	SAE J1979/ ISO 15031-5	ISO 14230-3	SAE J1979/ ISO 15031-5
Presentación (Nivel 6)	---	---	---	ISO 15765-4
Sesión (Nivel 5)	---	---	---	---
Transporte (Nivel 4)	---	---	---	---
Red (Nivel 3)	---	---	---	ISO 15765-4
Enlace (Nivel 2)	SAE J1850	ISO 9141	ISO 14230-2	ISO 15765-4
Físico (Nivel 1)	SAE J1850	ISO 9141	ISO 14230-1	ISO 15765-4

Tabla 2 – Torre de protocolos OSI

En el presente proyecto se implementará el protocolo ISO 9141-2 ya que aunque se trata de uno de los pioneros, lo soportan actualmente la mayoría de los vehículos europeos y asiáticos. A continuación se explicarán los diferentes protocolos detallando en profundidad el que vamos a implementar.

2.3.1 SAE J1850 VPW y PWM

Es el estándar SAE [12] para las clases A y B (velocidad de transmisión baja y media). Es una combinación del SCP de Ford y del Protocolo Clase 2 de General Motors y fue aprobado por la SAE en 1988 y revisado finalmente en 1994.

Existen dos versiones (al ser desarrollo de dos protocolos propietarios), cuya diferencia consiste en la codificación de bit y la velocidad de transmisión.

- La versión más lenta emplea una codificación VPM (Variable Pulse Modulation – Modulación por ancho de pulso variable) alcanzando 10,4 kbit/s y transmite con un solo cable referido a masa.
- La versión más rápida usa una codificación PWM (Pulse Width Modulation – Modulación por ancho de pulso) consiguiendo 41,6 kbit/s y transmite en modo diferencial con dos cables.

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

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

La utilización en series comerciales empezó en el momento de su estandarización, siendo quizás el primer protocolo en ser aplicado de forma masiva y actualmente todavía está en uso.

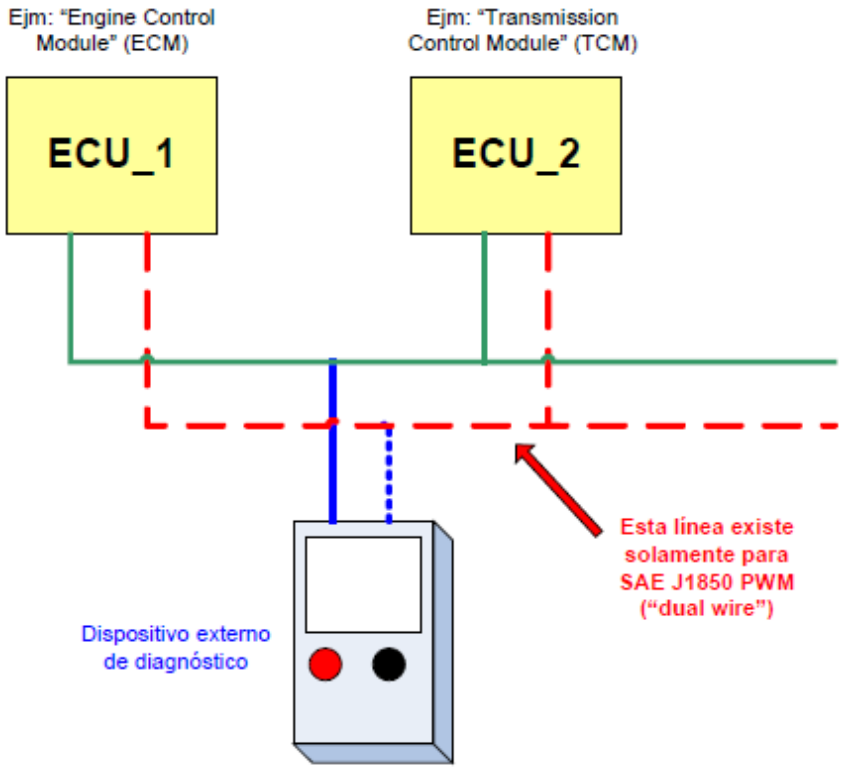


Fig. 8 – Topología de la red J1850

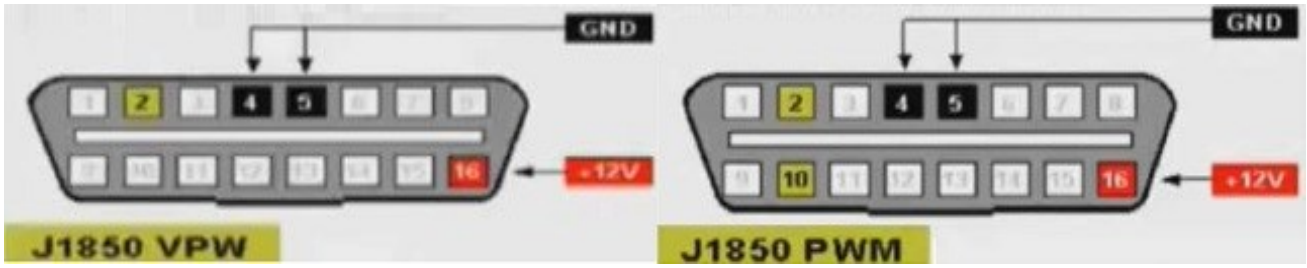


Fig. 9 – Pines usados por protocolo J1850 VPW/PWM

2.3.1.1 Nivel Físico

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

Si se transmite usando VPW, tendremos símbolos como los ilustrados en la Figura 10 con las características:

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

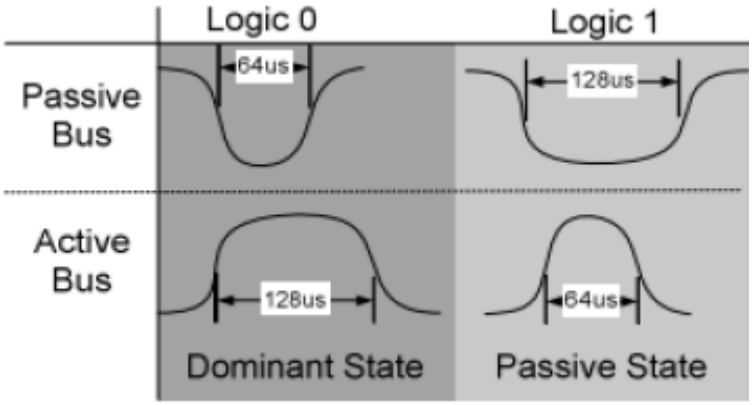


Fig. 10 – Símbolos en VPW

En caso de transmitir usando una modulación PWM, se tendrán símbolos como los ilustrados en la Figura 11 con las características:

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

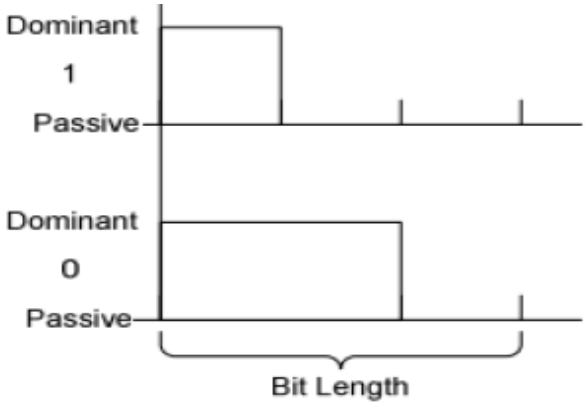


Fig. 11 – Símbolos en PWM

2.3.1.2 Nivel de Enlace

La estructura del mensaje es común a los dos métodos de modulación VPW y PWM. En la Figura 12 se muestra el formato de trama para el protocolo SAE J1850, a continuación se describe brevemente la utilidad de cada uno de sus campos:

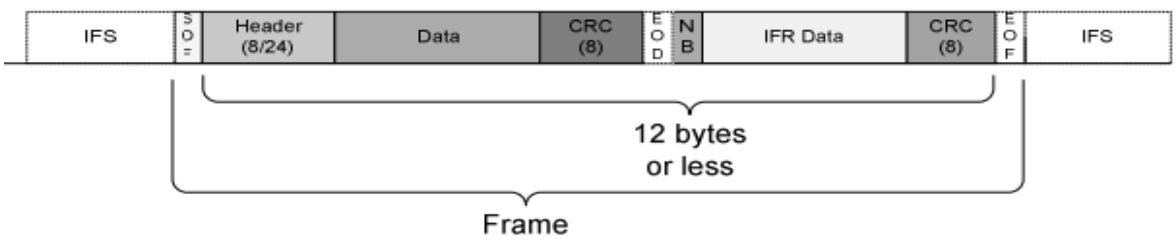


Fig. 12 – Estructura mensaje del protocolo J1850

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

BYTES DE CABECERA			BYTES DE DATOS						
PRIORIDAD/TIPO	DIRECCION DESTINO	DIRECCION ORIGEN	#1	#2	#3	#4	#5	#6	#7
Petición de Diagnostico a 10.4 Kbit/s - SAE J1850									
68	6A	F1	Máximo 7 bytes de datos						
Respuesta de Diagnostico a 10.4 Kbit/s - SAE J1850									
48	6B	ECU addr	Máximo 7 bytes de datos						
Petición de Diagnostico a 41.6 Kbit/s (SAE J1850)									
61	6A	F1	Máximo 7 bytes de datos						
Respuesta de Diagnostico a 41.6 Kbit/s (SAE J1850)									
41	6B	ECU addr	Máximo 7 bytes de datos						

Fig. 13 – Trama protocolo J1850

2.3.2 CAN – ISO 15765 (Red de área del controlador)

El protocolo CAN [13] fue desarrollado por Robert Bosch Gmbh, compañía alemana, a principios de los años 80 del siglo pasado y es la referencia obligada de cualquier protocolo en el campo del automóvil. Además, su aplicación se extiende al campo del control industrial. Existen dos versiones básicas de este protocolo: CAN 1.0 y CAN 2.0.

El estándar CAN 1.0 es un protocolo que emplea par trenzado como medio de transmisión, la codificación de bit es del tipo NRZ con bit stuffing cada 5 bits para evitar la desincronización de bit (si hay 5 bits del mismo nivel, el siguiente se fuerza de nivel contrario y no cuenta como dato para la trama). El método empleado para el acceso al

medio es el de contienda CSMA/CA y permite una velocidad de transmisión de hasta 125 kbit/s, lo que lo ubica en la clase B. Esta velocidad le permite realizar funciones de control.

Este protocolo está orientado a un modelo de funcionamiento de tipo productor/consumidor, si bien pueden emplearse otros como el tipo maestro/esclavo múltiple. En el primer modo, un nodo puede enviar un mensaje al bus cuando unilateralmente lo decida. El mensaje es finalmente transmitido si gana la contienda entre los nodos candidatos a transmitir en el instante de acceso. Esta contienda se realiza en base al valor del campo identificador de la trama y requiere el uso de dos niveles de señalización asimétricos de bus, denominados nivel recesivo (reposo) y nivel dominante.

El campo identificador (11 bits) de la trama es funcional indicando el contenido del campo de datos de la misma. Una característica importante de este protocolo es el reconocimiento de recepción (bit ACK) incluido en la propia trama (in frame response) útil cuando existe un único destinatario de la trama. También permite, incluida en la propia trama de envío, la petición de una respuesta de la estación destino (mediante el bit RTR).

Finalmente, merece una cierta consideración el confinamiento local de errores en un nodo ante reiteradas detecciones de error, es decir, cuando un nodo detecta 255 veces error en su transmisión, se desconecta del bus. Este protocolo fue estandarizado con la denominación ISO 11519.

En 1991 apareció CAN 2.0, una nueva versión del protocolo CAN que es compatible con el estándar CAN 1.0, cuya aportación consistió básicamente en aumentar el campo identificador de la trama de 11 bits a 29 bits, mediante la inclusión de un campo de identificación extendido de 18 bits. La velocidad se aumentó a 500 kbit/s al estandarizarse bajo SAE J2284-500 y a 1 Mbit/s bajo ISO 11898, lo que los ubica en la clase C (velocidad de transmisión alta).

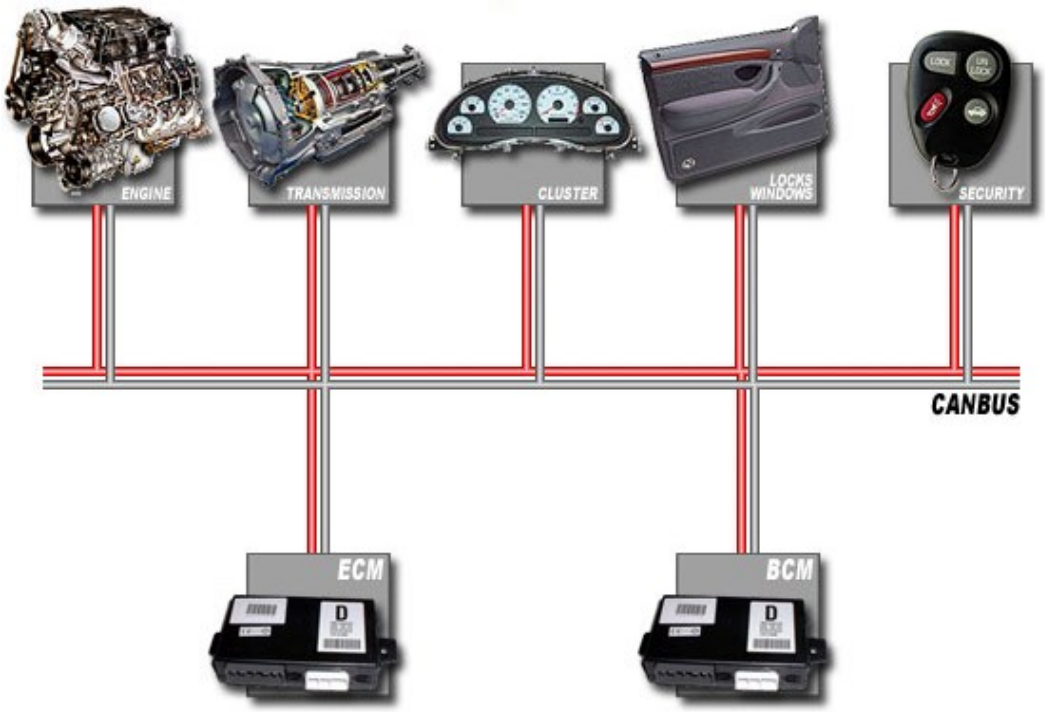


Fig. 14 – Topología del protocolo CAN

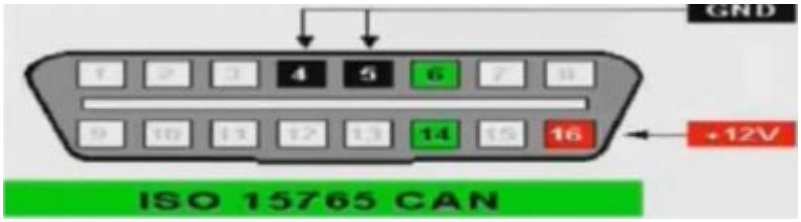


Fig. 15 – Pines usados por el protocolo CAN

De acuerdo al modelo de referencia OSI (Open Systems Interconnection), la arquitectura de protocolos CAN incluye tres capas: física, de enlace de datos y aplicación, además de una capa especial para gestión y control del nodo llamada capa de supervisor. A continuación se define cada una de ellas.

2.3.2.1 Nivel Físico

Define los aspectos del medio físico para la transmisión de datos entre nodos de una red CAN, los más importantes son niveles de señal, representación, sincronización y tiempos en los que los bits se transfieren al bus. La especificación del protocolo CAN no define una capa física, sin embargo, los estándares ISO 11898 establecen las características que deben cumplir las aplicaciones para la transferencia en alta y baja velocidad.

2.3.2.2 Nivel de Enlace

Define las tareas independientes del método de acceso al medio, además debido a que una red CAN brinda soporte para procesamiento en tiempo real a todos los sistemas que la integran, el intercambio de mensajes que demanda dicho procesamiento requiere de un sistema de transmisión a frecuencias altas y retrasos mínimos. En redes multimaestro, la técnica de acceso al medio es muy importante ya que todo nodo activo tiene los derechos para controlar la red y acaparar los recursos. Por lo tanto, la capa de enlace de datos define el método de acceso al medio así como los tipos de tramas para el envío de mensajes.

2.3.2.3 Nivel de Supervisor

La sustitución del cableado convencional por un sistema de bus serie presenta el problema de que un nodo defectuoso puede bloquear el funcionamiento del sistema completo. Cada nodo activo transmite una bandera cuando detecta algún tipo de error y puede ocasionar que un nodo defectuoso pueda acaparar el medio físico. Para eliminar este riesgo el protocolo CAN define un mecanismo autónomo para detectar y desconectar un nodo defectuoso del bus, dicho mecanismo se conoce como aislamiento de fallos.

2.3.2.4 Nivel de Aplicación

Existen diferentes estándares que definen la capa de aplicación; algunos son muy específicos y están relacionados con sus campos de aplicación. Entre las capas de aplicación más utilizadas cabe mencionar CAL, CAN open, Device Net, SDS (Smart Distributed System), OSEK y CAN Kingdom.

2.3.3 ISO 9141-2

El protocolo ISO 9141 [14] (y su posterior revisión 9141-2) es muy parecido al protocolo RS-232 ya que es orientado a byte, a diferencia del SAE J1850 que está orientado a mensajes con símbolos de encabezado, datos y CRC. Los mensajes de diagnóstico se forman en la capa de aplicación (SAE J1979) y son enviados byte a byte por la capa de enlace.

En la Figura 16 se muestra la arquitectura del protocolo. Las líneas K y L son independientes y referenciadas a tierra. La línea K trabaja en dos direcciones (bidireccional) y se usa para transmitir los mensajes de solicitud y respuesta del estado de los parámetros; La línea L es unidireccional y sólo se usa en el proceso de inicialización en determinados vehículos que lo soportan (en este proyecto únicamente usaremos la línea K).

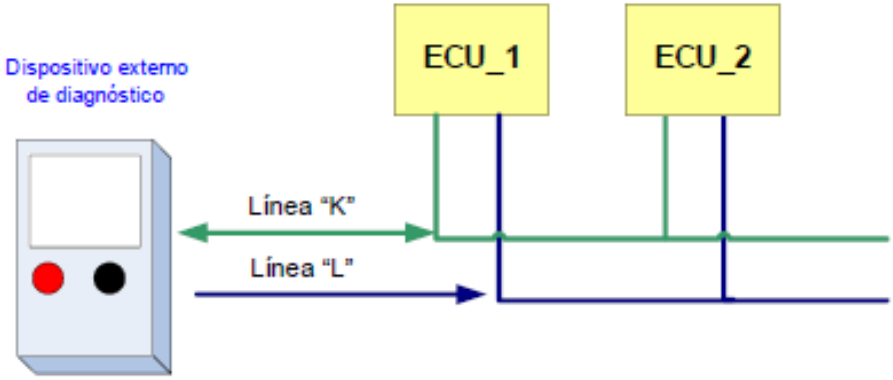


Fig. 16 – Topología del protocolo ISO 9141

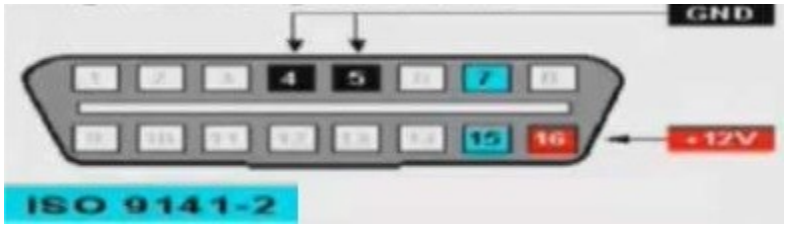


Fig. 17 – Pines usados por el protocolo ISO 9141-2

2.3.3.1 Nivel Físico

En ISO 9141 sólo existen dos símbolos: el "0" y el "1" y la codificación es NRZ (Not Return Zero), por lo que el voltaje no vuelve a cero entre dos valores consecutivos a uno. El "1" lógico se representa con un nivel igual al voltaje positivo de la batería ($V_B = 12V$)y el "0" lógico con el nivel de tierra. La Figura 18 muestra el rango de valores que deben tener los símbolos transmitidos para ser interpretados correctamente por el receptor.

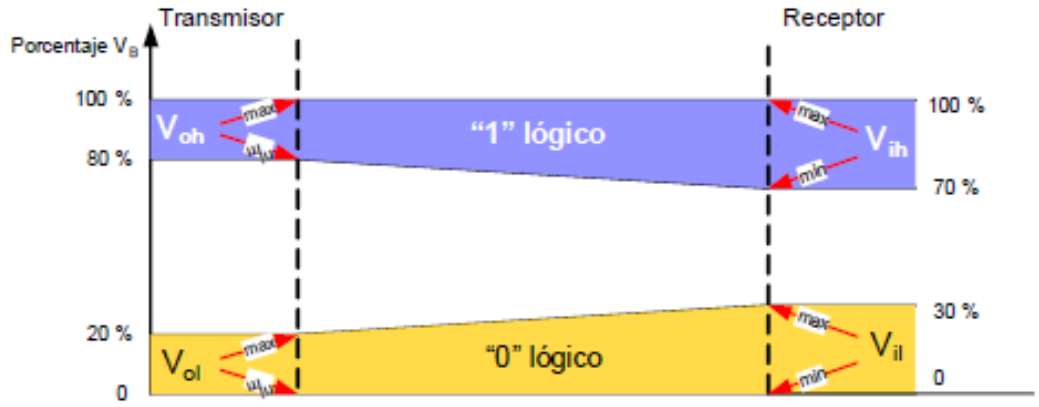


Fig. 18 – Niveles de tensión según el valor lógico en ISO 9141

ISO 9141 permite un rango muy amplio de tasas de bits: 10 baudios a 10.000 baudios. Sin embargo, en la segunda revisión del estándar (ISO 9141-2) se fijó la tasa de bits en 10.4 Kbps, originando un tiempo de bit de 96,15 microsegundos. El tiempo máximo de transición entre dos niveles lógicos (del 20% al 80% de V_B y viceversa) no debe exceder el 10% del tiempo de bit.

2.3.3.2 Nivel de Enlace

Las tramas a nivel 2 de capa son muy parecidas a las usadas en el protocolo RS-232 y están formadas por un bit de inicio (Start), 8 bits de datos y por lo menos un bit de parada (stop). Cuando el bus está inactivo, el estado de las líneas K y L es "1" lógico, por lo que el bit de Start debe ser "0" y el del stop "1". Los bits de datos se envían comenzando con el menos significativo.

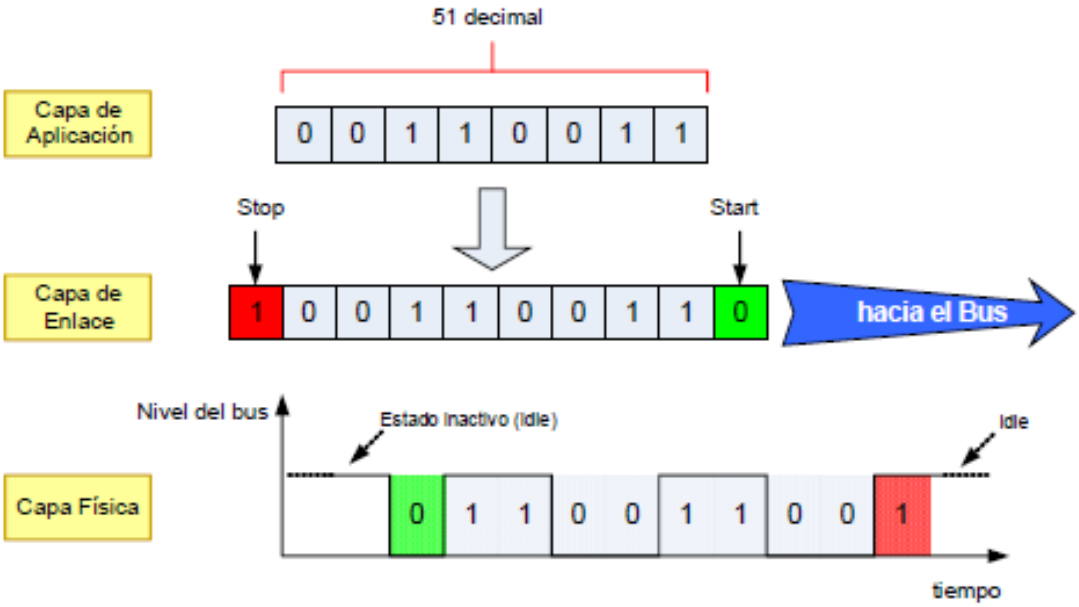


Fig. 19 – Trama ISO 9141 en los 3 niveles de capa OSI

A diferencia del protocolo SAE J1850, sólo se usa para la comunicación entre el dispositivo de diagnóstico (tester) con las ECU y además, esta comunicación debe ser inicializada. El protocolo de inicialización es el siguiente:

- El tester envía a la ECU la dirección $33 a 5 bit/s tal y como se observa en la Figura 20 (el tiempo de transmisión total es de 2 segundos).

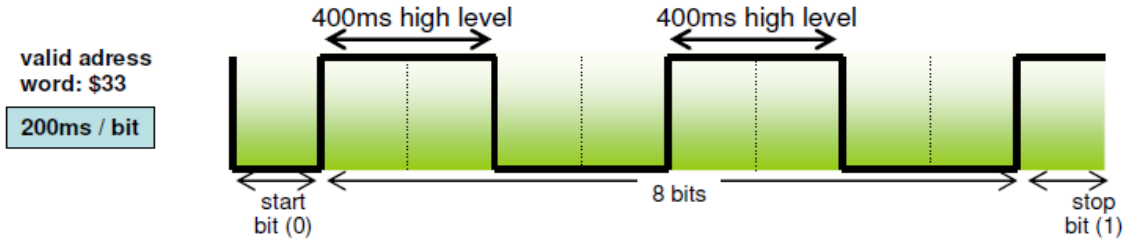


Fig. 20 – dirección $33 a 5 bit/s

- La ECU responde con el byte de sincronización $55 informando al tester de la nueva tasa de baudio que será ahora 10.4 kbps.
- La ECU espera a que el tester reconfigure la nueva tasa y a continuación envía dos palabras claves (Keywords, KW). Existen dos variedades en cuanto a las KWs; Si la ECU envía $08 $08 se indica que pueden existir uno o más ECUs en la red y se configura el tiempo P_2= [25-50 ms]. Si se envía $94 $94 se indica que sólo existe una única ECU en la red de diagnóstico y se configura el tiempo P_2= [0-50 ms]. Para entender estos tiempos consultar la Tabla 4.
- Como confirmación de la recepción de las palabras clave, el tester envía el segundo KW invertido ($F7 para KW_2= $08 o $6B para KW_2= $94).
- Por último, la ECU envía la inversa de la dirección de inicialización $33 que se envió a 5 bit/s ($CC) finalizándose así el protocolo de inicialización y estando ambos dispositivos preparados para la comunicación.

El comentado proceso de inicialización puede apreciarse en la siguiente Figura 21 y los tiempos entre bytes están recogidos en la Tabla 3.

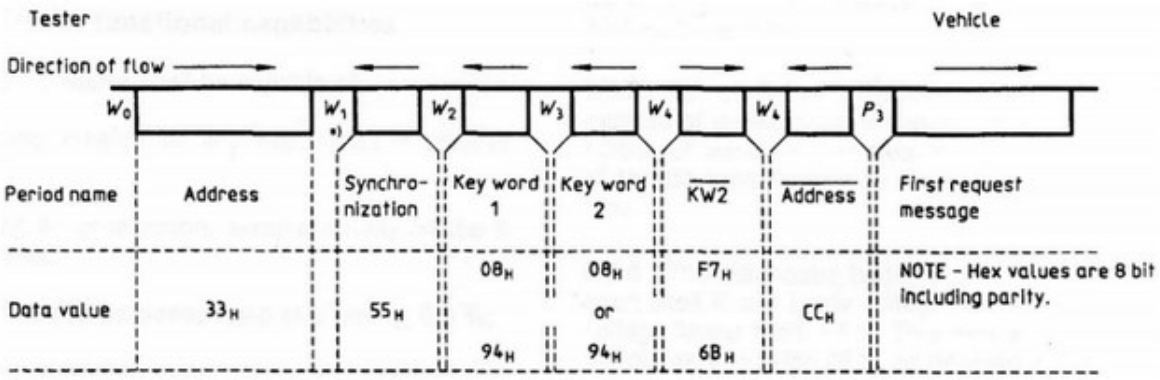


Fig. 21 - Proceso de inicialización ISO 9141-2

Símbolo	Tiempo (ms)		Descripción
	mín.	máx.	
W_0	2	-	Tiempo bus desocupado hasta el envío del byte de dirección
W_1	60	300	Tiempo desde el final del byte de dirección hasta el comienzo del patrón de sincronización
W_2	5	20	Tiempo desde el final del patrón de sincronización hasta el comienzo del KW_1
W_3	0	20	Tiempo entre los dos KWs
W_4	25	50	Tiempo entre el final del KW_2 y su inversión enviada por el tester
W_5	300	-	Tiempo del bus desocupado hasta que el tester retransmita el byte de dirección

Tabla 3 – Tiempos entre bytes en la inicialización ISO 9141-2

A partir de la inicialización comienzan a mandarse peticiones por parte del tester y respuestas por parte de la(s) ECU(s) como se observa en la siguiente Figura 22 y los tiempos entre datos se hayan recogidos en la Tabla 4.

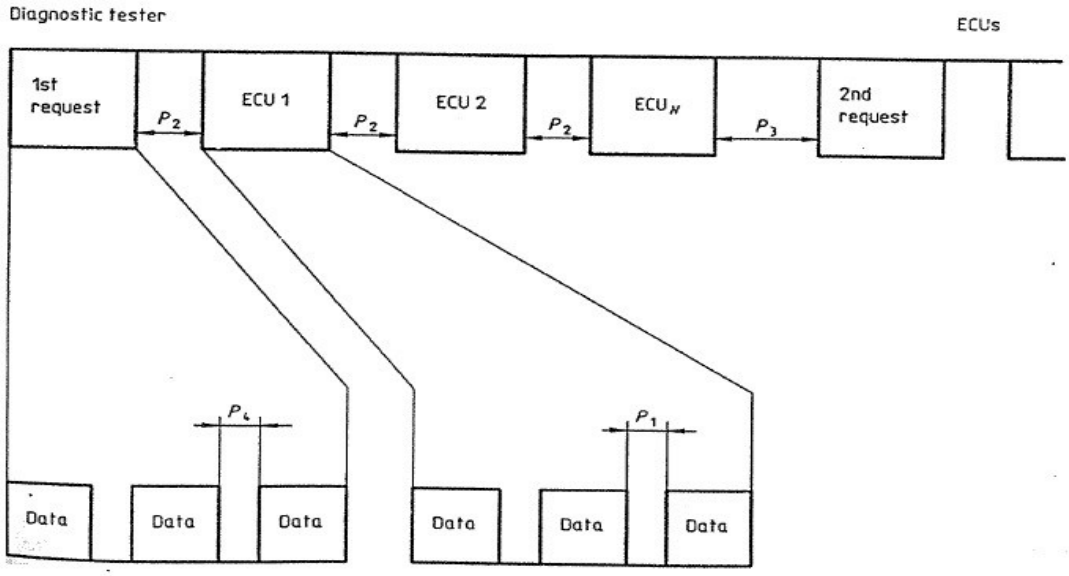


Fig. 22 – Formato de peticiones y respuestas ISO 9141-2

Símbolo	Tiempo (ms)		Descripción
	mín.	máx.	
P_1	0	20	Tiempo entre bytes para mensajes de la ECU al tester
P_2 (94)	0	50	Tiempo entre mensajes para vehículos con KW 94
P_2 (08)	25	50	Tiempo entre mensajes para vehículos con KW 08
P_3	55	5.000	Tiempo entre mensajes desde la última respuesta de la ECU a la siguiente petición del tester
P_4	5	20	Tiempo entre bytes para mensajes del tester a la ECU

Tabla 4 – Tiempos entre datos en la comunicación ISO 9141-2

Los primeros tres bytes de todos los mensajes de diagnóstico son los bytes de cabecera. El primer byte depende de la velocidad de tráfico binario (bit rate) del enlace de datos (data link) y del tipo de mensaje. El segundo byte de cabecera tiene un valor que depende del tipo de mensaje que es, petición o respuesta. El tercer byte es la dirección física del remitente del mensaje. El equipo externo de diagnóstico siempre tiene la dirección $F1. Los siete bytes siguientes son los bytes de datos y ellos varían dependiendo del diagnóstico específico de servicio. Por último se añade un código de redundancia cíclica (CRC, CS o checksum) que consiste en:

Si el mensaje es <1> <2> <3>…<N>,<CS>, donde $< i > (1 \leq i \leq N)$ es el valor numérico del iésimo byte del mensaje, entonces:

$$< CS > = < CS >_N$$

Donde $< CS >_i$ (i = 2 a N) se define como:

$$< CS >_i = \{ < CS >_{i-1} + < i > \} \text{ Módulo 256}$$

Y siendo $< CS >_1 = < 1 >$.

En la siguiente Tabla 5 queda recogido el formato de la trama ISO 9141-2.

Secuencia	Descripción	Bytes (hexadecimal)		Referencia
		Tester a ECU	ECU a tester	
Byte 1	Cabecera 1	68	48	SAE J1979
Byte 2	Cabecera 2	6A	6B	SAE J1979
Byte 3	Dirección de la fuente	F1	Dirección ECU	SAE J1979
Bytes 4 a 10	Datos (hasta 7 bytes)	XX	XX	SAE J1979
Final byte	Checksum	YY	YY	Comentado anteriormente

Tabla 5 – Formato trama ISO 9141-2

Si la ECU recibe una trama con estructura de mensaje incorrecta o checksum no válido, no responde ante la petición por lo que el tester considerará error de tiempo incorrecto y retransmitirá el mensaje. Si es el tester el que recibe la trama incorrecta retransmitirá el mensaje original después de agotar el tiempo P3min.

2.3.4 ISO 14230-4 / KWP2000

El estándar ISO 14230 [15], más conocido como KWP2000 (Keyword Protocol 2000), es un protocolo de comunicación serie muy similar al ISO 9141. Al igual que el protocolo ISO 9141 basado en un bus bidireccional sobre una única línea (K-line), opcionalmente puede haber una segunda línea (L-line) para las señales de llamada o wakeup. La tasa de datos puede oscilar entre los 1,2 y los 10,4 kbps.

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

La codificación en KWP2000 es NRZ y la velocidad de 10.4 kbps, al igual que en ISO9141-2. El envío se hace byte a byte, pero en KWP2000 existe una estructura ya predeterminada para los datos, a diferencia de ISO 9141-2 que se adapta al protocolo de aplicación. KWP2000 dispone de dos modos diferentes de inicialización:

- KWP a 5 bit/s es idéntica a la inicialización de ISO9141-2 excepto por las palabras clave que en este caso hay 19 diferentes y son de la forma 8F XX, donde XX describe el formato de la cabecera. La norma ISO 14230-4 sólo permite los siguientes cuatro: 8F E9, 8F 6B, 8F 6D y 8F EF y además se recomienda usar 8F E9 para la comunicación entre tester y vehículo.
- KWP de inicialización rápida trabaja a 10.4 kbps. La secuencia inicial empieza con un mensaje de "wake-up" del tester (50ms de duración) seguido de una petición de StartCommunication. El vehículo responderá entonces una respuesta de StartCommunication. La petición de StartComunnication es de la forma C1 33 F1 81 66, donde C1 es el byte de formato, 33 es la dirección destino, F1 la dirección origen, 81 el ID de servicio de petición y 66 el checksum. La respuesta es similar pero el byte de formato es de la forma 10 LL LLLLb, donde LL LLLL es el número de bytes de datos. Para este ejemplo, el byte de formato sería $83 = 10 00 0011b (3 bytes de datos) y por tanto la respuesta StartComunnication sería 83 F1 10 C1 YY ZZ CS, con F1 la dirección destino, 10 la dirección origen (en este caso), C1 el ID de servicio de respuesta, YY ZZ los KW explicados anteriormente y CS el checksum.

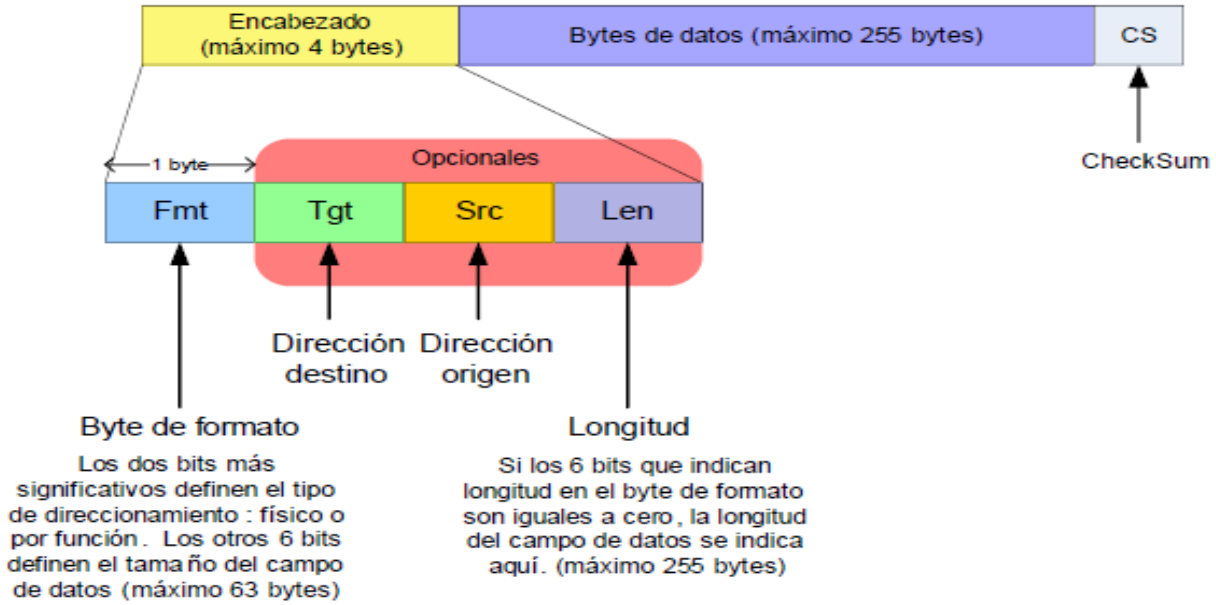


Fig. 23 – Formato trama KWP2000

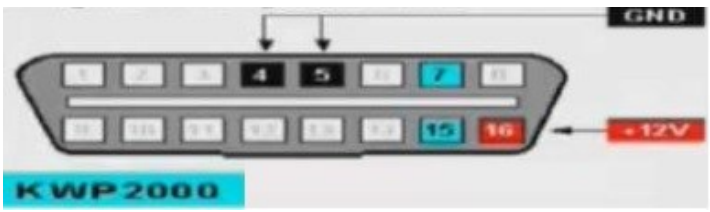


Fig. 24 – Pines usados por el protocolo KWP2000

2.4. Modos de medición

Todos los estándares antes mencionados implementan varios modos de trabajo, es decir, según la parte de información a la que queramos acceder necesitamos utilizar un modo diferente, dentro de cada uno de ellos podemos usar un abanico de parámetros muy amplio.

Los modos de medición [16] son comunes a todos los vehículos (estandarizado por SAE J1979) y permiten desde registrar datos para su verificación, extraer códigos de averías, borrarlos y realizar pruebas dinámicas de actuadores. Un software de diagnosis se deberá encargar de presentar los datos y facilitar la comunicación.

El formato estándar de trama es de 7 bytes y aunque no se suelen usar todos se recomienda que el resto se pongan a $00 o $55. Los modos en que se presenta la información se hayan estandarizados y son los siguientes:

2.4.1 Modo 01 - Obtención de datos actualizados

Es el acceso a datos en tiempo real de valores analógicos o digitales de salidas y entradas a la ECU. Este modo es también llamado flujo de datos. Aquí es posible ver, por ejemplo, la temperatura de motor o el voltaje generado por una sonda lambda entre otra gama extensa de parámetros.

La información en este modo incluye un parámetro de identificación (PID o Parameter ID) que indica al sistema de diagnóstico de a bordo la información específica requerida. Existe una gran variedad de ellos y según el sistema a diagnosticar se soportarán unos u otros. Existen una serie de PIDs (PID $00, PID $20, PID $40) para conocer cuáles de ellos son soportados. Todas las ECUs deben soportar por defecto el PID $00.

En la Tabla 6 podemos observar el estándar de una trama de petición en este modo. Aunque es posible solicitar varios PIDs en una misma petición, lo normal es pedir sólo uno, ya que por cada PID solicitado la ECU debe mandar una respuesta distinta.

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

Tabla 6 – Mensaje de petición del Modo 01

En la Tabla 7 se muestra el formato de respuesta, en el cual se indica que respondemos al modo 01 sumándole 40 en hexadecimal (0x41). Los datos solicitados se traducen en A, B, C y D y según la información a pedir se usan todos o sólo algunos.

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una respuesta en el Modo 01	41
#2	Valor hexadecimal del PID al que se está respondiendo	XX
#3	Dato A	XX
#4	Dato B	XX
#5	Dato C	XX
#6	Dato D	XX

Tabla 7 – Mensaje de respuesta del Modo 01

Aunque existen más de 60 PIDs diferentes (véase Anexo B), a continuación se citarán los más relevantes en la Tabla 8. Se añade la fórmula para traducir los datos solicitados así como el número de bytes de datos.

PID	Bytes	Descripción	Min.	Max.	Uds.	Fórmula
00	4	PIDs soportados [01-20]	-	-	-	32 Bits: BitX=0 > PIDX Soportado BitX=1 > PIDX No sop.
01	4	Estado de MIL y número de DTCs almacenados.	-	-	-	Bits codificados
03	2	Estado del sistema de combustible	-	-	-	Bits codificados
04	1	Valor de carga del motor	0	100	%	A×100/255
05	1	Temperatura del refrigerante	-40	215	ºC	A-40
0A	1	Presión del combustible	0	765	kPa	A×3
0C	2	Revoluciones por minuto del motor	0	16.383	rpm	((A×256)+B)/4
0D	1	Velocidad del vehículo	0	255	Km/h	A
0F	1	Temperatura del aire de entrada	-40	215	ºC	A-40
11	1	Posición del acelerador	0	100	%	A×100/255
13 … 1B	2	Voltaje en los distintos sensores de oxígeno	0	1.275	V	A/200 (B-128)×100/128
1C	1	Estándar OBD conforme al vehículo	-	-	-	Bits Codificados
1F	2	Tiempo transcurrido desde el arranque	0	65.535	seg.	(A×256)+B
20	4	PIDs soportados [21-40]	-	-	-	32 Bits: BitX=0 > PIDX Soportado BitX=1 > PIDX No sop.
21	2	Distancia recorrida con el indicador MIL encendido	0	65.535	Km	(A×256)+B
24 … 2B	4	Voltaje equivalente de la sonda lambda	0	8	V	((A×256)+B)×R ((C×256)+D)×R Con R=2/65535
2C	1	Recirculación de gases de escape o EGR	0	100	%	A×100/255
2E	1	Depuración por evaporación	0	100	%	A×100/255
2F	1	Nivel de combustible	0	100	%	A×100/255

31	2	Distancia recorrida desde el último limpiado de errores	0	65535	km	(A×256)+B
34 … 3B	4	Corriente equivalente de la sonda lambda	-128	128	mA	((A×256)+B) /32.768 ((C×256)+D)/128
3C … 3F	2	Temperatura del catalizador	-40	6.513	ºC	((A×256)+B)/10-40
40	4	PIDs soportados [41-60]	-	-	-	32 Bits: BitX=0 > PIDX Soportado BitX=1 > PIDX No sop.
42	2	Módulo de control de voltaje	0	65.535	V	(A×256)+B
46	1	Temperatura ambiente	-40	215	ºC	A-40
4D	2	Tiempo de marcha mientras el indicador MIL está encendido	0	65.535	min.	(A×256)+B
51	1	Tipo de combustible	-	-	-	Bits codificados
5B	1	Nivel de batería híbrida	0	100	%	A×100/255

Tabla 8 - PIDs más relevantes del modo 01

2.4.2 Modo 02 – Acceso a cuadro de datos congelados

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

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

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

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

Tabla 9 - Mensaje de petición del Modo 02

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una respuesta en el Modo 01	42
#2	Valor hexadecimal del PID al que se está respondiendo	XX
#3	Número de Cuadro	XX
#4	Dato A	XX
#5	Dato B	XX
#6	Dato C	XX
#7	Dato D	00

Tabla 10 - Mensaje de respuesta del Modo 02

2.4.3 Modo 03 – Obtención de los códigos de falla

Este modo permite extraer de la memoria de la ECU todos los Códigos de Diagnóstico de Error o DTCs (Data Trouble Code) almacenados. Los DTCs consisten en códigos de tres dígitos precedidos por un identificador alfanumérico. Cuando la ECU reconoce e identifica un problema, se almacena en la memoria un DTC que corresponde a ese fallo. Estos códigos tienen por objetivo ayudar al usuario a determinar la causa fundamental de un problema. Las tramas de petición y respuesta se muestran en las Tablas 11 y 12.

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una petición de DTC en el Modo 03	03

Tabla 11 - Mensaje de petición del Modo 03

Byte	Parámetro	Valor Hex.
#1	Indica que estamos realizando una respuesta en el Modo 03	43
#2	Número de DTCs almacenados	XX
#3	High Byte	XX
#4	Low Byte	XX

Tabla 12 - Mensaje de respuesta del Modo 03

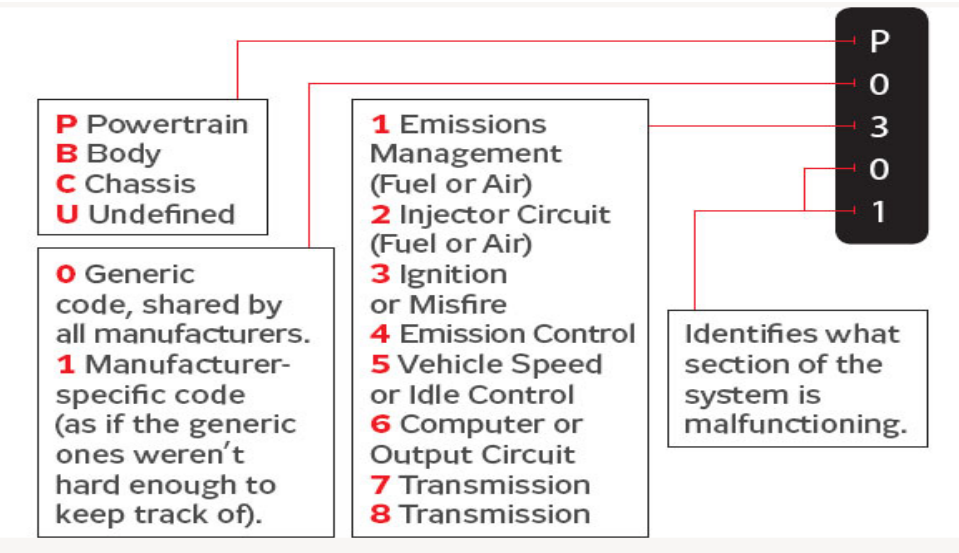


Fig. 25 – Estándar DTC

Estos cinco caracteres se traducen en dos bytes: HB (High Byte) y LB (Low Byte). El primer byte recoge los tres primeros caracteres y el segundo los dos últimos, tal como vemos en la Tabla 13.

Carácter	1	2	3	4	5
Bits	HB7-HB6	HB5-HB4	HB3-HB0	LB7-LB4	LB3-LB0
Código	00 = P 01 = C 10 = B 11 = U	00 = 0 01 = 1 10 = 2 11 = 3	xxxx su valor en Hexadecimal	xxxx su valor en Hexadecimal	xxxx su valor en Hexadecimal

Tabla 13 – Traducción DTC

Por tanto, poniendo el ejemplo de la Figura 25, tendríamos un fallo del sistema de propulsión (P) que se debe al incorrecto encendido de uno de los cilindros del motor (en este caso el 1) y su traducción sería:

HB = 00000011 (P = 00; 0 = 00; 3 = 0011)

LB = 00000001 (0 = 0000; 1 = 0001)

2.4.4 Modo 04 – Borrado de códigos de falla y valores almacenados

Con este modo se pueden borrar todos los códigos almacenados en el Módulo de Control del Motor y Transmisión (PCM, Power Train Control Module), incluyendo los códigos de falla y el cuadro de datos grabados temporalmente. Cabe destacar que si el problema no se corrigió, el o los códigos de falla que fueron borrados volverán a aparecer. Esta opción también apaga la lámpara que indica los códigos de falla en el tablero (MIL).

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

Tabla 14 - Mensaje de petición del Modo 04

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

Tabla 15 - Mensaje de respuesta correcta del Modo 04

Byte	Parámetro	Valor Hex.
#1	Indica que ha habido fallo al borrar	7F
#2	Modo	04
#3	Condiciones incorrectas	22

Tabla 16 - Mensaje de respuesta incorrecta del Modo 04

2.4.5 Modo 05 – Resultado de las pruebas de los transductores de oxígeno

Este modo devuelve los resultados de las pruebas realizadas a los sensores de oxígeno para determinar el funcionamiento de los mismos y la eficiencia del convertidor catalítico, vital para el control de las emisiones del vehículo automotor y para un correcto funcionamiento del mismo. Puede haber más de un sensor de oxigeno según el vehículo.

El sensor devuelve una serie de voltajes producto de la mezcla aire-combustible y lo mismo se observa para el tiempo de encendido. Los valores devueltos son:

1. Umbral de voltaje en el sensor de oxigeno rico a pobre.
2. Umbral de voltaje en el sensor de oxigeno pobre a rico.
3. Voltaje bajo en el sensor para el cambio.
4. Voltaje alto en el sensor para el cambio.
5. Tiempo en el cambio de rico a pobre.
6. Tiempo en el cambio de pobre a rico.
7. Voltaje mínimo en el sensor en el momento de la prueba.
8. Voltaje máximo en el sensor en el momento de la prueba.
9. Tiempo entre la transición de los sensores.

Los cuales se ilustran en la gráfica de la Figura 26.

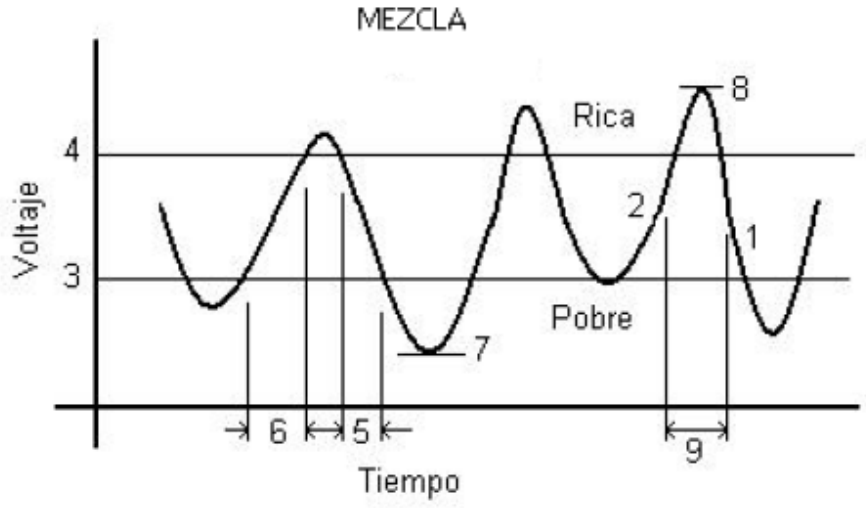


Fig. 26 – Gráfica del sensor de oxígeno

2.4.6 Modo 06 – Resultados de las pruebas de otros transductores

Este modo permite obtener los resultados de todas las pruebas de a bordo del sistema OBD2, resultado del sistema completo. Los monitores del sistema identifican que sistemas pueden ser monitoreados y en el caso de que se pueda, el estado de estos. Por ejemplo, los sensores de oxigeno o el catalizador normalmente se pueden observar, pero si el vehículo esta frío o apagado, marcará esta prueba como no preparada para ejecutarse.

Las pruebas más comunes que se efectúan en esta opción son las siguientes:

- Fallas de combustión (referentes a cables, bujías, encendido, inyectores, etc.)
- Sistema de combustible (incluye sensores de oxígeno, sensores de ciclo cerrado, retroalimentación de combustible y sistema de encendido)
- Catalizador.
- Calentamiento del catalizador.
- Sistema evaporativo.
- Sistema secundario de aire.
- Refrigerante del aire acondicionado.
- Sensores de Oxigeno.
- Calentamiento del sensor de oxígeno.
- Recirculación de gases de escape (EGR).

En el caso de tener fallas indicará cuantas se tienen, pero será necesario ir a la opción de códigos de falla para ver cuáles son, corregirlas y luego nuevamente realizar esta prueba. No todos los vehículos soportan este modo de reporte de fallas por lo que se recomienda usar la opción de fallas para esto. Las respuestas que arrojara son:

"N" Monitoreo no soportado

"C" Monitoreo completo

"I" Todavía no termina el monitoreo de esta prueba.

"?" Incongruencia del sistema.

2.4.7 Modo 07 – Muestra de códigos de falla pendientes

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

2.4.8 Modo 08 – Control de funcionamiento de componentes

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

2.4.9 Modo 09 – Información del automóvil

Este modo es opcional y no todos los vehículos están equipados con el mismo, básicamente este modo pide información sobre el vehículo automotor como número de serie y posiblemente información extra sobre el mismo.

2.5. Simuladores existentes

En el mercado existen numerosos simuladores como el que vamos a desarrollar, una de las marcas más populares es Özen Elektronik [17], que dispone de simuladores tanto para un protocolo en específico (como es nuestro caso), como para todos los protocolos actuales.

El simulador que más se ajusta a nuestros objetivos es el ECUSim 1010 (Fig. 27), que soporta el protocolo ISO 9141-2, dispone de una serie de potenciómetros que simulan diferentes PIDs del vehículo, un botón con el que se añaden DTCs, el indicador MIL y los conectores para el tester OBD (J1962) y la fuente DC de 12V. Su precio medio es de 120€.

Fig. 27 – Simulador ECUSim 1010

Un ejemplo de simulador que soporta todos los protocolos es el ECUSim 5100 (Fig. 28), que además de los potenciómetros para simular algunos PIDs del vehículo y el botón de fallo, también dispone de la opción de ser modificado por software para así tener acceso a más opciones de simulación. Su precio medio en el mercado es de 1.500€.

Fig. 28 – Simulador ECUSim 5100

3 DESARROLLO HARDWARE

A lo largo de este capítulo se explicarán los diferentes dispositivos físicos que se usan en el proyecto, comenzando por la elección de la plataforma Arduino, sus diferentes modelos y la elección del Arduino Mega.

Se explica el dispositivo de diagnóstico (tester) que se va a utilizar, eligiéndose el ELM327 Mini mediante interfaz bluetooth con un Smartphone.

Por último, se explica el desarrollo del simulador explicando las decisiones tomadas a lo largo del mismo, tanto la parte de interfaz OBD como la de simulación de sensores del vehículo. Finalmente, se explican los pasos de fabricación de la tarjeta PCB que será posteriormente programada por el Arduino Mega.

3.1 Arduino

Arduino es una plataforma de prototipos electrónica de código abierto (open – source) basada en hardware y software flexibles y fáciles de usar. Está pensado e inspirado en artistas, diseñadores, aficionados y para cualquier interesado en crear objetos o entornos interactivos.

Arduino consta de una placa principal de componentes eléctricos, donde se encuentran conectados los controladores principales que gestionan los demás complementos y circuitos ensamblados en la misma. Además, requiere de un lenguaje de programación para poder ser programado, configurado y utilizado a nuestra necesidad, por lo que se puede decir que Arduino es una herramienta "completa" en cuanto a las herramientas principales nos referimos, ya que sólo debemos instalar y configurar los componentes eléctricos que queramos para realizar el proyecto que tenemos en mente, haciéndola una herramienta no sólo de creación, sino también de aprendizaje en el ámbito del diseño de sistemas electrónicos-automáticos y, además, fácil de utilizar.

La placa Arduino también simplifica el proceso de trabajo con microcontroladores, ya que está fabricada de tal manera que viene “pre-ensamblada” y lista con los controladores necesarios para poder operar con ella una vez que la saquemos de su caja, ofreciendo una ventaja muy grande para profesores, estudiantes y aficionados interesados en el desarrollo de tecnologías. Las posibilidades de realizar proyectos basados en esta plataforma tienen como limite la imaginación de quien opera esta herramienta.

Existen numerosas plataformas hardware de código abierto, como pueden ser Raspberry Pi, BeagleBone, Sharks Cove, Minnowboard MAX, Nanode, Waspmote o LittleBits, pero se ha elegido la plataforma Arduino debido a sus apreciables ventajas:

- Open Source
- Fácil de programar
- Documentación y tutoriales en exceso
- Numerosas Librerías
- Diferentes placas
- Shields y periféricos

- Precio
- Infinidad de aplicaciones

3.1.1 Modelos de Arduino

Actualmente existen en el mercado una gran variedad de modelos según la aplicación que se vaya a implementar o los recursos que se necesiten para la misma. A continuación, se estudian los diferentes tipos de placa y sus características principales.

- **UNO**

Tiene 14 Entradas/Salidas digitales (6 de las cuáles pueden ser usadas como salidas PWM), 6 entradas analógicas, velocidad de reloj de 16MHz, conexión USB, jack de alimentación y un botón de reset. Contiene todos los componentes necesarios para que el microcontrolador funcione correctamente; Simplemente conectándolo a una computadora con un cable USB o alimentandolo con el adaptador AC-DC o una bateria para comenzar. El Arduino Uno es el modelo de referencia para la plataforma Arduino y es compatible con la gran mayoría de los shields existentes.

Fig. 29 – Arduino UNO

- **PRO**

El Pro viene tanto en 3.3V/8MHz y 5V/16MHz. Tiene la misma cantidad de pines y periféricos que el Arduino UNO ya que está basado en el mismo microcontrolador (ATmega328) pero carece de convertidor Serial-USB por lo que requiere de un cable FTDI para programarse, lo que lo hace una opción más barata que el Arduino UNO si piensas utilizar varias tarjetas a la vez (Un solo cable FTDI podrá programar todas las tarjetas). Está diseñado para instalaciones semi-permanentes en objetos o exhibiciones. La tarjeta viene sin headers pre-soldados para permitir el uso de varios tipos de conectores o directamente cables/alambres soldados.

Fig. 30 – Arduino PRO

- **PRO Mini**

Esta tarjeta es la versión miniatura del Arduino Pro, pensado para aplicaciones en las que el espacio sea muy reducido. Requiere un cable FTDI para programarse y puede encajar en un protoboard si se le soldan pines de espaciamiento de 0.1in.

Fig. 31 – Arduino PRO Mini

- **Leonardo**

Esta tarjeta tiene una forma muy similar a la tarjeta Uno, asi que es compatible con todos los Shields (a excepción de algunos por incompatibilidad de ciertos pines). La gran ventaja de esta tarjeta es que está basada en el procesador ATmega32u4, el cual tiene comunicación USB integrada, por lo que no requiere de un convertidor Serial-USB ni de un cable FTDI para programarse, además de ser un poco más económico que el Arduino Uno por requerir menos componentes. Gracias a sus capacidades USB puede emular las funciones de un mouse y un teclado, permitiéndote dar clic, doble clic, scroll o escribir texto de una manera muy sencilla.

Fig. 32 – Arduino Leonardo

- **MEGA**

Esta tarjeta tiene poder y a su vez compatibilidad con shields y código para Arduino Uno. Cuenta con una cantidad mucho mayor de I/O que el Arduino Uno (54 vs 14 del Uno), además de tener 14 salidas PWM, 4 puertos UART, I2C y 16 entradas analógicas. Además, tiene una memoria de mayor capacidad lo que te permitirá utilizarlo para códigos muy extensos o que requieran de una gran cantidad de variables.

Fig. 33 – Arduino MEGA

- **MEGA ADK**

Esta tarjeta es compatible pin con pin con el Arduino Mega, con la gran ventaja de incluir una interfaz Host USB, que te permite conectar tu Arduino a un teléfono con SO Android y comunicarte con él para acceder a la información de sus sensores o recibir órdenes de tu celular para controlar motores, LEDs, etc. Con ésta tarjeta puedes incluso recargar tu teléfono celular si requiere menos de 750mA y tienes una fuente de suficiente capacidad de corriente.

Fig. 34 – Arduino MEGA ADK

- **Due**

Esta tarjeta está basada en un procesador ARM-cortex de 32-bits a 84MHz, es la tarjeta Arduino con mayor capacidad y velocidad de procesamiento de ésta lista. Tiene un footprint similar al del Arduino Mega, pero tiene periféricos adicionales, como 2 convertidores DA y 2 pines de comunicación CAN. A diferencia de otras tarjetas Arduino, esta tarjeta trabaja a 3.3V por lo que un voltaje mayor a 3.3v en sus pines de I/O puede dañar la tarjeta.

Fig. 35 – Arduino Due

- **Ethernet**

Ésta tarjeta está basada en el microcontrolador ATmega328 (al igual que el Arduino Uno), pero cuenta además con capacidad de conectarse a una red via su puerto Ethernet. Cuenta además con un slot para tarjetas uSD por lo que podrás guardar una gran cantidad de información y utilizarla cuando lo requieras, aún después de haber reseteado la tarjeta. Nota: Se requieren de los pines 10 – 13 para la comunicación Ethernet, por lo que no deberán usarse para otros propósitos y el número de pines digitales disponibles se reduce a 9, contando con 4 salidas PWM.

Fig. 36 – Arduino Ethernet

- **LilyPad**

La tarjeta Lilypad está diseñada para ser utilizada en la ropa para crear e-textiles. Tiene pines especiales para coser sensores y actuadores utilizando un hilo conductivo.

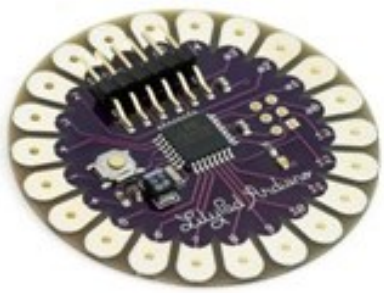


Fig. 37 – Arduino LilyPad

- **Esplora**

El Arduino Esplora está basado en el Leonardo y difiere de las anteriores tarjetas en que cuenta con un número de sensores integrados (de luz, temperatura, acelerómetro, joystick etc.) listos para usarse para interacción. Está diseñado para gente que quiere tomarlo y comenzar a programar su Arduino sin tener que aprender electrónica previamente.

Fig. 38 – Arduino Esplora

3.1.2 Arduino MEGA

Debido a su alto número de pines digitales y analógicos y los necesarios puertos UART (nos hacen falta dos), se ha elegido el Arduino MEGA como cerebro del simulador. Además, dispone de una memoria de gran capacidad, que soporta un elevado número de variables al que podría llegar el código en un posible uso futuro.

Las especificaciones técnicas generales de esta tarjeta se encuentran en la siguiente tabla 17 y la funcionalidad de cada uno de sus pines en la Figura 39.

Microcontrolador	ATmega1280
Tensión de operación	5V
Tensión de entrada (recomendada)	7-12V
Tensión de entrada (límite)	6-20V
Pines digitales I/O	54 (15 de ellos con salidas PWM)
Pines de entrada analógica	16
Corriente DC por Pin I/O	40 mA
Corriente DC para pines de 3.3V	50 mA
Memoria Flash	128 KB (4 KB usados para el arranque)
SRAM	8 KB
EEPROM	4 KB
Velocidad del reloj	16 MHz

Tabla 17 – Especificaciones técnicas Arduino Mega

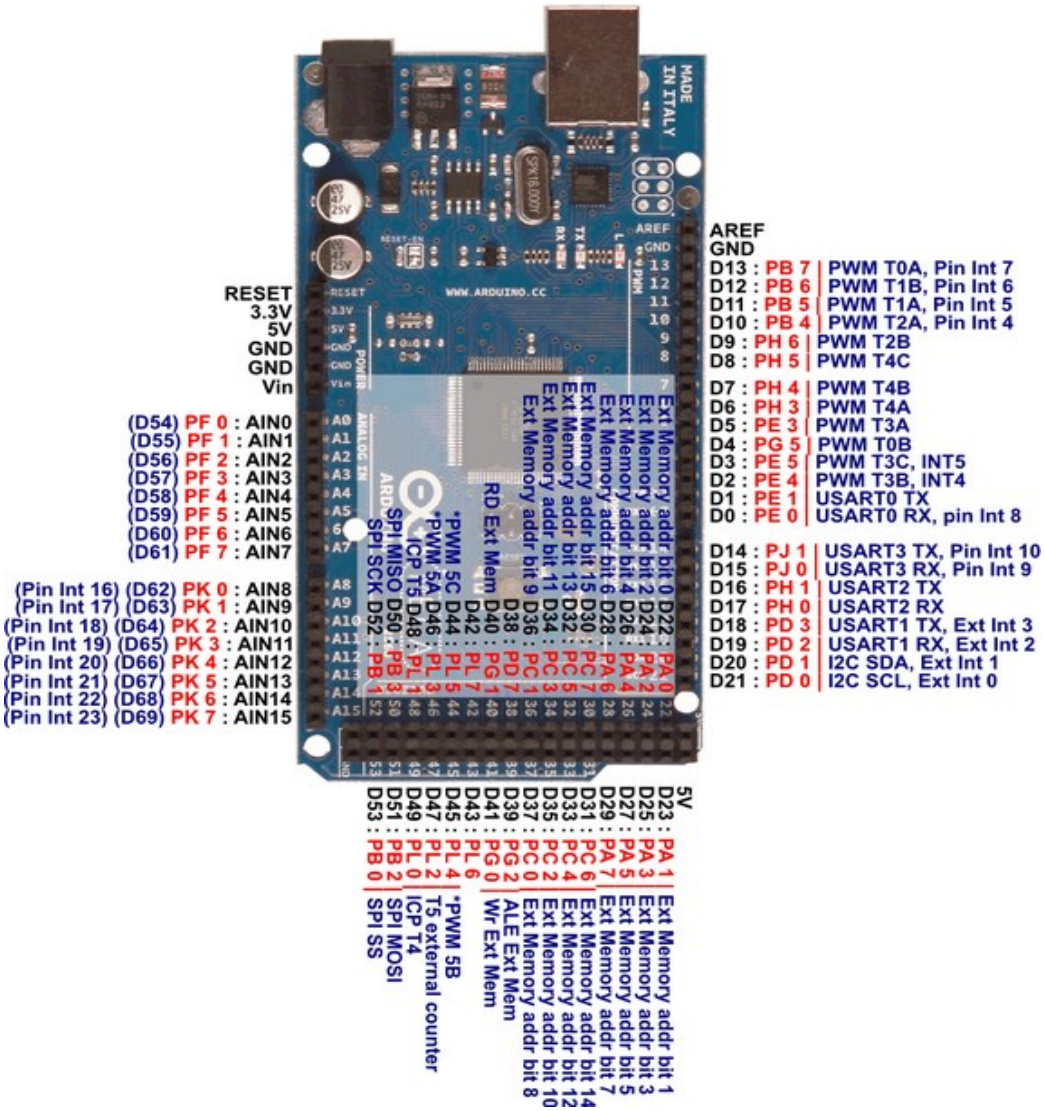


Fig. 39 – Pinout Arduino Mega 2560

Los pines que usaremos para desarrollar el simulador serán:

Pin	Función
A0	Potenciómetro que simula la posición del acelerador (%)
A1	Potenciómetro que simula la velocidad
A2	Potenciómetro que simula las revoluciones por minuto
A3	Potenciómetro que simula el nivel de combustible (%)
A4	Potenciómetro que simula la temperatura ambiente
A5	Potenciómetro que simula la temperatura del refrigerante
52	LED que simula el Ready to Comunnicate (inicialización completa)
50	LED que simula el indicador MIL del vehículo
38	Interruptor 1 (activa el modo manual)
36	Interruptor 2 (activa el DTC 1 = P0301)
34	Interruptor 3 (activa el DTC 2 = P0340)
32	Interruptor 4 (activa el DTC 3 = P0217) – Activa indicador MIL
30	Interruptor 5 (activa el DTC 4 = P0171)
28	Interruptor 6 (activa el DTC 5 = P0500)
26	Interruptor 7 (activa el DTC 6 = P0420) – Activa indicador MIL
24	Interruptor 8 (activa el DTC 7 = P0068) – Activa indicador MIL
15	Receptor del puerto serie 3 (protocolo ISO 9141-2) – También se usará como pin digital en la inicialización a 5 bps
14	Transmisor del puerto serie 3 (protocolo ISO 9141-2)
5V	Alimentación para inversor 7404N, potenciómetros y resistencias pull-up de interruptores
GND	Tierra del circuito

Tabla 18 – Pines de Arduino MEGA usados para el simulador

3.2 Dispositivos de diagnostico y visualización de resultados

Como ya se ha comentado en los aspectos teóricos, se necesita un “traductor” de OBD al estándar RS-232 para la visualización en PCs, PDAs o Smartphones. Este dispositivo que hace de puente entre los dos sistemas de comunicación será el ELM327 (véase apartado 2.2.7.). Las interfaces de salida que puede tener este dispositivo son la interfaz USB (requiere conexión por cable al PC), interfaz WiFi (estándar IEEE 802.11 a 2,4 o 5GHz) o interfaz Bluetooth (banda ISM a 2,4GHz) que será la usada en este proyecto debido a su mayor comodidad.

Fig. 40 – Dispositivo de diagnóstico ELM327 Mini

Los pines del tester que debemos conectar a nuestro simulador serán (simétricos al conector OBD del vehículo):

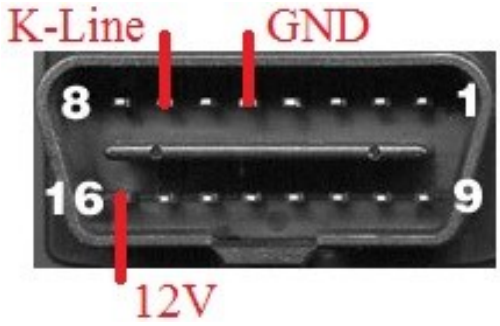


Fig. 41 – Pines del tester usados para el protocolo ISO 9141-2

Como dispositivo de visualización de resultados se utilizará cualquier smartphone con S.O Android para que soporte algún software compatible con nuestro dispositivo de diagnóstico (véase más adelante).

3.3 Simulador

3.3.1 Interfaz con Arduino

En primer lugar, debemos realizar un circuito que implementa la bidireccionalidad entre la línea K del protocolo ISO 9141-2 y los pines Tx y Rx de la UART de nuestro Arduino. Para ello, nos fijamos en el circuito inverso implementado dentro del dispositivo de diagnóstico, desde el pin 7 hasta los pines 21 y 12 del micro ELM327 que realiza una operación similar a la deseada:

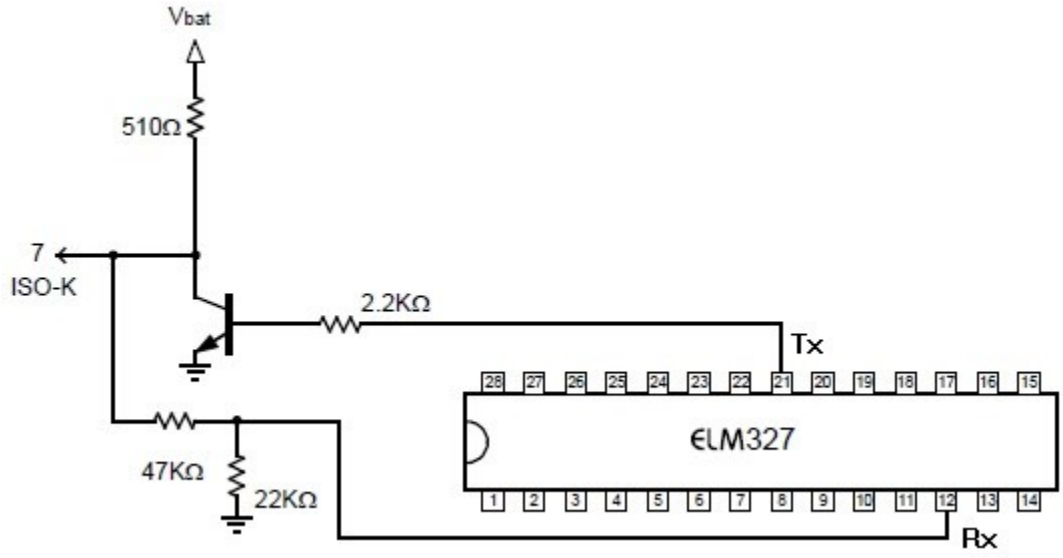


Fig. 42 – Interfaz ISO 9141-2 del dispositivo de diagnóstico

La línea ISO-K está a nivel alto en estado de reposo, por lo que Rx estará también a nivel alto (con la tensión correspondiente al resultado del divisor de tensiones) y cuando ISO-K se ponga a cero, Rx también lo hará.

En el lado contrario, Tx está a cero en estado de reposo y por tanto el transistor BJT (que trabaja en conmutación) se encuentra en corte (como un interruptor abierto) ya que no existe corriente en la base y es como si no estuviera la línea Tx. Cuando Tx se pone a nivel alto, el BJT se encuentra en saturación (como un interruptor cerrado) y pone a cero la línea ISO-K (ya que el emisor está conectado a tierra) y así conseguimos la independencia de las líneas Tx y Rx así como la bidireccionalidad por la línea ISO-K.

En nuestro caso, siendo Vbat de 12V, cambiaremos la resistencia de 22kΩ por una de 33kΩ, para que caigan aproximadamente 5V en nuestro pin Rx del Arduino (tensión de operación).

$$Vrx = 12 \cdot \frac{33k}{47k + 510 + 33k} = 4{,}92V$$

Por otro lado, añadiremos un inversor a la salida del pin Tx de nuestro Arduino ya que en nuestro caso, éste pin se encuentra a nivel alto en estado de reposo y por tanto cuando impongamos un cero, se invertirá a uno para que el BJT sature y la línea ISO-K se ponga a cero.

Es recomendable asegurarnos del comportamiento del transistor mediante un análisis transitorio observando cómo se comportarían los puntos Tx y Rx del Arduino con el circuito en cuestión. Si simulamos el transmisor mediante una fuente variable de 0 a 5V (en este caso no hace falta el inversor para asegurar el correcto funcionamiento) y con una fuente V1 de 12V:

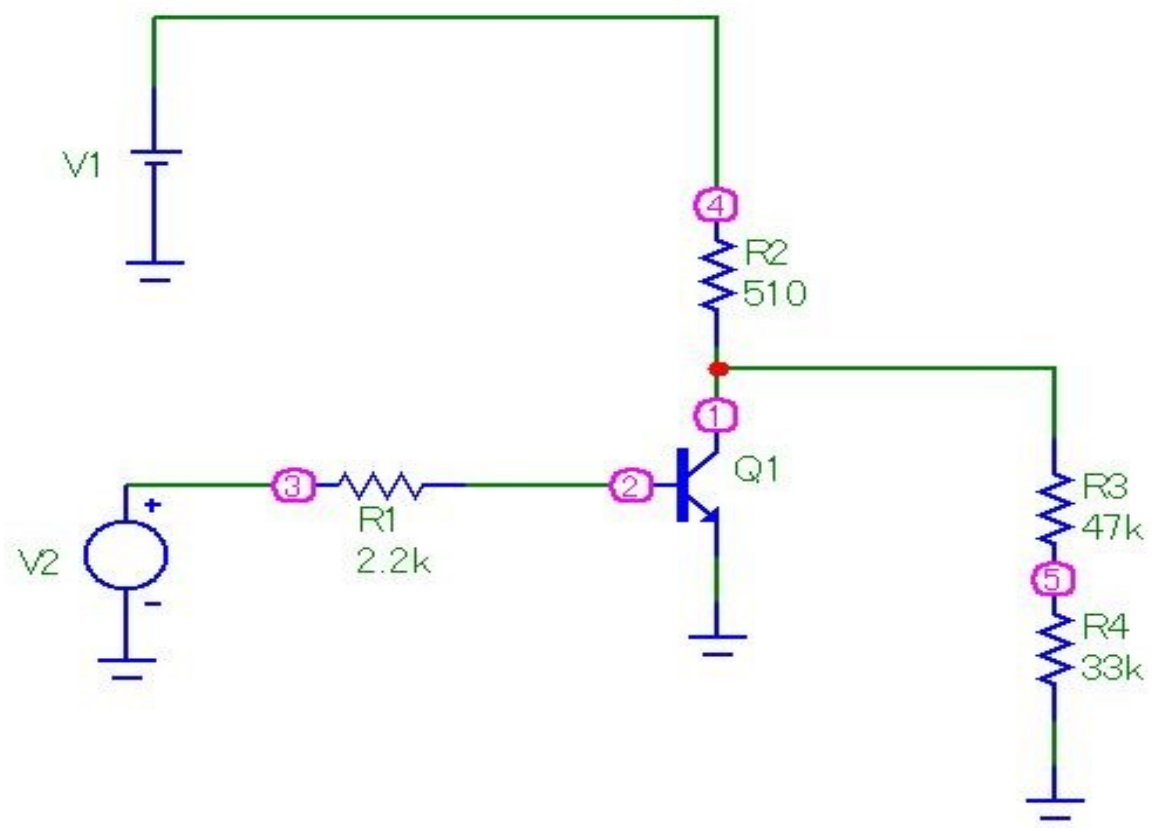


Fig. 43 – Circuito para la simulación de la línea ISO-K en Micro-Cap

Y realizando un análisis transitorio en los nodos 3(Tx) y 5(Rx) con la fuente V2 conmutando de 0 a 5V cada 0,5ms, obtenemos la siguiente gráfica:

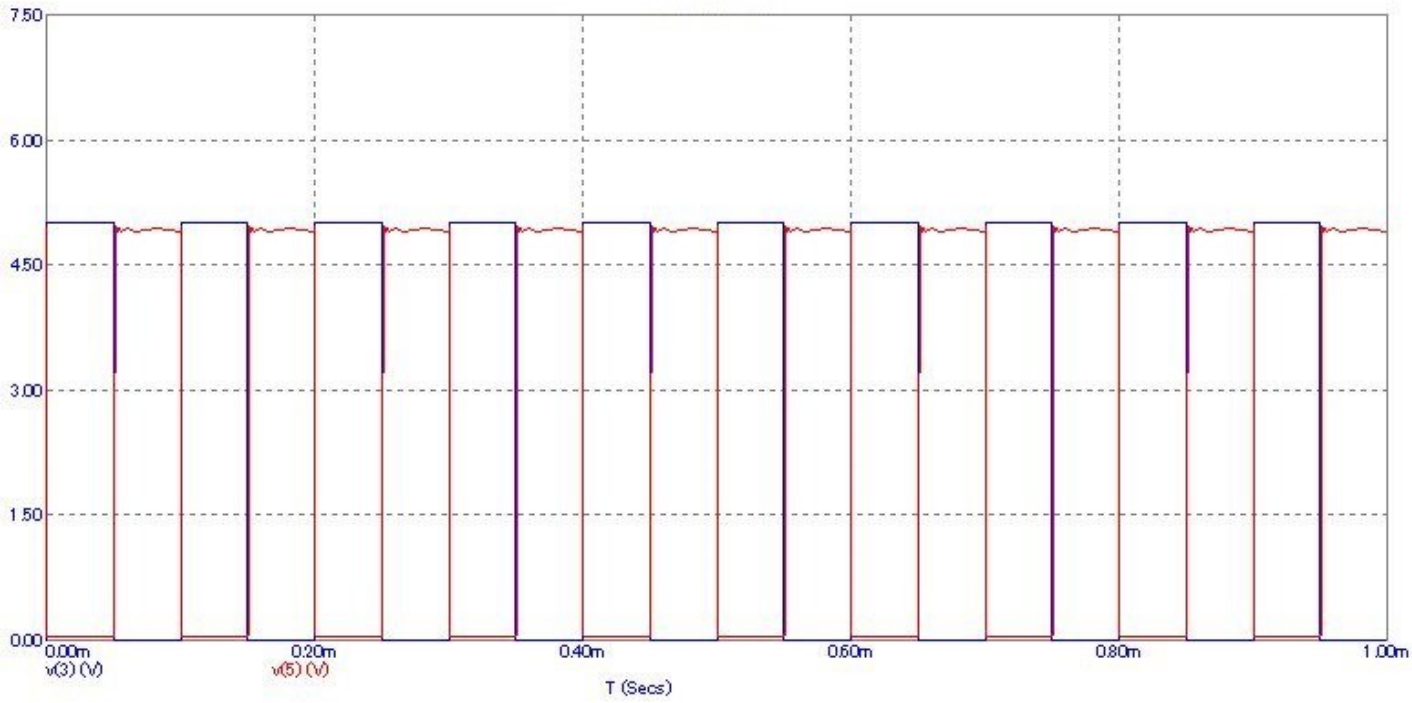


Fig. 44 – Resultado de análisis transitorio de la línea ISO-K en Micro-Cap

Se observa que efectivamente cuando el Arduino transmite, el BJT satura y pone un cero en el receptor serie. De manera análoga, cuando el transmisor está en reposo, el BJT se encuentra en corte y en el receptor caen los 5V deseados.

A la hora de integrar los componentes que componen este circuito, se deben tener en cuenta algunos detalles:

- En el inversor 7404N se usarán los pines 1 y 2 (entrada y salida del inversor 1), el pin 7 (tierra) y el pin 14 (alimentación de 5V).

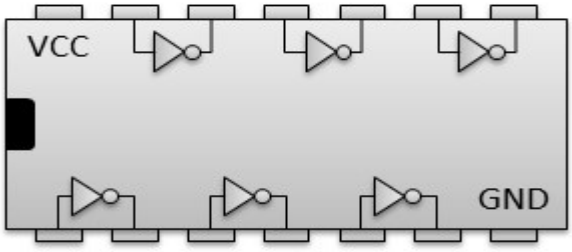


Fig. 45 – Pinout inversor 7404N

- La resistencia de 510Ω, al no ser un valor comercial, la implementaremos con dos resistencias en serie de 330Ω y 180Ω y por seguridad, se requiere que sean de $1/2\,W$ ya que la potencia exigida por ellas es de:

$$P = \frac{V^2}{R} = \frac{144V^2}{510\Omega} = 0{,}28W$$

3.3.2 Simulador del comportamiento del vehículo

Los dispositivos usados para el comportamiento serán:

- 6 potenciómetros que simularán señales analógicas como la posición del acelerador o la velocidad.
- 8 interruptores que simularán errores DTC del vehículo así como la activación del modo manual (véase más adelante).
- 3 diodos LEDs que simularán el indicador MIL, inicialización correcta y alimentación.

Para integrar estos componentes en el circuito, se deben tener en cuenta algunos detalles:

- Los potenciómetros serán de 10KΩ, valor común con buen rango de valores.
- Los interruptores llevarán resistencias pull-up de 10kΩ, por lo que el pin digital estará a nivel alto con el interruptor abierto (pestaña del interruptor hacia abajo) y a nivel bajo con el interruptor cerrado (hacia arriba).

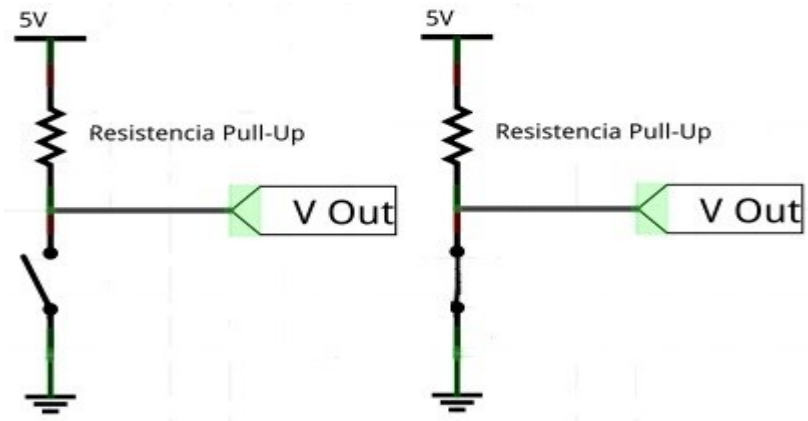


Fig. 46 – Resistencia Pull-up interruptores

- Los diodos LEDs llevarán una resistencia serie por su terminal positivo (ánodo) según la tensión y corriente que debe circular por sus terminales.

  - En el caso del LED verde (simulará la inicialización correcta o lo que es lo mismo, preparado para la comunicación), se conectará al pin digital del Arduino nº 52 que tendrá una tensión de 5V. Este LED funciona con una caída de tensión entre sus terminales de 3,2V y una corriente de 20mA. Como queremos que su vida se alargue en la medida de lo posible, intentaremos no sobrepasar los 15mA. Por tanto, su resistencia serie se calcula de la forma:

$$R = \frac{V}{I} = \frac{5V - 3{,}2V}{0{,}015A} = 120\Omega$$

Con una resistencia de 220 Ω nos servirá, que proporciona al LED algo menos de 10mA, suficiente para que se encienda. Su potencia, aunque ya sabemos que no exige mucha, será:

$$P = \frac{V^2}{R} = \frac{3{,}24V^2}{220\Omega} = 0{,}015W$$

- El LED rojo que simula el indicador MIL estará conectado al pin 50 del Arduino, y sus exigencias para conducir son unos 2V (entre 1,9 y 2,1V) y 0,015A. Su resistencia será:

$$R = \frac{V}{I} = \frac{5V - 2V}{0{,}015A} = 200\Omega$$

Usaremos también una resistencia serie de 220 Ω y su potencia será:

$$P = \frac{V^2}{R} = \frac{9V^2}{220\Omega} = 0{,}041W$$

- Por último, añadiremos un LED rojo que se encienda cuando se le conecte al circuito los 12V necesarios para la alimentación del tester, por lo que su resistencia será:

$$R = \frac{V}{I} = \frac{12V - 2V}{0{,}015A} = 666{,}67\Omega$$

Usaremos una resistencia serie de 680 Ω exigiéndose una potencia de:

$$P = \frac{V^2}{R} = \frac{100V^2}{680\Omega} = 0{,}15W$$

Con respecto a la potencia, podemos comprobar que con las resistencias de $1/4$ W nos vale de sobra.

3.3.3 Circuito en placa de pruebas

Recogiendo toda la información anterior, se procede a montar el circuito completo en una placa de pruebas (la parte de la interfaz del protocolo y la parte de simulación), para ello se deberá alimentar con 12V procedentes de una fuente de alimentación o de un transformador AC/DC conectado a la red eléctrica.

Por otro lado, las conexiones que van al dispositivo de diagnóstico se realizarán mediante cables con terminales de tipo cocodrilo, para garantizar su correcto agarre al mismo.

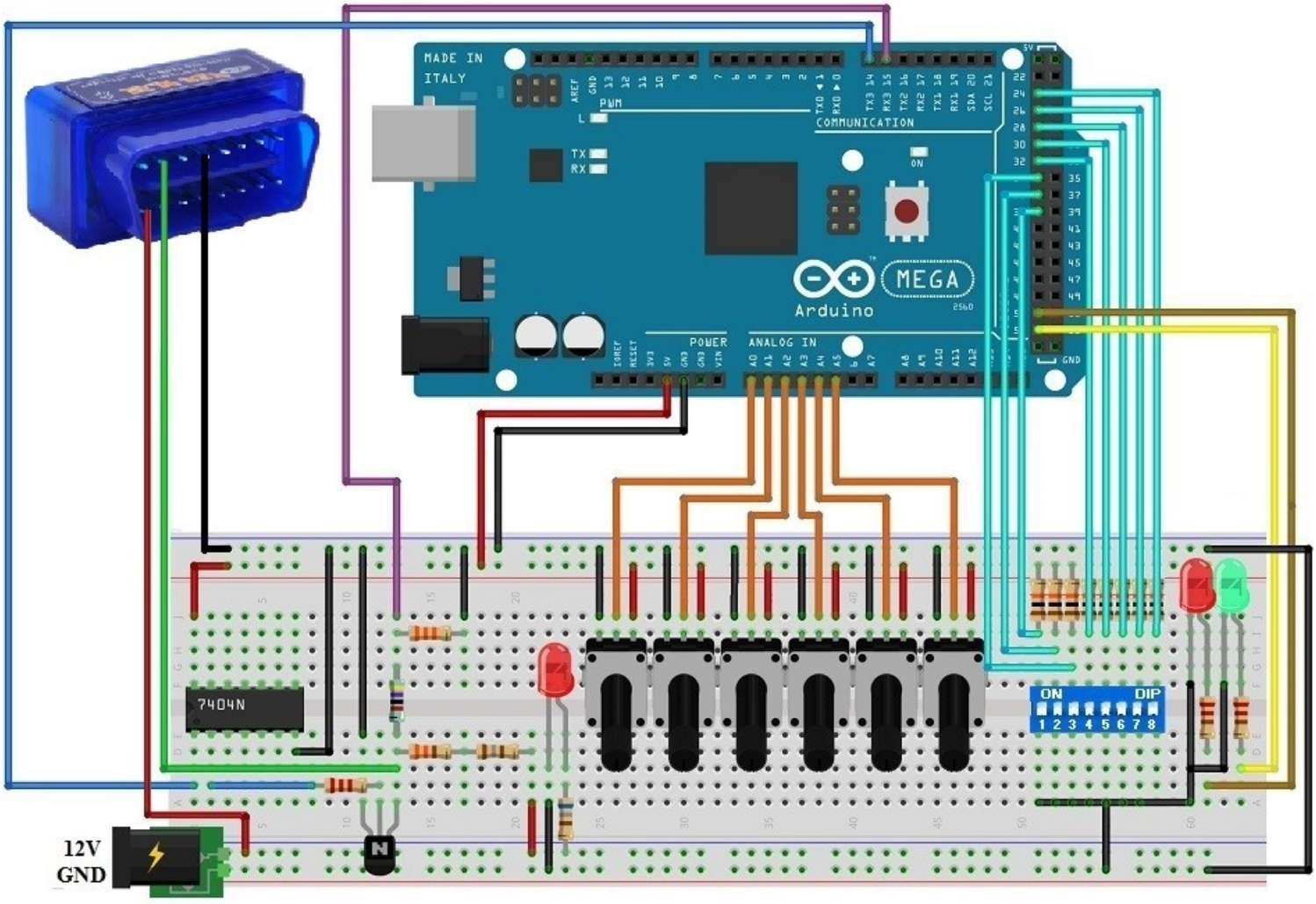


Fig. 47 – Montaje del circuito completo en placa de pruebas

Cabe destacar que para alimentar la placa Arduino, cargar el código en la misma, así como para la implementación del modo manual, hará falta además un cable USB de tipo A/B que conecte la placa con nuestro PC.

Fig. 48 – Cable USB tipo A/B

3.3.4 Circuito en PCB

Comprobado que funciona el circuito, procedemos a fabricar la placa PCB. Para ello, usaremos un software de diseño de PCB llamado Eagle, por tener versión gratuita y gran cantidad de bibliotecas de componentes.

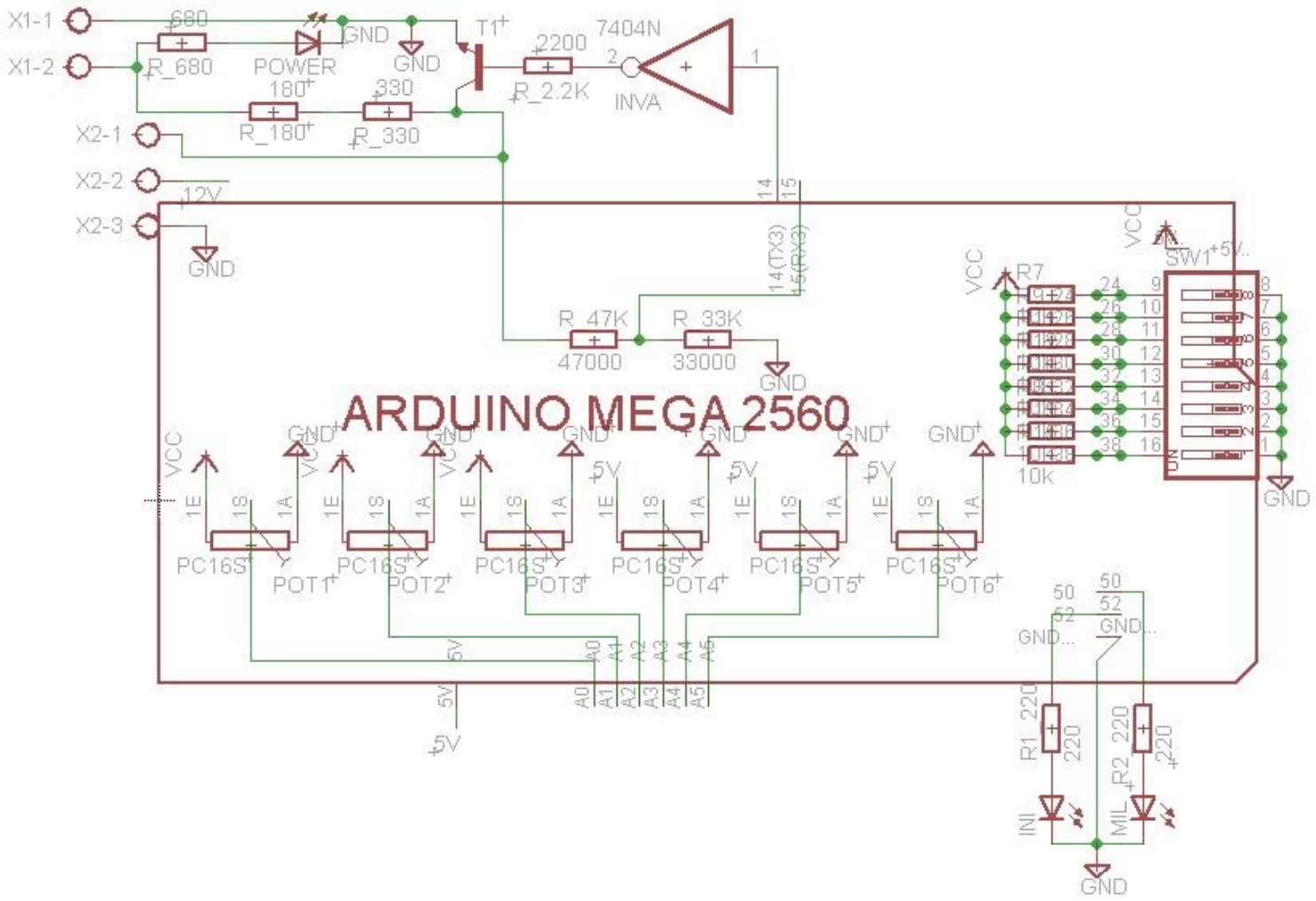


Fig. 49 – Esquemático del simulador en Eagle

A la hora de montar el esquemático, se ha descargado la huella del Arduino MEGA 2560 para conocer la ubicación de los pines a utilizar, para así poder insertar luego la tarjeta PCB en el Arduino.

Además, la alimentación se aplicará a través de una borna de 2 pines (12V, GND) y las conexiones hacia el dispositivo de diagnóstico a través de una borna de 3 pines (12V, GND, ISO-K).

A la hora de la ubicación de componentes, colocaremos el switch PID lo más cerca de los pines digitales del Arduino, las bornas lo más cerca de un extremo de la tarjeta y los potenciómetros de 3 en 3 a cada lado para optimizar el espaciado (tarjeta cuadrada) y para compensar el peso de la misma.

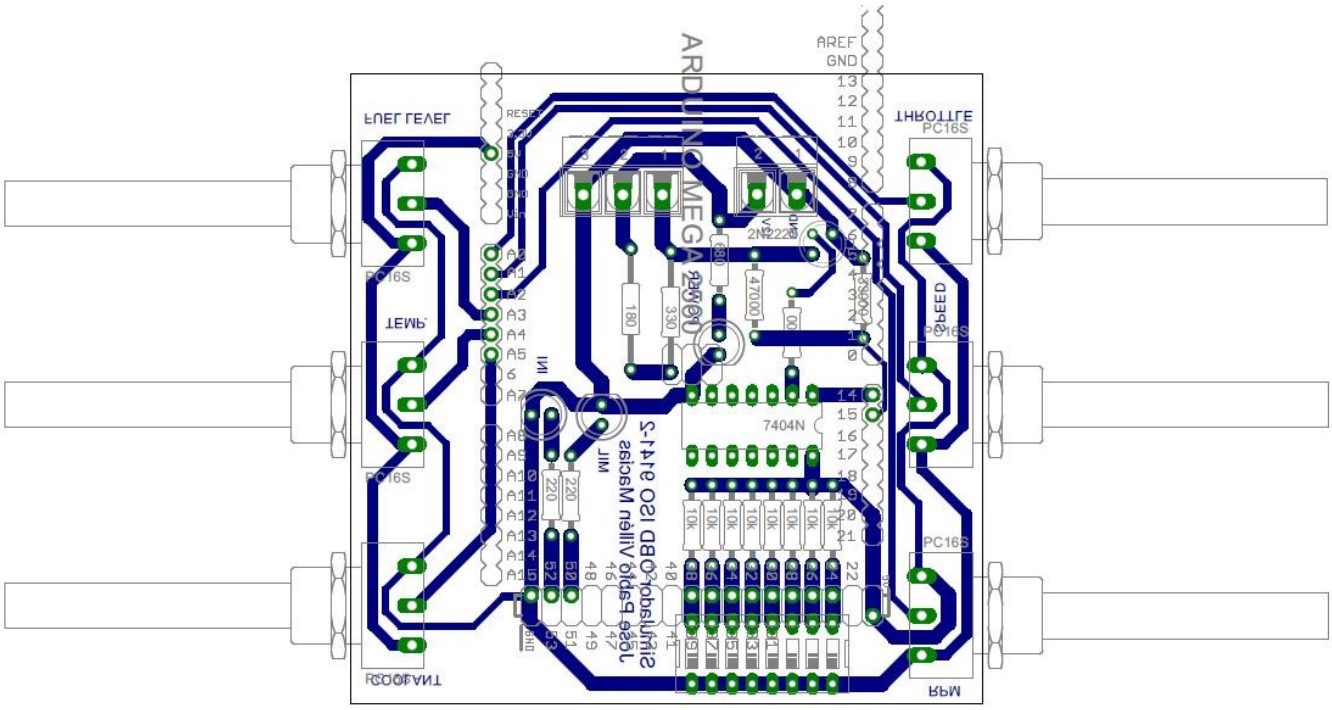


Fig. 50 – Colocación de componentes en Eagle

Una vez colocados e intentando tener cada pista lo más grande posible, dejamos únicamente los pads, las pistas, las dimensiones de la placa así como algún texto escrito en cobre para posible ayuda futura.

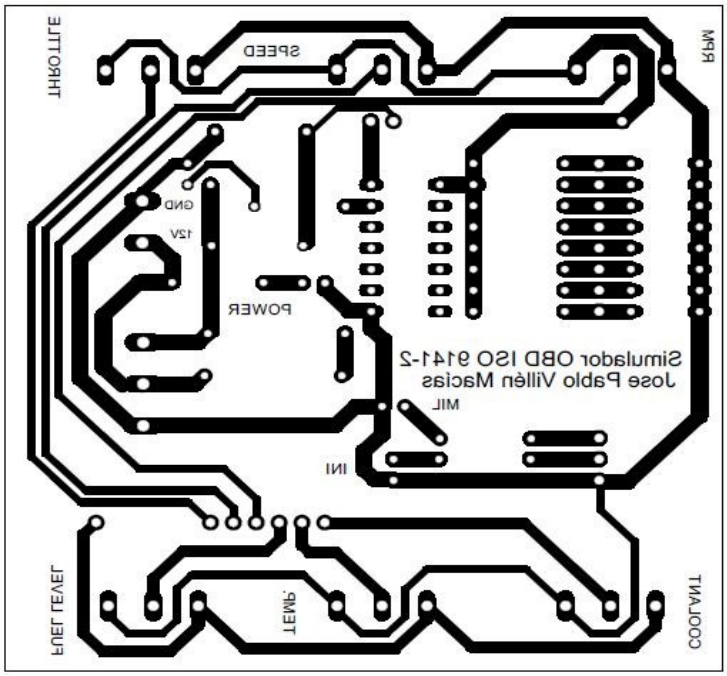


Fig. 51 – Impreso para PCB en Eagle

No se ha añadido plano de masa ya que el circuito no trabaja a alta frecuencia (no habrá problemas de ruido) y dificultaría el soldado de componentes.

Una vez impreso el circuito en papel vegetal con una impresora láser en buena calidad con toner negro, se expone el PCB virgen a luz ultravioleta en la insoladora (con el circuito sobre el lado del cobre del PCB). A continuación se revela (con líquido revelador) y se ataca mediante una mezcla de ácido: agua fuerte $(1/4)$, agua oxigenada $(1/4)$ y agua $(2/4)$. Tras ello, se frota con acetona para exponer completamente el cobre. Finalmente, perforamos los agujeros en los pads y soldamos los componentes.

Ya tenemos nuestro simulador OBD2 con protocolo ISO 9141-2 para ser programado en Arduino MEGA 2560 según los recursos HW que hemos implementado.

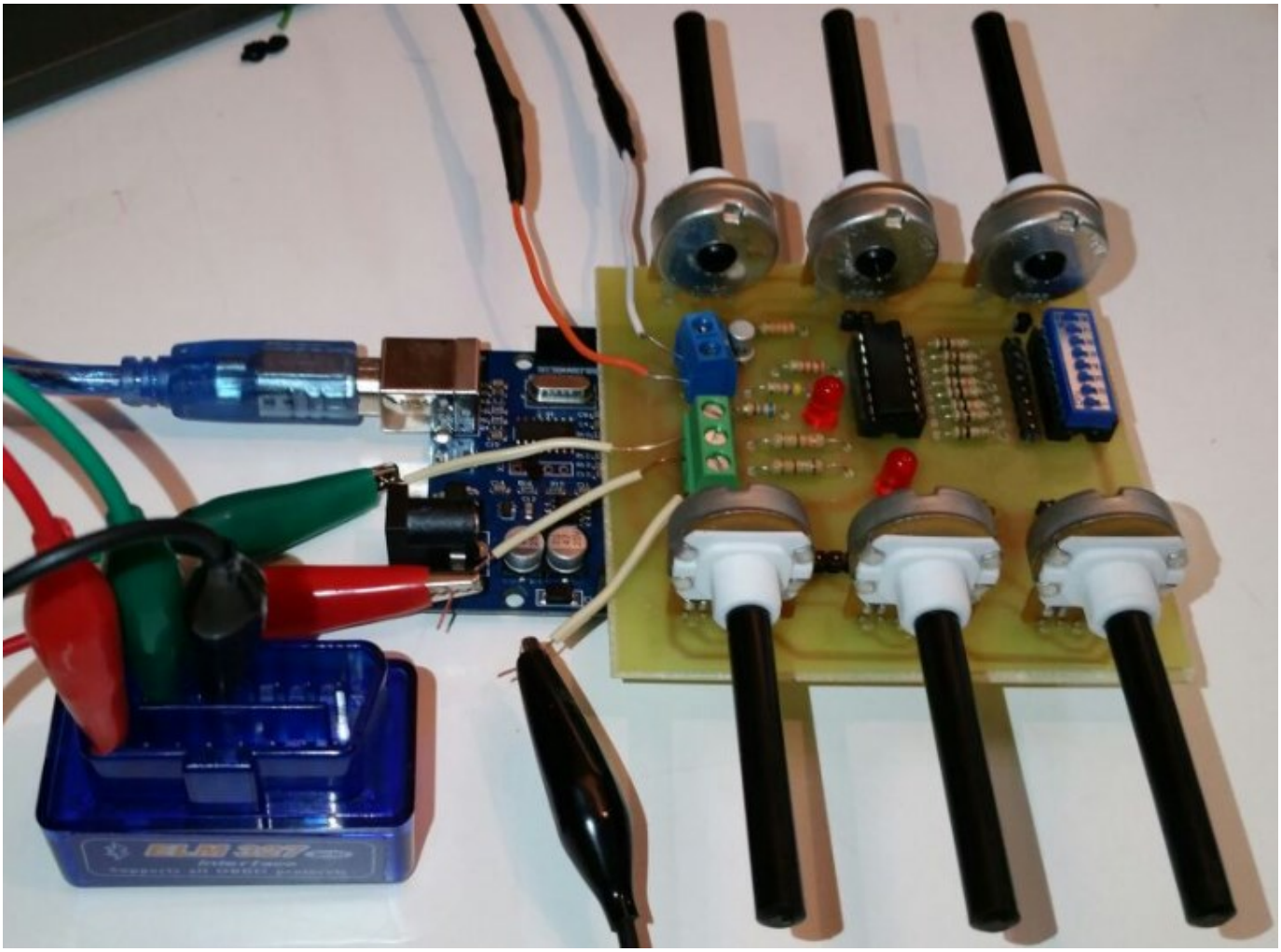


Fig. 52 – Simulador OBD2 con protocolo ISO 9141-2 para Arduino MEGA 2560

3.3.5 Montaje del sistema

Una vez fabricada la tarjeta PCB del simulador, se inserta en el Arduino MEGA por sus pines correspondientes (mencionados anteriormente) y se conecta con el resto del sistema tal como se indica en la Figura 53.

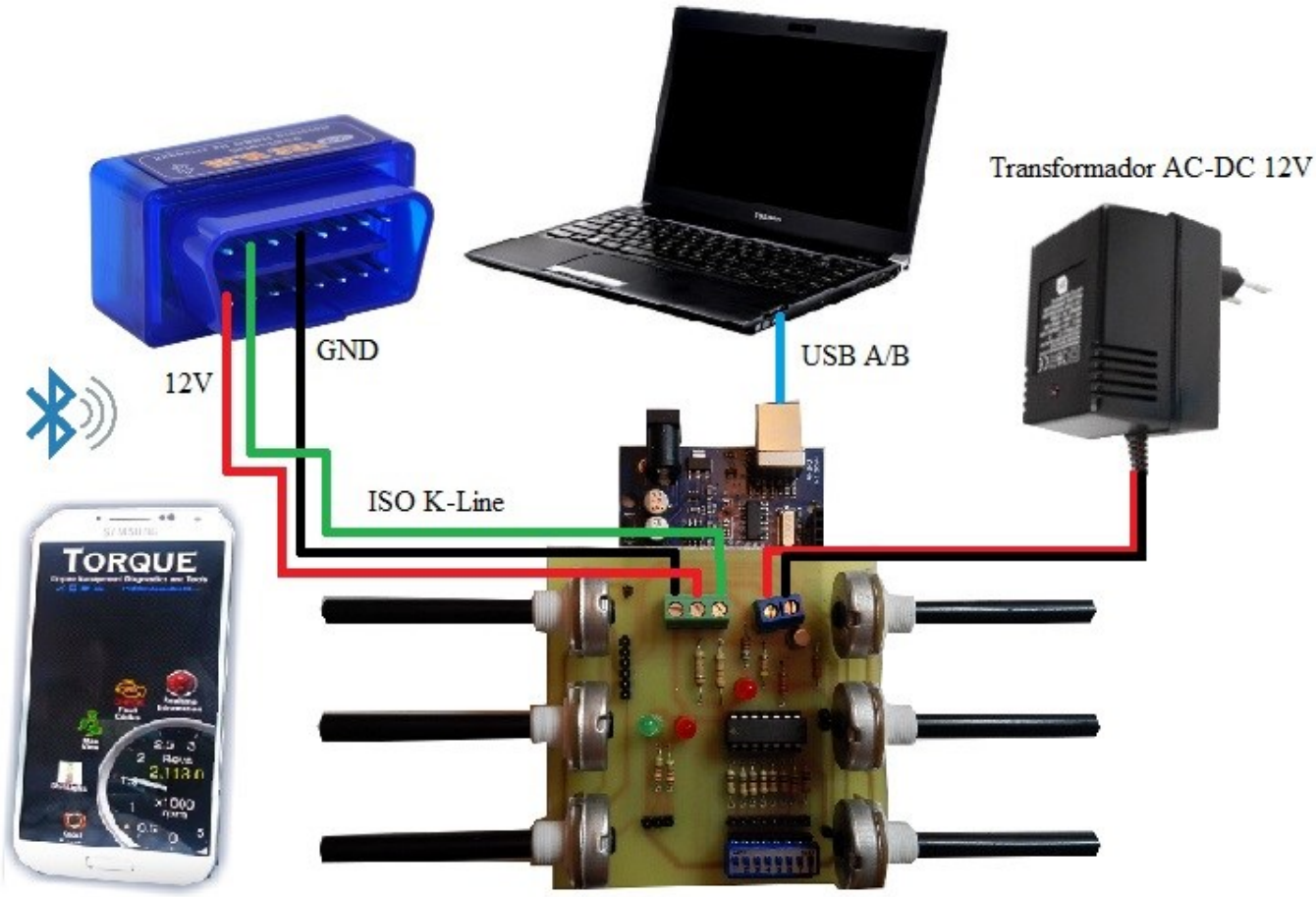


Fig. 53 – Esquema de utilización del sistema

4 DESARROLLO SOFTWARE

En el siguiente capítulo se explicará el código que simula el comportamiento de un vehículo con protocolo ISO 9141-2. Se explicará con anterioridad el entorno de programación Arduino, así como su configuración y funciones principales. Por último, veremos la interfaz del equipo de visualización (tester) que se tratará de la app para Smartphone Torque Pro.

4.1 Software Arduino

Dado que el Arduino es como un pequeño ordenador que ejecuta una serie de códigos que previamente le hemos introducido, necesitaremos un programa para poder meter estos códigos a la propia placa. Este programa se llama IDE, que significa "Integrated Development Environment" ("Entorno de Desarrollo Integrado"). Este IDE estará instalado en nuestro PC, es un entorno muy sencillo de usar y en él escribiremos el programa que queramos que el Arduino ejecute. Una vez escrito, lo cargaremos a través del USB y Arduino comenzará a trabajar de forma autónoma.

4.1.1 Entorno de programación y configuración

Al abrir el programa veremos cómo nos aparece la consola principal del IDE Arduino en la cual podemos ver las siguientes zonas:

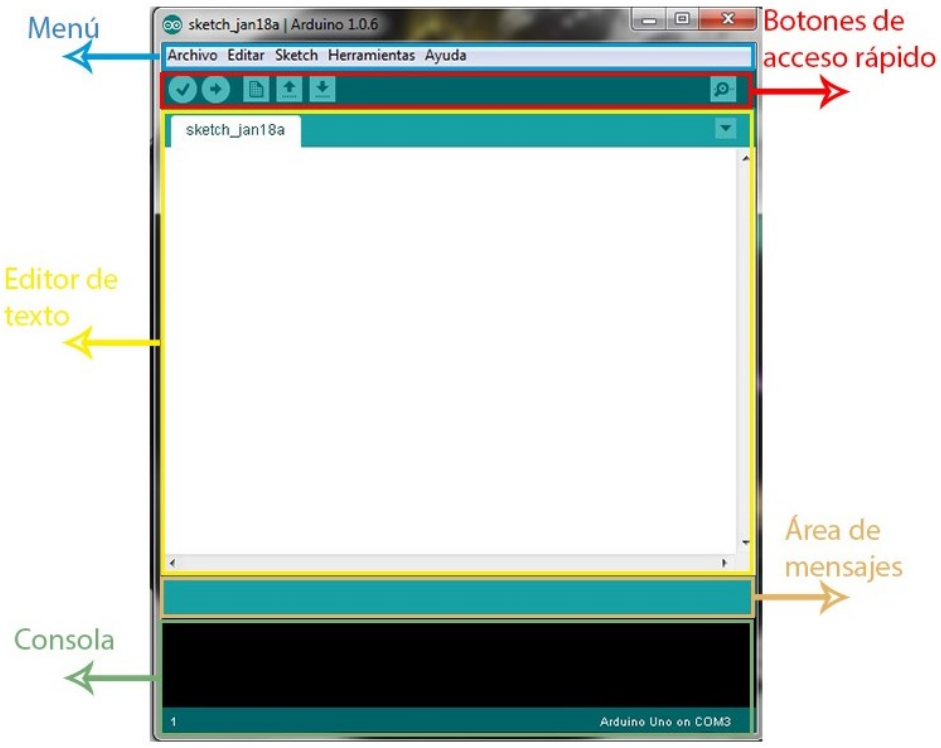


Fig. 54 – IDE Arduino

En la parte de menú tenemos una zona para acceder a funciones como carga de archivos, edición del texto del código, carga de librerías y ejemplos, configuración, herramientas, etc.

En los botones de acceso rápido tenemos los siguientes iconos:


Verifica si tu programa está bien escrito y puede funcionar.


Carga el programa a la placa de Arduino tras compilarlo.


Crea un programa nuevo.


Abre un programa.


Guarda el programa en el disco duro del ordenador.


Monitor Serial, abre una ventana de comunicación con la placa Arduino en la que podemos ver las respuestas que nuestro Arduino nos está dando, siempre que tengamos el USB conectado.

En el cuadro del editor de texto escribiremos el código del programa que queramos que Arduino ejecute.

Finalmente, el área de mensajes y la consola Arduino nos irán dando información sobre si la consola está compilando, cargando… y sobre los fallos o errores que se produzcan tanto en el código como en el propio IDE.

Con el fin de configurar nuestro IDE para que se comunique con nuestra placa Arduino, conectaremos nuestro Arduino mediante el cable USB al PC y después de que el sistema operativo haya reconocido e instalado la tarjeta automáticamente, nos dirigimos a la zona de menú, pulsamos en Herramientas y después en Tarjeta. Ahí seleccionamos el modelo de tarjeta Arduino que tengamos, en nuestro caso "Arduino Mega 2560". Después vamos a la opción Puerto Serial y elegimos el COM en el que tenemos conectado nuestro Arduino.

4.1.2 Estructura de un programa

La estructura básica del lenguaje de programación de Arduino es bastante simple y se compone de al menos dos partes. Estas dos partes o funciones necesarias, encierran bloques que contienen declaraciones o instrucciones. Se trata de las funciones setup() (parte encargada de recoger la configuración) y loop() (contiene el programa que se ejecutará cíclicamente). Ambas funciones son necesarias para que el programa trabaje.

La función de configuración (setup) debe contener la declaración de las variables. Es la primera función a ejecutar en el programa, se ejecuta sólo una vez, y se utiliza para configurar o inicializar pinMode (modo de trabajo de las E/S), configuración de la comunicación en serie y otras.

La función bucle (loop) contiene el código que se ejecutará continuamente (lectura de entradas, activación de salidas, etc) Esta función es el núcleo de todos los programas de Arduino y la que realiza la mayor parte del trabajo.

En un programa de Arduino también podemos encontrar funciones de usuario, escritas para realizar tareas repetitivas y para reducir el tamaño de un programa. Las funciones se declaran asociadas a un tipo de valor "type". Este valor será el que devolverá la función, por ejemplo 'int' se utilizará cuando la función devuelve un dato numérico de tipo entero. Si la función no devuelve ningún valor entonces se colocará delante la palabra "void", que significa "función vacía".

Además, existen las funciones predefinidas por Arduino, que facilitan la programación, permitiendo desde manejar las entradas y salidas de la placa, hasta permitir comunicaciones serie. En la siguiente tabla 18 se muestran las que usaremos en el código del proyecto.

Función	Descripción
pinMode(pin,mode)	Configura el pin para que se comporte como entrada o salida
digitalRead(pin)	Lee el valor (HIGH o LOW) de un pin digital específico
digitalWrite(pin,valor)	Escribe un valor HIGH o LOW a un pin digital específico
millis()	Devuelve el número de milisegundos desde que se arrancó el programa
delay(ms)	Pausa el programa la cantidad de milisegundos especificada
Serial.begin(baud rate)	Establece el ratio de bits por segundo para la transmisión de datos serie
Serial.write(data)	Escribe datos binarios hacia el Puerto serie
Serial.println(data)	Imprime datos por el Puerto serie en formato ASCII seguido del carácter retorno de carro y el carácter de nueva línea
Serial.available()	Obtiene el número de bytes (caracteres) disponibles para lectura por serie
Serial.flush()	Espera que se complete la transmisión de datos de salida por serie
Serial.read()	Lee datos entrantes por el Puerto serie
random(min,max)	Genera números seudo-aleatorios entre los valores mín y máx
analogRead(pin)	Lee el valor de un pin analógico específico

Tabla 19 – Funciones predefinidas usadas en el proyecto

Para más información del lenguaje de programación Arduino (estructuras de control, tipos de datos, variables, constantes o librerías) consultar la página web oficial de Arduino [18].

4.1.3 Código del proyecto (completo en Anexo A)

El código realizado que se cargará en la placa Arduino Mega simula la ECU de un vehículo con protocolo OBD ISO 9141-2 y es capaz de:

- Realizar la inicialización que exige el protocolo, tanto el mensaje 0x33 recibido a 5 bauds por segundo, como los posteriores mensajes intercambiados por la ECU y el tester, hasta que se establece la comunicación.
- Comprobar continuamente si recibe tramas del tester y si éstas son correctas, las estudia y responde convenientemente según el modo exigido.
- Comprobar continuamente el estado de los 8 interruptores. Si se activa el modo manual (interruptor 1), espera a recibir datos por el puerto serie con el PC y si éstos son correctos, tiene preferencia el modo manual ante el analógico y se responde al PID con el valor escrito por el PC. Los otros 7 interruptores activan por separado 7 DTCs del modo 3 (algunos de ellos encienden el indicador MIL).

- Responder al modo 1 según el PID requerido, con un total de 13 PIDs con 6 de ellos controlados analógicamente mediante potenciómetros.
- Responder al modo 3 con los DTCs que tenga almacenados (dependiente de los interruptores).
- Responder al modo 4 poniendo el número de DTCs a cero y apagando el indicador MIL. Cabe destacar que al simular los DTCs con interruptores, para que el DTC desaparezca se tiene que apagar el interruptor.
- Simular el testigo o indicador MIL mediante un LED rojo que se enciende si se activa uno de los DTCs que conlleva su activación en el reglamento. Además, se dispone de un LED verde que se enciende cuando se establece la comunicación con el tester (inicialización correcta) y otro LED rojo cuando se conectan los 12V a la placa.

El código completo del proyecto se encuentra en el Anexo A.

4.1.3.1 Función setup()

Establece la comunicación serie con el PC a 9.600 bps (Serial.begin) y con el tester a 10.400 bps (Serial3.begin) ya que es la usada en el protocolo a simular.

Por otro lado se configuran los pines digitales de entrada (el proveniente del tester por la línea K para la inicialización a 5bps y los 8 interruptores digitales), así como los de salida (los LEDs de indicador MIL y inicialización correcta).

4.1.3.2 Función loop()

Es la función principal del programa que se repite constantemente. Su funcionalidad cambia según el estado en el que se encuentre:

- Inicialización 1: Llama a una función que se encarga de comprobar que se recibe 0x33 a 5bps y si es así paso al siguiente estado de inicialización.
- Inicialización 2: Llama a una función que acaba la inicialización a 10,4kbps tal como está establecido en el estándar del protocolo y si todo es correcto paso al siguiente estado.
- Petición 1: Testea continuamente la línea k del protocolo esperando un 0x68 que conlleva el inicio de trama de petición y por tanto el cambio al siguiente estado de petición. Si no se respetan los tiempos establecidos se vuelve al primer estado de inicialización.
- Petición 2: Recibe el resto de la trama de petición y si todo es correcto (tiempos, longitud y código de redundancia correctos) paso al estado de respuesta (si no es así vuelvo al inicio).
- Respuesta: Según el modo que nos indique el tester, llama a las funciones encargadas de responder al modo 1 y 3, o reseteo las variables oportunas si se trata del modo 4. Si no es ninguno de estos tres modos se vuelve al estado de espera de trama.

Además, la función loop llamará siempre a la función que comprueba el estado de los interruptores.

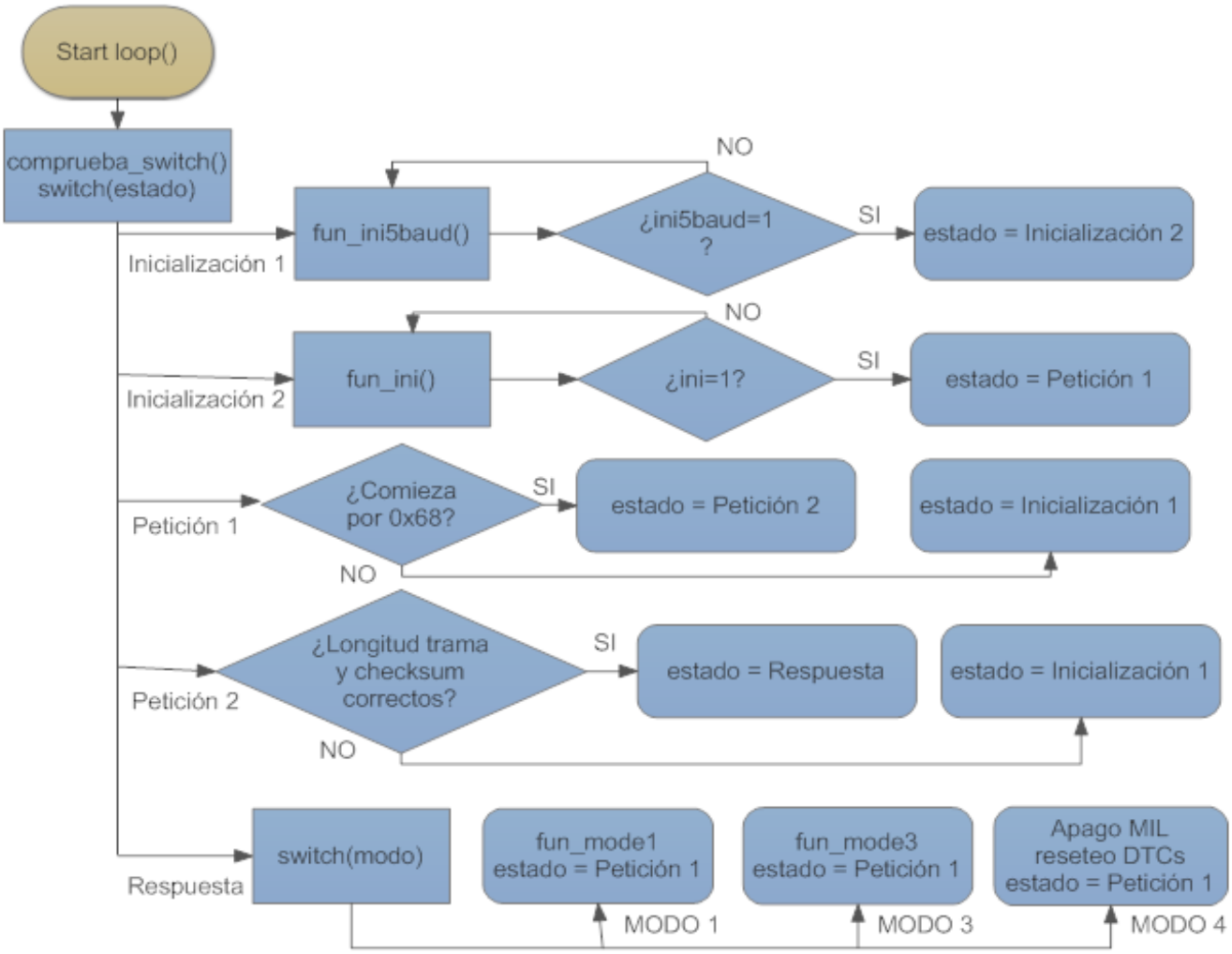


Fig. 55 – Diagrama de bolas de la función loop()

4.1.3.3 Función fun_ini5baud()

Como marca el estándar del protocolo ISO 9141-2 (véase apartado 2.3.3.2.), la inicialización comienza con un mensaje de 0x33 (por parte del tester) a 5bps, por lo que tenemos que comprobar el estado del pin digital proveniente de la línea K así como el tiempo que se mantiene en ese estado. Recordando la estructura del mensaje:

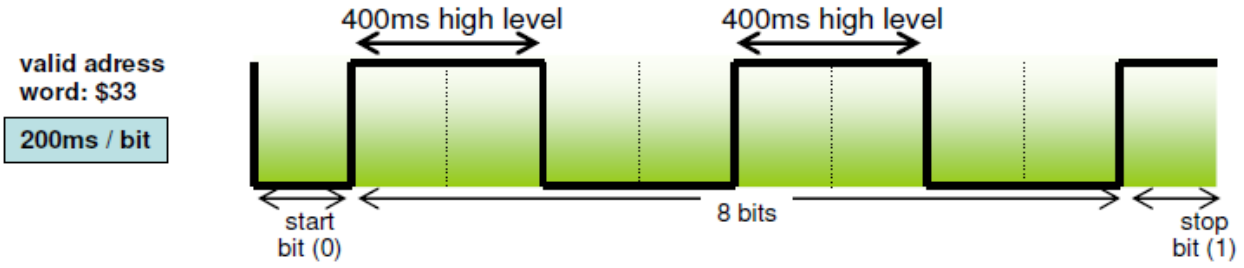


Fig. 56 – Inicialización a 5bps

Comprobaremos mediante distintos estados que el pin está a nivel bajo 200ms (Start bit), luego 400ms a nivel alto, 400ms bajo, 400ms alto y 400ms bajo. Los tiempos se medirán con la función millis() que se igualará a una variable cada vez que se cambie de estado y se aceptará un margen de 10 ms en las comprobaciones temporales para recoger posibles fallos.

Fig. 57 – Diagrama de bolas de la función fun_ini5baud()

4.1.3.4 Función fun_ini()

Después del mensaje 0x33 enviado por el tester, la ECU envía los mensajes de sincronización (0x55) y los dos KeyWords o palabras clave (0x94). Una vez enviados pasamos a un estado de espera hasta recibir el último KW negado (0x6B) por parte del tester y por último enviamos el primer mensaje 0x33 invertido (0xCC). Si todo va bien, levantamos la bandera que cambia de estado en la función loop().

Cabe destacar que hemos utilizado los KWs=0x94 ya que únicamente tenemos una ECU y se asigna el tiempo P2min=0ms (tiempo entre mensajes). En el caso que fueran varias ECUs se utilizarían los KWs=0x08 con su correspondiente KW negado (0xF7) emitido por el tester.

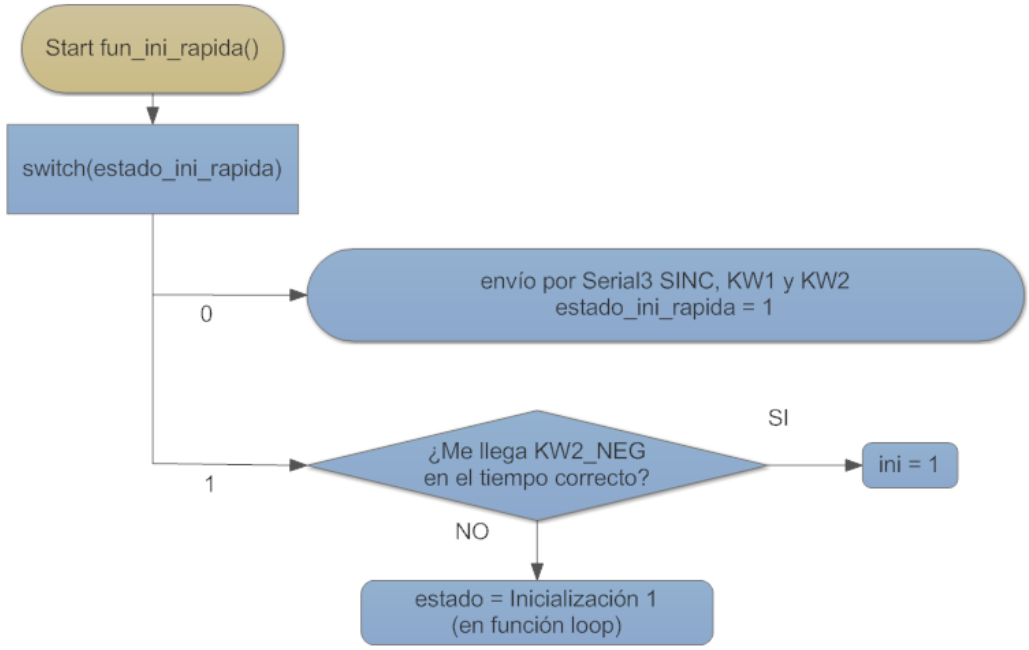


Fig. 58 – Diagrama de bolas de la función fun_ini_rapida()

4.1.3.5 Función fun_mode1 ()

Es la encargada de atender a los mensajes de petición del modo 1 (PIDs en tiempo real). Según el PID requerido por el tester, devolveremos un valor fijo o uno analógico leído de uno de los potenciómetros. Además, el valor de alguno de los 6 PIDs que simularemos de forma analógica también se podrá fijar por serial mediante el PC, si el interruptor 1 está activado. Los PIDs que usaremos en este modo son:

PID	Bytes	Descripción	Min.	Max.	Uds.	Fórmula
00	4	PIDs soportados [01-20]	-	-	-	32 Bits: BitX=1 > PIDX Soportado BitX=0 > PIDX No sop.
01	4	Estado de MIL y número de DTCs almacenados.	-	-	-	Ver más abajo
05	1	Temperatura del refrigerante	-40	215	ºC	A-40
0C	2	Revoluciones por minuto del motor	0	16.383	rpm	((A×256)+B)/4
0D	1	Velocidad del vehículo	0	255	Km/h	A

11	1	Posición del acelerador	0	100	%	A×100/255
1C	1	Estándar OBD conforme al vehículo	-	-	-	Ver más abajo
1F	2	Tiempo transcurrido desde el arranque	0	65.535	seg.	(A×256)+B
20	4	PIDs soportados [21-40]	-	-	-	32 Bits: BitX=1 > PIDX Soportado BitX=0 > PIDX No sop.
2F	1	Nivel de combustible	0	100	%	A×100/255
40	4	PIDs soportados [41-60]	-	-	-	32 Bits: BitX=1 > PIDX Soportado BitX=0 > PIDX No sop.
46	1	Temperatura ambiente	-40	215	ºC	A-40
51	1	Tipo de combustible	-	-	-	Bits codificados

Tabla 20 – PIDs usados en el proyecto

- **PIDs soportados**

Se señalan con un 1 lógico los PIDs que soporta nuestra ECU. A modo de ejemplo veamos el caso de PIDs soportados del 1 al 20 (PID 00):

Byte	A								B								C								D							
Hexadecimal	8				8				1				8				8				0				1				3			
Binary	1	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1
Supported?	Y	N	N	N	Y	N	N	N	N	N	N	Y	Y	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	Y	N	N	Y	Y
PID number	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20

Tabla 21 – MODO 1 PID 00

- **Estado MIL y número de DTCs**

Se trata de 4 bytes (A, B, C y D). Los bits C y D indican si existen test de diferentes partes del vehículo dependiendo del tipo de encendido (chispa o compresión) y si éstos test están completados.

En nuestro caso sólo nos importará el byte A que indica si el indicador MIL está encendido y el número de DTCs almacenados. El resto los dejaremos a 0 ya que no supondrán cambio alguno en nuestro simulador.

Bit	Name	Definition
A7	MIL	Indica si el indicador MIL está encendido (o debería estarlo)
A6-A0	DTC_CNT	Número de errors DTC confirmados para mostrar.
B7	RESERVED	Reservado (debe ser 0)
B3	*NO NAME*	0 = Encendido del monitor por chispa 1 = Encendido del monitor por compresión

Tabla 22 – MODO 1 PID 01 Parte 1

	Test available	Test incomplete
Misfire	B0	B4
Fuel System	B1	B5
Components	B2	B6

Tabla 23 - MODO 1 PID 01 Parte 2

- **Temperatura del refrigerante (coolant)**

Se trata de uno de los 6 PIDs que podemos simular analógicamente (pin A5). El potenciómetro devuelve valores entre 0 y 1023 $(2^{10}\ valores)$ según la posición del rotatorio y escalaremos la temperatura a un rango de 0 a 200ºC. Además, la tabla nos indica que en la codificación del PID se restan 40ºC, por lo que debemos sumar esta cantidad antes de mandarlo al tester. Con A el valor obtenido analógicamente:

trama_tx[5] = A*(200.0/1023.0) + 40

- **Revoluciones por minuto (rpm)**

También se puede leer analógicamente (A2) y se escalará a un rango de 0 a 4.000 rpm. Para que en la codificación del PID quede un resultado de A + B, los dos bytes a enviar serán:

A = A *(4000.0/1023.0)

trama_tx[5] = A*(4/256)

trama_tx[6] = A - A*(4/256)

- **Velocidad del vehículo (speed)**

Se puede leer analógicamente (A1) y ésta vez lo escalamos a un rango de 0 a 255 para que abarque todos los valores posibles en la simulación (2^8 valores). Este PID no tiene codificación.

trama_tx[5] = A*(255.0/1023.0)

- **Posición del acelerador (throttle)**

Se puede leer analógicamente (A0). Hacemos el mismo escalado que con la velocidad pero ésta vez la codificación lo convertirá en un rango de 0 a 100 (porcentaje).

trama_tx[5] = A*(255.0/1023.0)

- **Estándar OBD**

Value	Description
1	OBD-II as defined by the CARB
2	OBD as defined by the EPA
3	OBD and OBD-II
4	OBD-I
5	Not OBD compliant
6	EOBD (Europe)
7	EOBD and OBD-II
8	EOBD and OBD
9	EOBD, OBD and OBD II
10	JOBD (Japan)
11	JOBD and OBD II
12	JOBD and EOBD
13	JOBD, EOBD, and OBD II
14	Reserved
15	Reserved
16	Reserved
17	Engine Manufacturer Diagnostics (EMD)
18	Engine Manufacturer Diagnostics Enhanced (EMD+)
19	Heavy Duty On-Board Diagnostics (Child/Partial) (HD OBD-C)
20	Heavy Duty On-Board Diagnostics (HD OBD)
21	World Wide Harmonized OBD (WWH OBD)
22	Reserved
23	Heavy Duty Euro OBD Stage I without NOx control (HD EOBD-I)
24	Heavy Duty Euro OBD Stage I with NOx control (HD EOBD-I N)
25	Heavy Duty Euro OBD Stage II without NOx control (HD EOBD-II)
26	Heavy Duty Euro OBD Stage II with NOx control (HD EOBD-II N)
27	Reserved
28	Brazil OBD Phase 1 (OBDBr-1)
29	Brazil OBD Phase 2 (OBDBr-2)
30	Korean OBD (KOBD)
31	India OBD I (IOBD I)
32	India OBD II (IOBD II)
33	Heavy Duty Euro OBD Stage VI (HD EOBD-IV)
34-250	Reserved
251-255	Not available for assignment (SAE J1939 special meaning)

Tabla 24 – Tipos de estándar OBD

En nuestro caso tendremos un valor de 9 ya que nuestro protocolo es soportado por OBD, OBDII y EOBD.

trama[5] = 0x09

- **Tiempo desde el arranque**

Haremos uso de la función millis() (ms desde el arranque) escalada a segundos y preparada para la codificación del PID.

A = millis() / 1000
trama_tx[5] = A/256
trama_tx[6] = A - A/256

- **Nivel de combustible**

Se puede leer analógicamente (A3) y como en el caso de la posición del acelerador, escalaremos a un rango de 0 a 255 y en la codificación se pasará a porcentaje.

Trama_tx[5] = A*(255.0/1023.0)

- **Temperatura ambiente**

Último de los PIDs que se pueden leer de forma analógica (A4). Lo escalaremos a un valor de 0 a 40ºC y como en el caso de la temperatura refrigerante se le deberán sumar 40ºC.

Trama_tx[5] = A*(40.0/1023.0) + 40

- **Tipo de combustible**

0	Not available
1	Gasoline
2	Methanol
3	Ethanol
4	Diesel
5	LPG
6	CNG
7	Propane
8	Electric
9	Bifuel running Gasoline
10	Bifuel running Methanol
11	Bifuel running Ethanol

12	Bifuel running LPG
13	Bifuel running CNG
14	Bifuel running Propane
15	Bifuel running Electricity
16	Bifuel running electric and combustion engine
17	Hybrid gasoline
18	Hybrid Ethanol
19	Hybrid Diesel
20	Hybrid Electric
21	Hybrid running electric and combustion engine
22	Hybrid Regenerative
23	Bifuel running diesel

Tabla 25 – Tipos de combustible

Para nuestro ejemplo usaremos el tipo diésel.

trama[5] = 0x04

En cualquier PID, según el número de bytes de datos a enviar, siendo

trama_tx[5] = A; trama_tx[6] = B; trama_tx[7] = C y trama_tx[5] = D,

tendremos una longitud de trama diferente, que debemos tener en cuenta para que se calcule correctamente el posterior checksum.

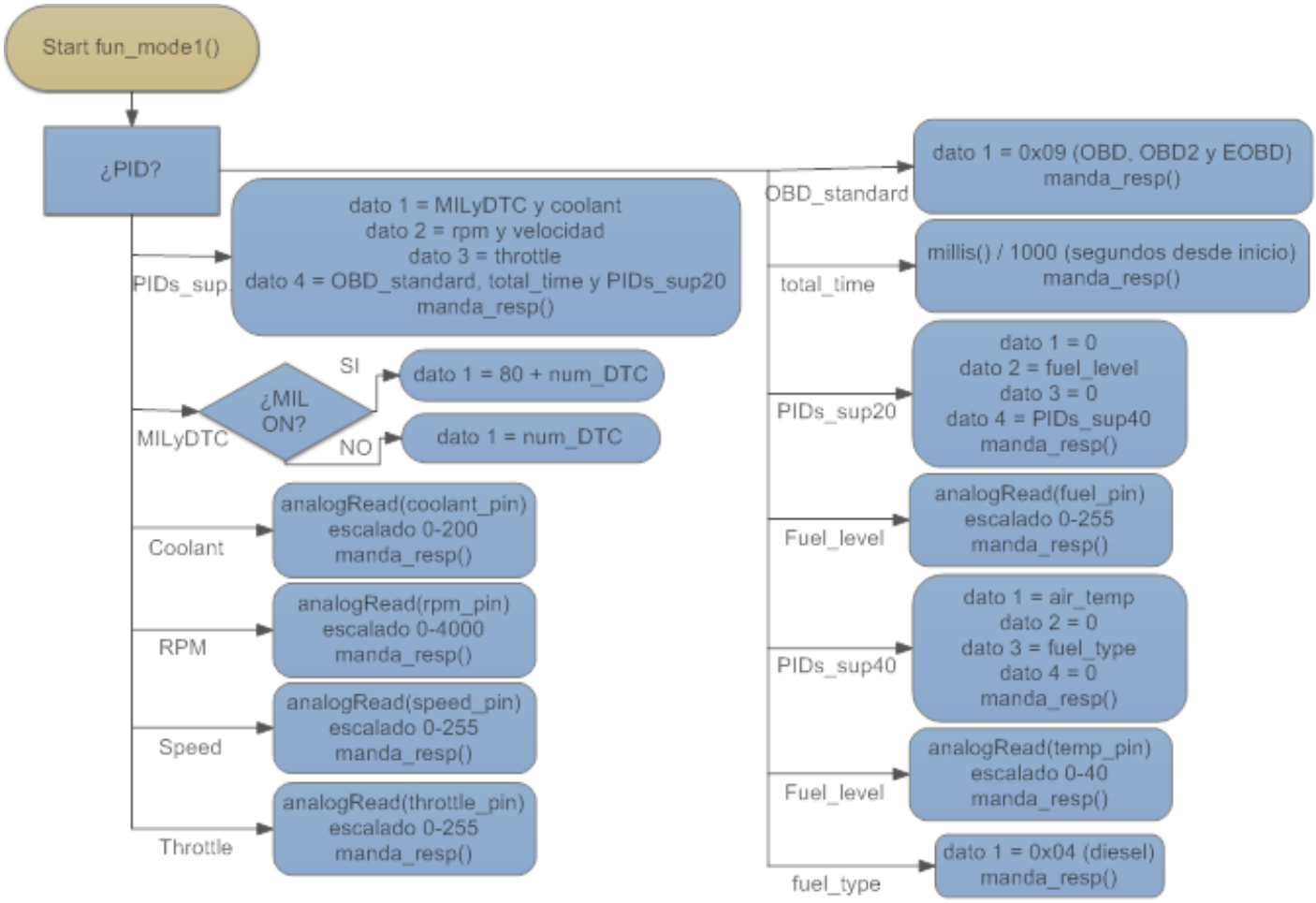


Fig. 59 – Diagrama de bolas de la función fun_mode1()

4.1.3.6 Función fun_mode3()

Función encargada de responder al modo 3 con el/los DTC/s que se haya/n activado mediante los interruptores. Por limitaciones de la aplicación a usar, únicamente se informará de cuatro DTCs activados a la vez, pero es lógico, ya que en la realidad no suelen aparecer más de cuatro errores simultáneamente.

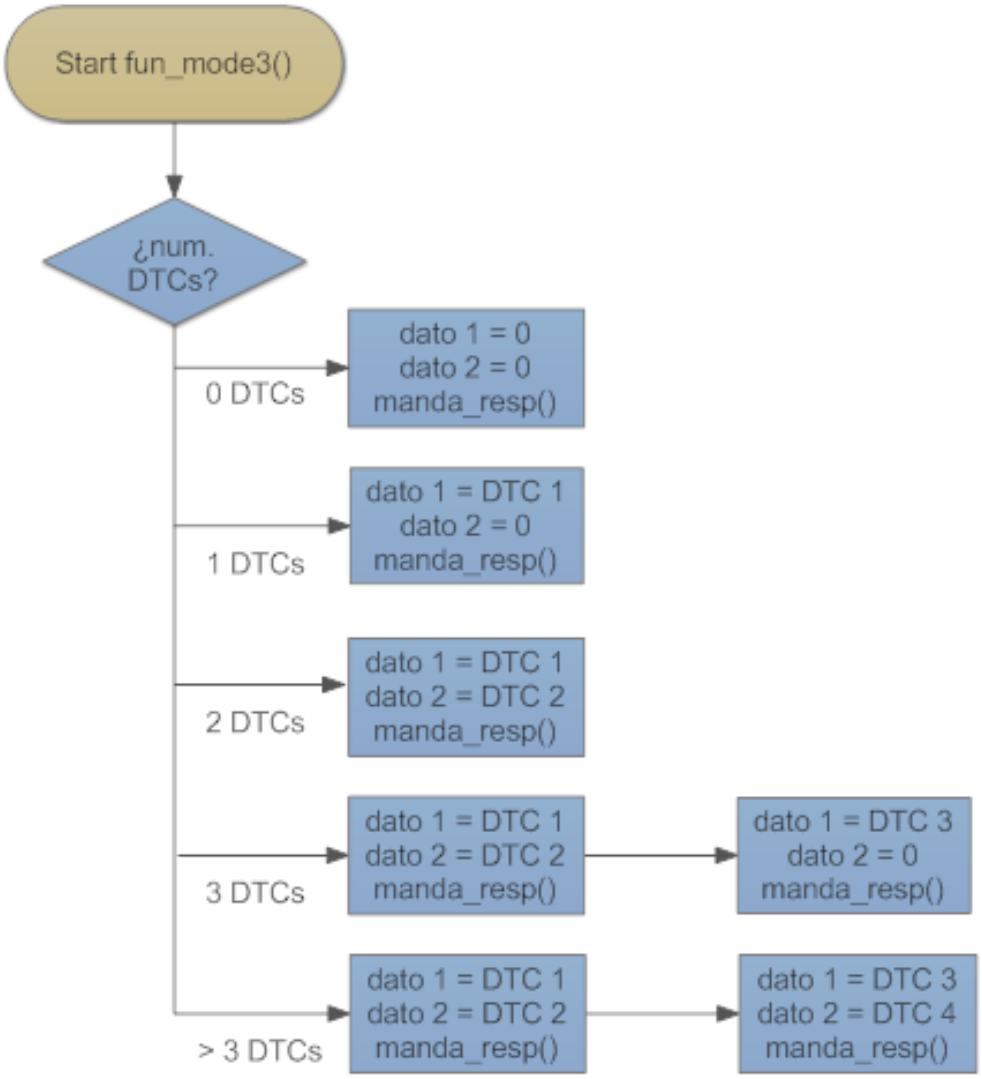


Fig. 60 – Diagrama de bolas de la función fun_mode3()

Los DTCs corresponientes a cada interruptor en nuestro simulador serán:

- **P0301 (SW2)**

Mala combustión detectada en cilindro 1.

HB=00000011 (0x03)

LB=00000001 (0x01)

Síntoma: Problemas para arrancar.

Causa: Fallo en el sensor de oxígeno, inyector de combustible o convertidor catalítico.

Corrección: Resetear código de error y si vuelve a aparecer revisar conectores y cableado que conducen al cilindro en cuestión.

- **P0340 (SW3)**

Sensor A de posición del árbol de levas, mal funcionamiento del circuito (banco 1).

HB=00000011 (0x03)
LB=01000000 (0x40)

Síntoma: Motor tarda en arrancar, marcha mínima inestable y humo negro en el escape.

Causa: Banda de distribución mal sincronizada.

Corrección: Sincronizar banda de distribución.

- **P0217 (SW4)**

Condición de sobrecalentamiento del motor.

HB=00000010 (0x02)
LB=00010111 (0x17)

Síntoma: MIL encendido, pérdidas de potencia o baja refrigeración.

Causa: Pérdidas en el sistema refrigerante.

Corrección: Cambiar el termostato y después de volver a llenar el sistema de refrigeración, reiniciar el motor y verificar que no se está sobrecalentando.

- **P0171 (SW5)**

Sistema demasiado desfavorable (banco 1).

HB=00000001 (0x01)
LB=01110001 (0x71)

Síntoma: Falta de potencia y sobretensión en la aceleración.

Causa: Sensor de masa de aire MAF sucio o fuga de vacío.

Corrección: Limpiar el sensor MAF y cambiar las mangueras de vacío.

- **P0500 (SW6)**

Sensor de velocidad del vehículo en mal funcionamiento.

HB=00000101 (0x05)
LB=00000000 (0x00)

Síntoma: Pérdida de frenos antibloqueo, el velocímetro no funciona adecuadamente o el límite de RPM del vehículo disminuido.

Causa: Cable que lleva al sensor de velocidad del vehículo roto o sensor de velocidad averiado.

Corrección: Resetear el código de error y si vuelve a aparecer comprobar los conectores y cables que conducen al sensor de velocidad.

- **P0420 (SW7)**

Sistema catalizador, eficiencia debajo del umbral (Banco 1).

HB=00000100 (0x04)
LB=00100000 (0x20)

Síntoma: MIL encendido y aparentemente ningún síntoma en la conducción.

Causa: Fallo en el sensor de oxígeno o combustible con plomo usado cuando se pedía sin plomo.

Corrección: Comprobar si hay fugas de escape en el colector, tuberías o convertidor catalítico e inspeccionar el sensor de oxígeno.

- **P0068 (SW8)**

Posición del acelerador (TP) incompatible con el sensor de masa de flujo de aire (MAF).

HB=00000000 (0x00)
LB=01101000 (0x68)

Síntoma: MIL encendido y motor en marcha duro.

Causa: fuga entre el sensor de masa de aire y colector de admisión o mangueras sueltas o agrietadas.

Corrección: Abrir el capó y comprobar el filtro de aire, comprobar si hay fuga en el colector de admisión y revisar el conector de corrosión.

4.1.3.7 Función manda_resp()

Se encarga de enviar por serie al tester OBD la trama de respuesta a su petición. Los bytes de datos ya se han ido rellenando según el modo requerido por lo que en esta función completamos la trama con los bytes de cabecera y el checksum final.

4.1.3.8 Función comprueba_switch()

Va comprobando uno por uno el estado de cada interruptor. Si se activa el primero se llama a una función que leerá datos del puerto serie con el PC. Si se activan cualquiera de los otros 7, se rellenan los datos del DTC tal y como indica el estándar (apartado 2.4.3.). Si el DTC conlleva el encendido del indicador MIL, encenderemos el LED correspondiente.

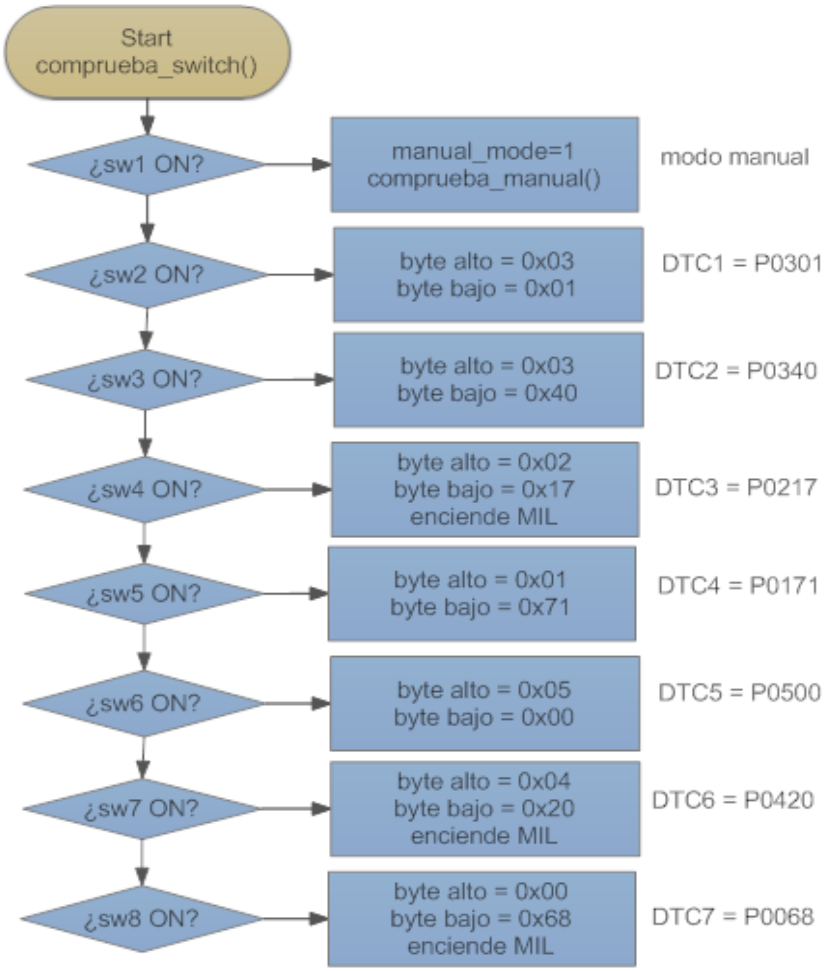


Fig. 61 – Diagrama de bolas de la función comprueba_switch()

4.1.3.9 Función comprueba_manual()

Si se ha activado el interruptor 1 (modo manual) se llama a esta función. Se encarga de leer los datos recibidos por el PC (puerto serie) escritos de la forma:

PID = valor

Cuando se activa el modo manual, todos los PIDs se ponen al valor predeterminado 0, a partir de entonces se van asignando los valores deseados por serie (uno por uno) para simular un escenario determinado.

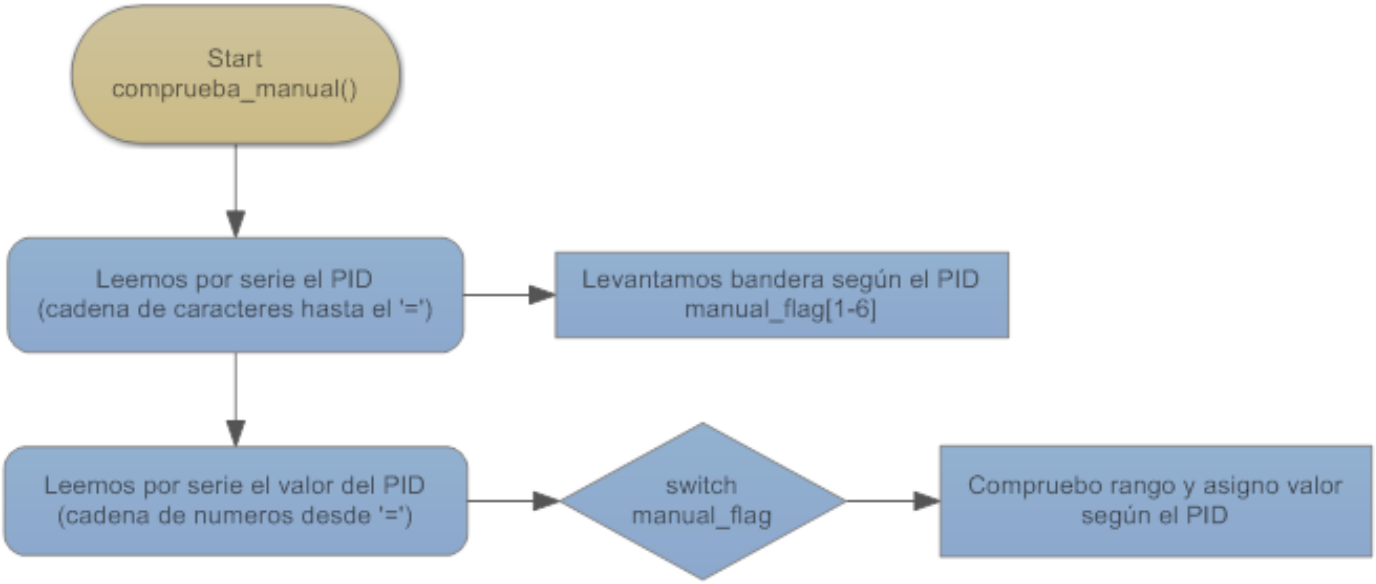


Fig. 62 - Diagrama de bolas de la función comprueba_manual()

4.2 Software del dispositivo tester

Existen numerosos programas para la visualización de los datos de diagnóstico. Para PC el más conocido es el ScanTool y para smartphone existen infinidad de apps, entre las más destacadas están Torque Pro, OBD Car Tracker, OBD Fusion, OBD Car Doctor, OBD Link y OBD eZWay. Para este proyecto usaremos un programa para Smartphone ya que supone mayor comodidad al tener el dispositivo tester en tu bolsillo y entre las diferentes opciones elegiremos la app Torque Pro, ya que es la más completa e intuitiva a la hora de visualizar los datos del vehículo. Con ella se pueden obtener datos de funcionamiento del vehículo, tales como:

- RPM reales del motor, aunque tu vehículo no disponga de dicho indicador en el salpicadero.
- Velocidad.
- Aceleración.
- Potencia del motor y par motor instantáneos.
- Códigos de error del motor con información detallada.
- Estado del sistema eléctrico y fusibles.
- Seguimiento del mantenimiento del vehículo.
- Lectura de las emisiones del vehículo.

- Temperatura de transmisión.
- Grabación de vídeo de viaje con superposición de datos OBDII.
- Modo HUD (Head Up Display) para conducción nocturna.

La interfaz del programa se muestra en la siguiente figura 64, donde se observan las diferentes opciones que tiene la app. En nuestro caso solo nos hará falta la información en tiempo real y los códigos de falla (primeros dos iconos de la rueda).

Fig. 64 – portada de la app Torque Pro

Lo primero que hace la aplicación es conectarse mediante bluetooth o wifi al dispositivo de diagnóstico y acto seguido comienza a probar las distintas inicializaciones de los diferentes protocolos OBD (cada una por su pin del conector J1962) y una vez se establece la comunicación (mensaje en la parte superior de ECU OK) comienza a mandar tramas de petición automáticamente según los PIDs que se encuentren en pantalla.

Dentro de la opción de información en tiempo real, la app nos deja la opción de elegir el PID que queramos, así como el modo de visualización del mismo (dial, semiesfera, barra, gráfico, pantalla digital o histograma, entre otros). Aquí añadiremos las pantallas de los PIDs que contempla nuestro programa.

Fig. 65 – Interfaz del modo 1 de Torque Pro

Dentro de la opción de códigos de falla, nos aparecen los distintos DTCs almacenados y su descripción. Aquí saldrán los DTCs que hayamos activado mediante los interruptores (modo 3).

Fig. 66 – Interfaz modo 3 de Torque Pro

Por útimo, dentro de la opción de códigos de falla, se encuentra el borrado de errores en la ECU. Esta opción envía a la ECU una petición del modo 4 por lo que la usaremos para resetear el número de DTCs y apagar el indicador MIL (si procede).

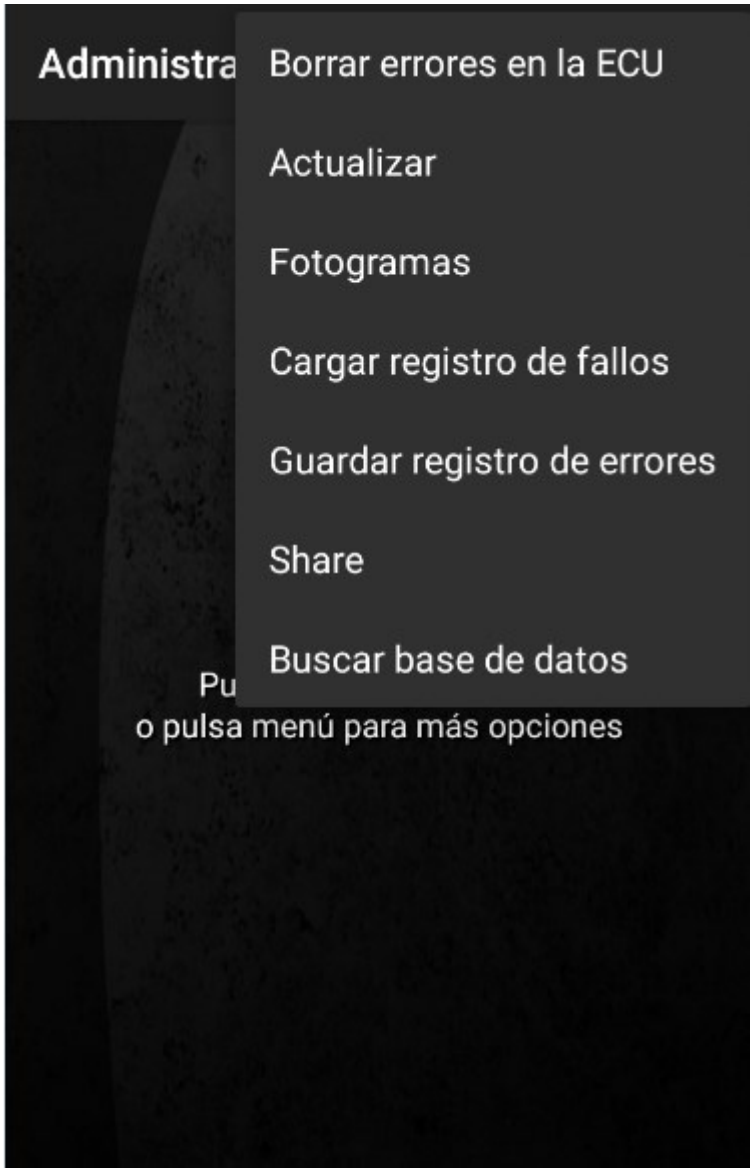


Fig. 67 – Interfaz del modo 4 de Torque Pro

5 PRUEBAS

En este capítulo vamos a comprobar todos los aspectos abarcados en el simulador. Primero veremos su inicialización, el correcto envío y recibo de tramas según el modo que se pide, la simulación de algunos parámetros del vehículo en tiempo real (ya sea mediante potenciómetros o por el puerto serie), la simulación de distintos DTCs, la activación del testigo MIL (si procede) y el borrado de los DTCs.

5.1 Inicialización

Lo primero que debemos hacer es insertar en el Arduino MEGA la placa PCB fabricada en la correcta posición, alimentar la placa por la borna de 2 pines (se ha usado un transformador comercial AC-DC de 12V), conectar correctamente el dispositivo de diagnóstico a la borna de 3 pines (alimentación, K-line del protocolo ISO9141-2 y tierra), conectar el Arduino al PC mediante el cable USB A/B y cargar en el mismo el programa del simulador.

En la siguiente figura 68 se observa cómo se enciende el LED rojo que simula la luz de encendido o “power” cuando se conecta la alimentación.

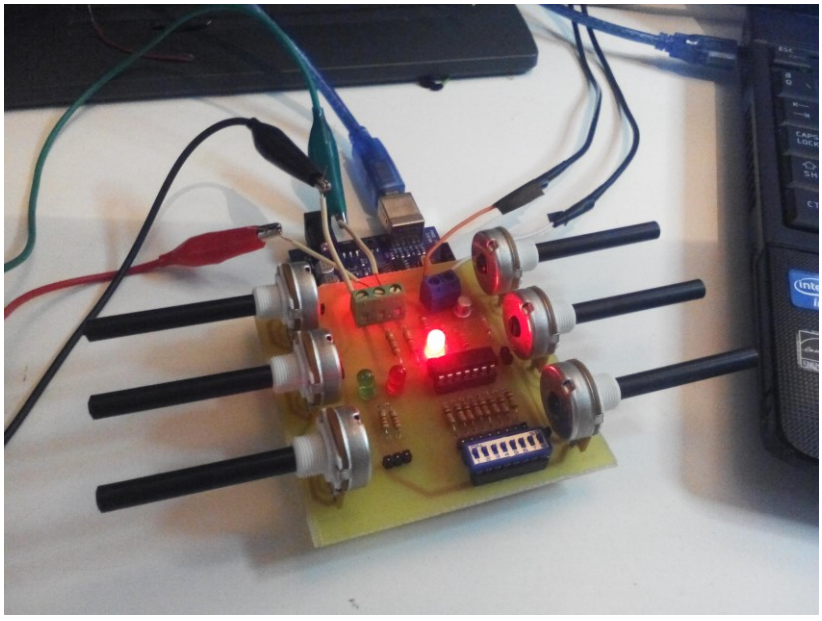


Fig. 68 – Simulación del LED POWER

A continuación abrimos la app Torque desde cualquier dispositivo con sistema Android y esperamos que se realice la inicialización. Previamente se debe vincular el dispositivo de diagnóstico por bluetooth (una vez alimentado con 12V).

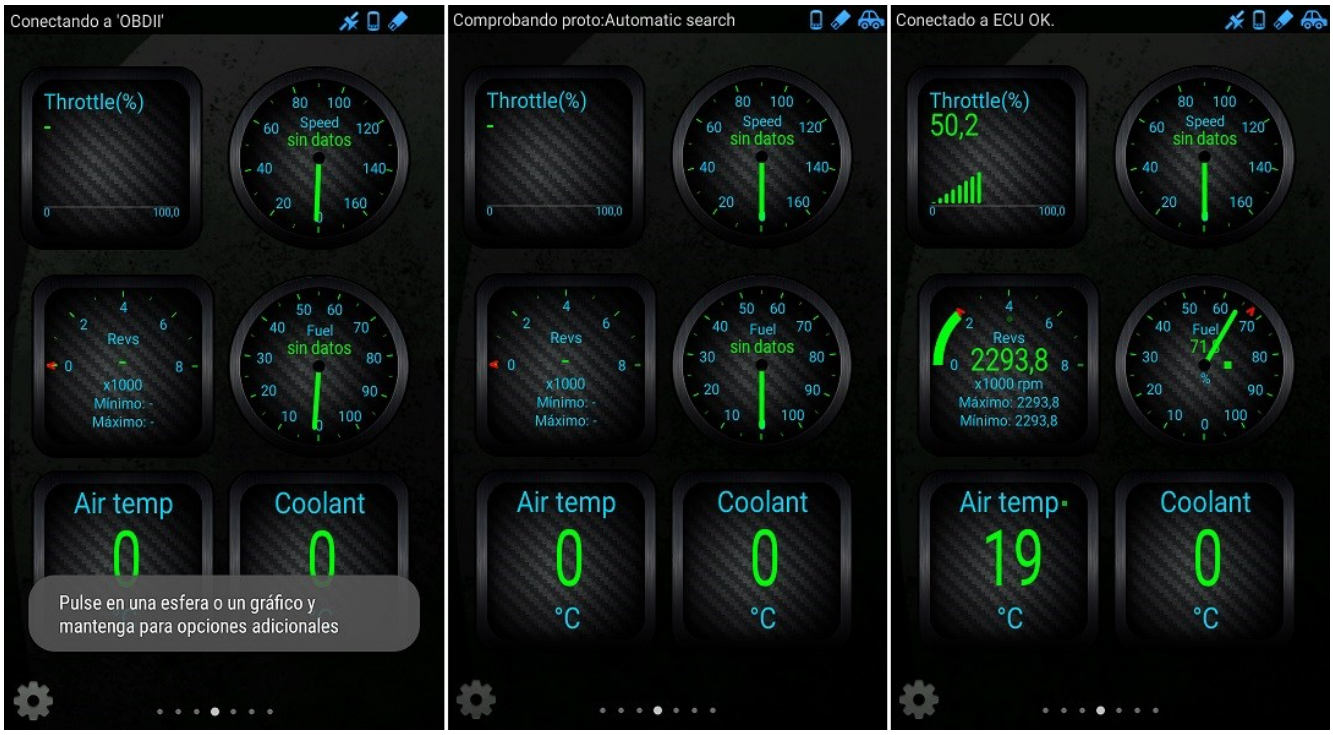


Fig. 69 – Simulación de la inicialización del protocolo ISO9141-2

Se pueden observar los diferentes pasos en la inicialización, el tester primero verifica la alimentación y la conexión bluetooth con el dispositivo Android, luego debe reconocer el protocolo que usa el simulador del vehículo y después enviar las tramas de petición de los diferentes PIDs qe se encuentran en pantalla, si éstos se reciben correctamente y en el tiempo establecido se tendrá un resultado satisfactorio. Los valores de los parámetros en la tercera pantalla son los que indican las posiciones de los potenciómetros del simulador.

En la siguiente figura 70 se observa cómo se enciende el LED verde que simula la luz de inicialización cuando el tester y la ECU están preparados para recibir y enviar tramas.

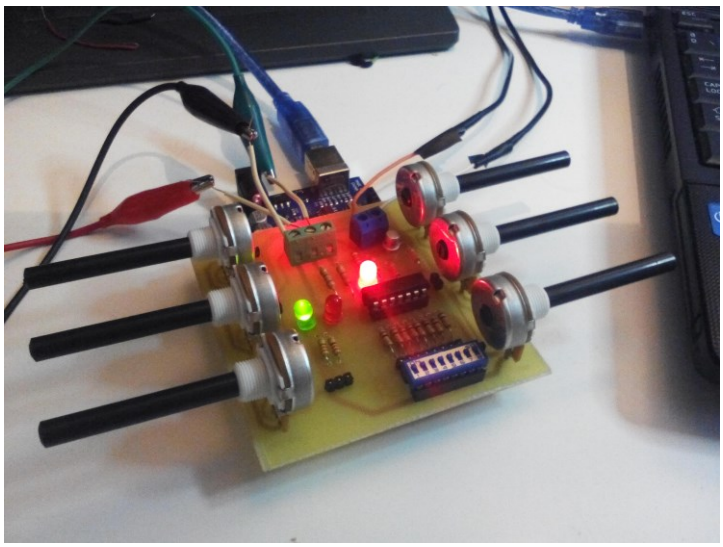


Fig. 70 – Simulación del LED INI

.

5.2 Parámetros PID en tiempo real

Una vez inicializado el protocolo, el tester envía peticiones sin cesar de los PIDs requeridos en pantalla. En la siguiente imagen del puerto serie de Arduino se observan las distintas tramas que nos manda el tester.

```
MODO=
1
PID=
0
PIDs soportados [1-20]

MODO=
1
PID=
2F
Nivel de combustible

MODO=
1
PID=
C
Revoluciones por minuto

MODO=
1
PID=
11
Posicion del acelerador

MODO=
1
PID=
46
Temperatura ambiente

MODO=
1
PID=
5
Temperatura del refrigerante

MODO=
1
PID=
D
Velocidad del vehiculo
```

Fig. 71 – Tramas recibidas desde el tester

A continuación comprobaremos los 6 PIDs implementados en el simulador mediante los potenciómetros.

5.2.1 Throttle

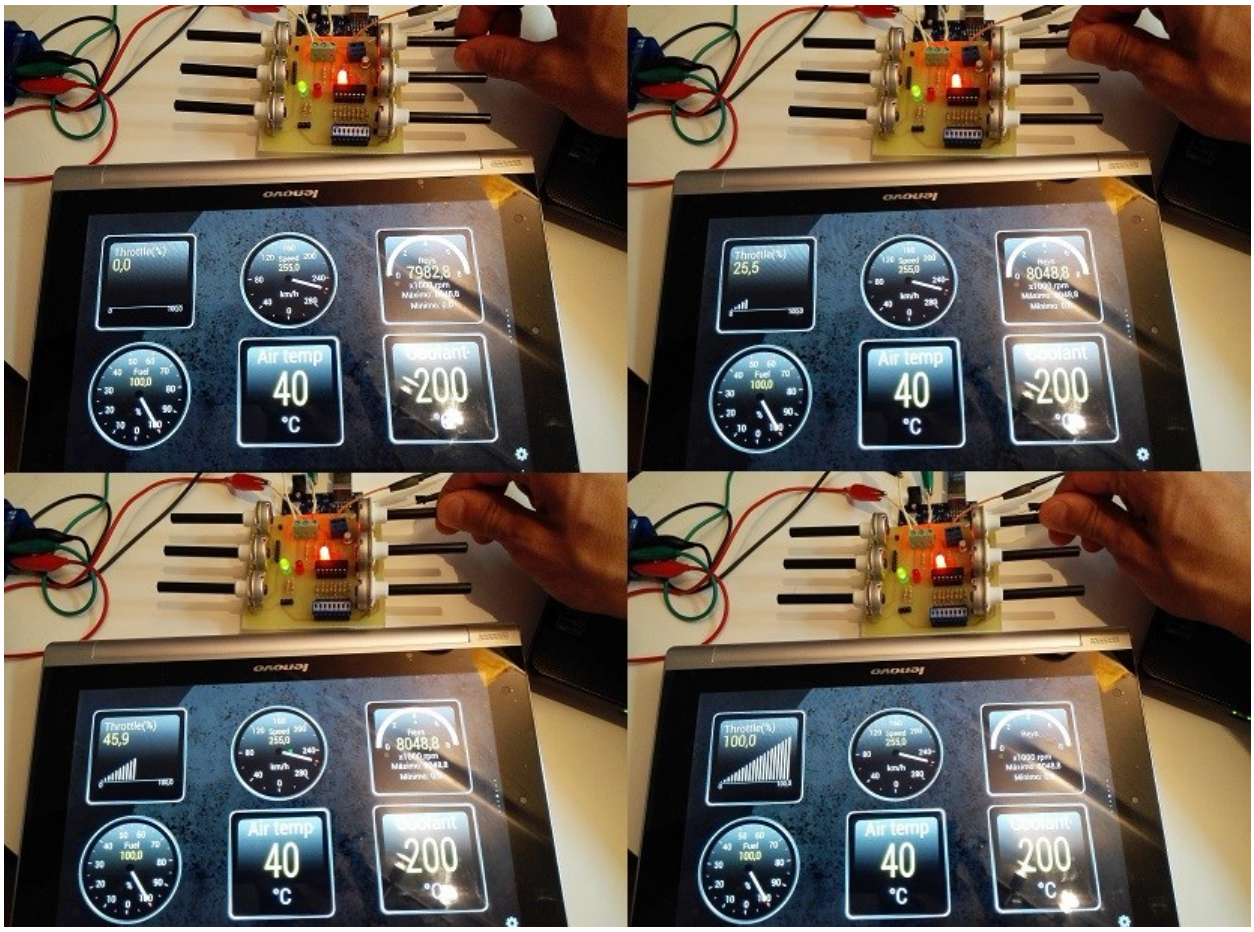


Fig. 72 – Simulación del parámetro throttle (posición del acelerador)

5.2.2 Speed

Fig. 73 – Simulación del parámetro speed (velocidad)

5.2.3 RPM

Fig. 74 – Simulación del parámetro RPM (revoluciones por minuto)

5.2.4 Fuel level

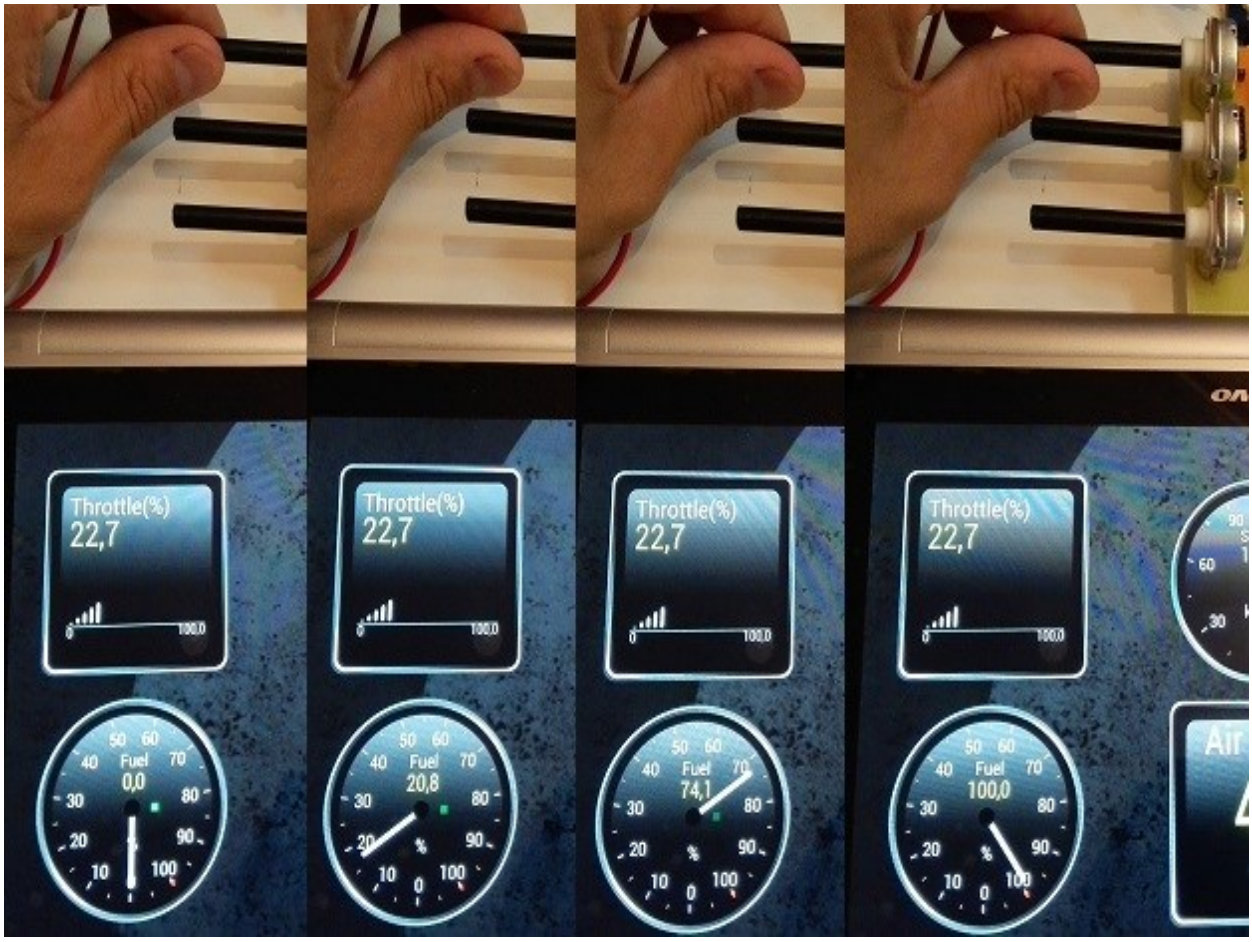


Fig. 75 – Simulación del parámetro fuel level (nivel de combustible)

5.2.5 Air temperature

Fig. 76 – Simulación del parámetro Air temperature (temperatura ambiente)

5.2.6 Coolant

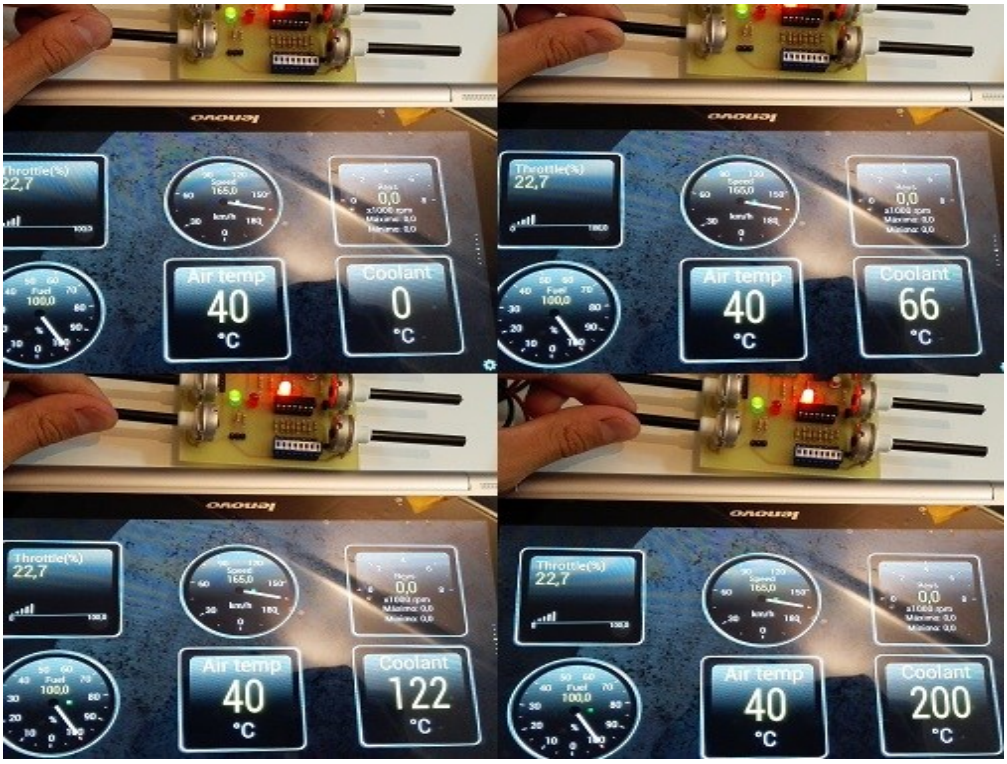


Fig. 77 – Simulación del parámetro coolant (temperatura refrigerante)

5.2.7 Otros PID

Por otro lado podemos ver el tiempo desde el arranque (PID adicional) e información que nos ofrece la app Torque una vez conectada al vehículo.

Fig. 78 – Simulación de otros parámetros adicionales

5.2.8 Simulación por serie

Además de la opción de simulación mediante potenciómetros, tenemos la posibilidad de hacerlo por el puerto serie de Arduino escribiendo en el terminal PID=valor y teniendo activado el interruptor 1 del switch DIP.

En el siguiente ejemplo pondremos la velocidad a 100km/h, habiendo activado antes el SW1 (en caso contrario no tendrá efecto).

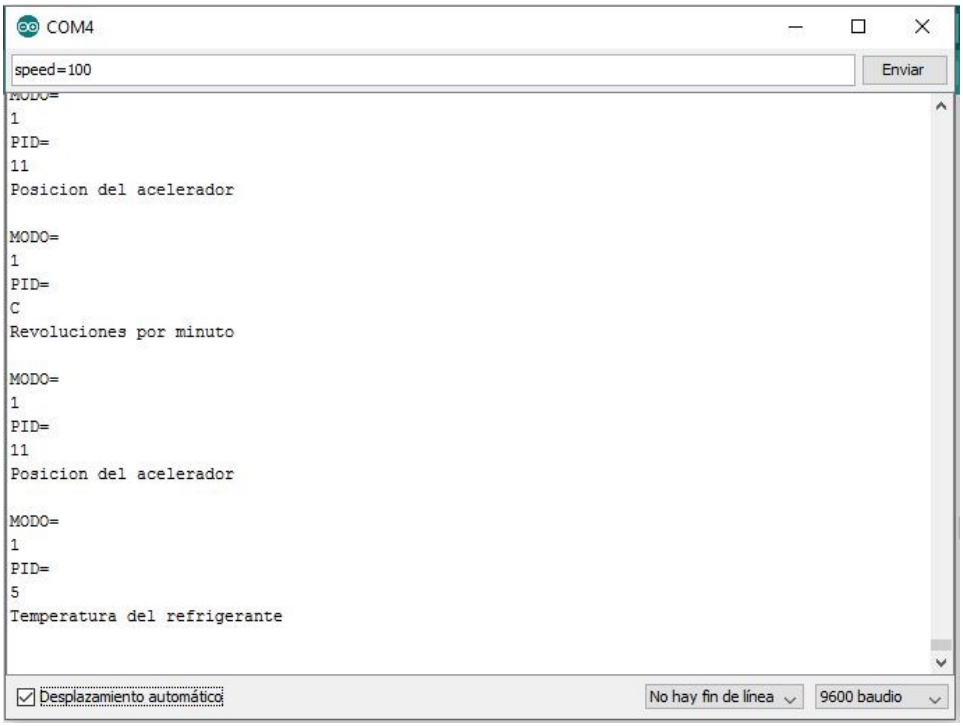


Fig. 79 – Ejemplo de simulación por el puerto serie

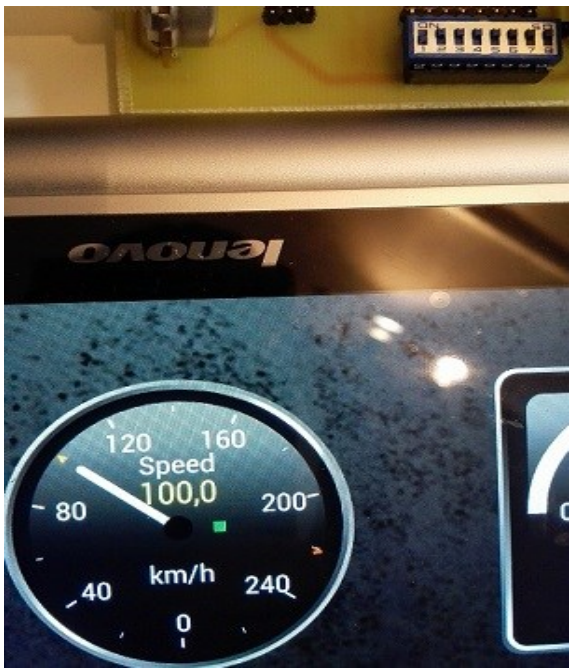


Fig. 80 – Simulación de la velocidad por el puerto serie

Cabe destacar que si simulamos otro parámetro por el puerto serie, el valor de la velocidad (en este caso) se mantiene, pero si desactivamos el SW1, el valor vuelve a ser el indicado por su correspondiente potenciómetro.

5.3 Simulación de DTCs

Cambiando al apartado de Fault Codes (códigos de falla) de la app Torque nos encontramos con que no tenemos ningún código de falla o DTC. Si procedemos a activar el SW2 que simula el DTC 1 = P0301 y actualizamos:

Fig. 81 – Simulación del DTC 1

Ahora activamos también el SW3 como se observa en la siguiente figura 82 (observamos que el indicador MIL sigue apagado) y comprobamos el nuevo resultado.

Fig. 82 – Simulación de los DTC 1 y 2

Activamos ahora únicamente el SW4 (DTC 3) que sí enciende el indicador MIL por tratarse de un código de avería grave.

Fig. 83 – Simulación del DTC 3

Activamos los 4 DTCs restantes (4, 5, 6 y7) y observamos el nuevo resultado.

Fig. 84 – Simulación de los DTCs 4, 5, 6 y 7

Por último, haremos un borrado de DTCs así como del indicador MIL, con el correspondientre apartado de la aplicación Torque. Cabe destacar que los interruptores deben de estar desactivados antes del borrado o volverá a saltar el código de falla correspondiente.

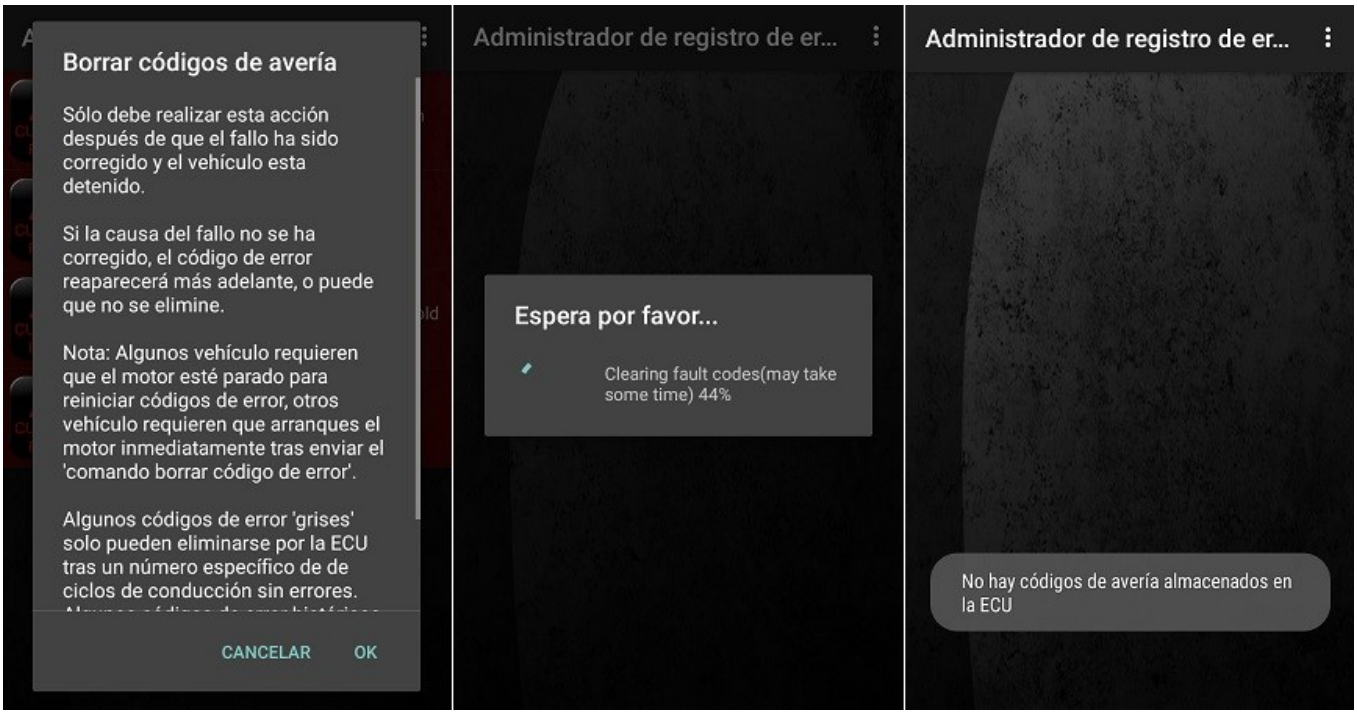


Fig. 85 – Borrado de DTCs

Esta acción deja el simulador del vehiculo limpio de averías y con el testigo Check Engine apagado, listo para nuevas pruebas.

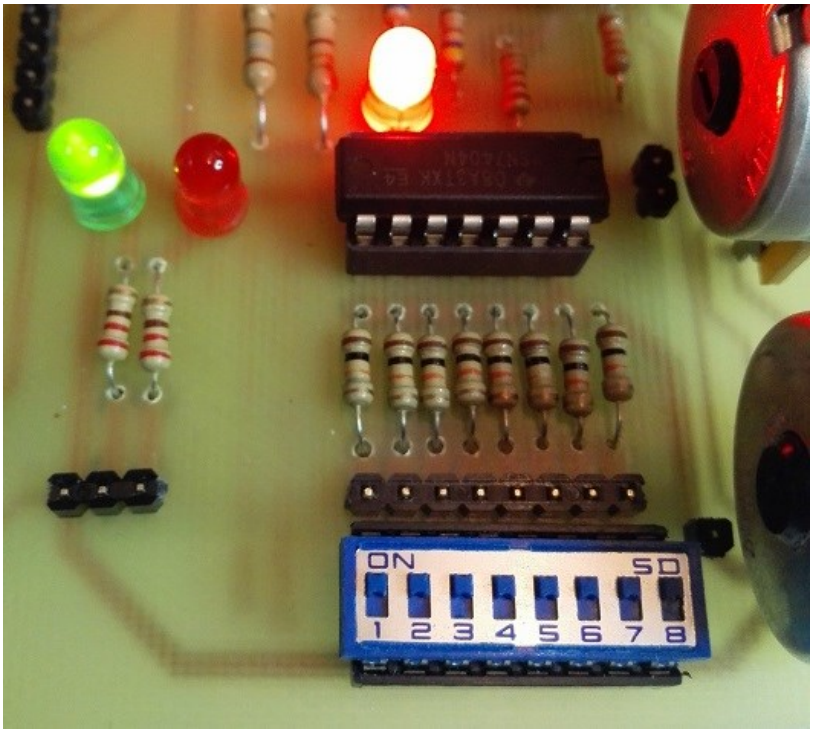


Fig. 86 – Simulador sin ninguna avería

6 COMENTARIOS FINALES

En este último capítulo se exponen los objetivos alcanzados y el resultado obtenido, distinguiendo sus características más relevantes. Finalmente se debaten posibles mejoras para hacer más completo el simulador.

6.1 Conclusiones

Se ha diseñado y fabricado una tarjeta compatible con el Arduino MEGA que simula la unidad de control de un vehículo que soporte el protocolo de diagnóstico ISO 9141-2. Este simulador es capaz de comunicarse con cualquier dispositivo de diagnóstico mediante el protocolo nombrado, simulando parámetros del vehículo en tiempo real y códigos de averías que puedan aparecer durante la conducción, encendiéndose el testigo Check Engine si se trata de una avería grave.

Con este proyecto se ha conseguido implementar un simulador con las siguientes características:

- Económico: Sin contar con el Arduino, el presupuesto del simulador no alcanza los 30€, a diferencia de los simuladores disponibles en el mercado que tienen un precio medio de 120€.
- Flexible: Conociendo la lista de PIDs (con su conversión) y de DTCs, se pueden agregar o eliminar los diferentes parámetros a simular, así como los códigos de averías que se quieran simular.
- Útil: Se pueden realizar una gran cantidad de pruebas en múltiples escenarios, sin la necesidad de tener un vehículo real.

6.2 Mejoras futuras

Este proyecto sirve como base para su continuación futura, ya que con la misma tarjeta ya fabricada, conociendo todos los aspectos teóricos del sistema OBD, se podrían implementar el resto de modos de medición del vehículo.

Conociendo el estándar del resto de protocolos, se puede extender fácilmente el código añadiendo nuevas funciones de inicialización, envío y recibo de tramas según el protocolo, ya que las funciones relacionadas con los modos de medición son comunes a todos ellos.

Por último, podría integrar en otros simuladores como OMNET++ y SUMO, para simular situaciones de tráfico y tener un entorno más realista con la interacción de nuestro vehículo con otros simulados por otros simuladores.

BIBLIOGRAFÍA

[1] *Autoxuga. "Diagnosis multimarca. 1.3 Antecedentes históricos de la diagnosis".*

http://www.autoxuga.com/libros/diagnosismultimarca/index.htm

[2] *Autoxuga. "Diagnosis multimarca. 1.5 Standard OBD, OBDII".*

[3] *Aficionados a la mecánica "OBD (On Board Diagnostic)"*

http://www.aficionadosalamecanica.com/obd2.htm

[4] *ISO 15031-3 (2016) "Road vehicles - Communication between vehicle and external equipment for emissions-related diagnostics ".*

http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64636

[5] *SAE 15031-3 (2001). "Diagnostic Connector Equivalent to ISO/DIS 15031-3: December 14, 2001"*

http://standards.sae.org/j1962_201207

[6] *Euskalnet. "Estándar de comunicaciones RS232-C"*

http://www.euskalnet.net/shizuka/rs232.htm

[7] *ELM Electronics "ELM327DSF – OBD to RS232 Interpreter"*

www.elmelectronics.com

[8] *SAE J1850 (2006) "Class B Data Communications Network Interface".*

http://standards.sae.org/j1850_200606

[9] *AENOR. UNE-ISO 9141-2 (2013) "Road vehicles -- Diagnostic systems -- Part 2: CARB requirements for interchange of digital information."*

http://www.aenor.es/aenor/normas/normas/fichanorma.asp?tipo=N&codigo=N0052032#.V2lx4KJh3_g

[10] *AENOR. UNE-ISO 14230-1 (2015) "Road vehicles. Diagnostic communication over K-Line (DoK-Line). Part 1: Physical layer."*

http://www.aenor.es/aenor/normas/normas/fichanorma.asp?tipo=N&codigo=N0055051#.V2lyHaJh3_g

[11] *AENOR. ISO 15765-4 (2016) "Road vehicles -- Diagnostic communication over Controller Area Network (DoCAN) -- Part 4: Requirements for emissions-related systems."*

http://www.aenor.es/aenor/normas/iso/fichanormaiso.asp?codigo=067245#.V2lxqKJh3_g

[12] *Society of Automotive Engineers "SAE J1850: Class B Data Communication Network Interface."*

https://law.resource.org/pub/us/cfr/ibr/005/sae.j1850.2001.pdf

[13] *CANopen solutions.*

http://www.canopensolutions.com/index.html

[14] *ISO 9141-2 (1994) "Road vehicles -- Diagnostic systems -- Part 2: CARB requirements for interchange of digital information."*

http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=16738

[15] *ISO 14230-4 (2000) "Road vehicles – Diagnostic systems – Keyword Protocol 2000 – Part 4: Requirements for emission-related systems"*

https://law.resource.org/pub/us/cfr/ibr/004/iso.14230-4.2000.pdf

[16] *SAE J1979 (2002) "E/E Diagnostic Test Modes"*

https://law.resource.org/pub/us/cfr/ibr/005/sae.j1979.2002.pdf

[17] *Özen Elektronik.*

https://www.ozenelektronik.com/obd-cozumleri-g.html

[18] *Arduino.*

https://www.arduino.cc/en/Guide/HomePage

GLOSARIO

ISO: International Organization for Standardization

ECU: Engine Control Unit

ECM: Engine Control Module

TCM: Transmission Control Module

BCM: Body Control Unit

OBD: On Board Diagnostic

EOBD: European OBD

CARB: California Air Resources Board

EPA: Environment Protection Agency

MIL: Malfunction Indicator Lamp

DTC: Diagnostic Trouble Code

PID: Parameter Identification

KW: KeyWord

PC: Personal Computer

EGR: Exhaust Gas Recirculation

SAE: Society of Automotive Engineers

DLC: Data Link Connector

ABS: Antiblockier system

DTE: Data Terminal Equipment

DCE: Data Communication Equipment

PDA: Personal Digital Assistant

CAN: Controller Area Network

AT: Comandos Hayes / de atención

CMOS: Complementary Metal-Oxide-Semiconductor

UART: Universal Asyncronous Receiver-Transmitter

OSI: Open System Interconnection

SCP: Secure Copy

CSMA/CR: Carrier Sense Multiple Access/Collision Resolution

CSMA/CA: Carrier Sense Multiple Access/Collision Avoidance

PWM: Pulse Width Modulation

VPM: Variable Pulse Modulation

SOF/EOF: Start/End Of Frame

CRC: Cyclical Redundancy Check

IFR: In Frame Response

NB: Normalization Bit

NRZ: Non Return to Zero

ACK: Acknowledgement

RTR: Remote Transmission Request

LPG: Liquefied Petroleum Gas

CNG: Compressed Natural Gas

CAL: CAN Application Layer

SDS: Smart Distributed System

KWP: KeyWord Protocol

FTDI: Future Technology Devices International

SUMO: Simulation of Urban Mobility

HB/LB: High/Low Byte

PCM: Powertrain Control Module

USB: Universal Serial Bus

AC/DC: Alternating/Direct Current

SRAM: Static Random Access Memory

EEPROM: Erasable Electrically Programmable Read-only Memory

IEEE: Institute of Electrical and Electronics Engineers

ISM: Industrial, Scientific and Medical

SO: Sistema Operativo

BJT: Bipolar Junction Transistor

LED: Light-Emitting Diode

IDE: Integrated Development Environment

HUD: Head Up Display

DIP: Dual In-line Package

ANEXO A. CÓDIGO DEL PROYECTO

```
//CONSTANTES

//INICIALIZACION
const int flanco_bajada = 0;
const int bit_start = 1;
const int up_bits_1 = 2;
const int low_bits_1 = 3;
const int up_bits_2 = 4;
const int low_bits_2 = 5;
const int margen_inf_200 = 195;
const int margen_sup_200 = 205;
const int margen_inf_400 = 395;
const int margen_sup_400 = 405;
const byte SINC = 0x55;
//const byte KW1 = 0x08;
//const byte KW2 = 0x08;
//const byte KW2_NEG = 0xF7;
const byte KW1 = 0x94; //KW para poner P2min a 0 (un solo ECU)
const byte KW2 = 0x94;
const byte KW2_NEG = 0x6B; //KW2_neg de 0x94
const byte DIR_NEG = 0xCC;

//PIDs
const byte PIDs_supported = 0x00; //PIDs soportados del 1 al 20
const byte MILyDTC = 0x01; //Estado del indicador MIL y numero de DTCs
almacenados
const byte coolant = 0x05; //Temperatura del refrigerante
const byte rpm = 0x0C; //Revoluciones por minuto del motor
const byte vehicle_speed = 0x0D; //Velocidad del vehiculo
const byte throttle_position = 0x11; //Posicion del acelerador
const byte OBD_standard = 0x1C; //Posicion del acelerador
const byte total_time = 0x1F; //Tiempo desde el arranque
const byte PIDs_supported_20 = 0x20; //PIDs soportados del 21 al 40
const byte fuel_level = 0x2F;
const byte PIDs_supported_40 = 0x40; //PIDs soportados del 41 al 60
const byte air_temp = 0x46; //Temperatura ambiente
```

```
const byte fuel_type = 0x51; //Posicion del acelerador

//MODOS DE OPERACION
const byte mode1 = 0x01;
const byte mode3 = 0x03;
const byte mode4 = 0x04;

//ESTADOS
const int INICIALIZACION_1 = 0;
const int INICIALIZACION_2 = 1;
const int PETICION_1 = 2;
const int PETICION_2 = 3;
const int RESPUESTA = 4;

//PINES
const int obdpin = 15; //Posterior RX3 de la UART (Para INI a 5 baud)
const int LED_ready = 52;
const int LED_MIL = 50;
const char throttle_pin = A0;
const char speed_pin = A1;
const char rpm_pin = A2;
const char fuel_pin = A3;
const char temp_pin = A4;
const char coolant_pin = A5;
const char sw1 = 38;
const char sw2 = 36;
const char sw3 = 34;
const char sw4 = 32;
const char sw5 = 30;
const char sw6 = 28;
const char sw7 = 26;
const char sw8 = 24;

//VARIABLES

float A;
int valor = 0;
boolean ini5baud = false;
boolean ini = false;
boolean iter = false;
int num_DTC = 0;
boolean MIL = false;
```

```
int estado = 0;
int estado_ini_5baud = 0;
boolean estado_ini_rapida = false;
long tiempo_parado = 0;
long previousMillis = 0;
long principio_W4 = 0;
byte trama_rx[11];
byte trama_tx[11];
byte pos_H[4];
byte pos_L[4];
byte sum = 0x00;
int length_rx = 0;
int length_tx = 0;
byte mode = 0x00;
byte PID = 0x00;
byte byte_aux = 0x00;
boolean manual_mode = false;
int manual_flag = 0;
float throttle_manual = 0;
float speed_manual = 0;
float rpm_manual = 0;
float fuel_manual = 0;
float temp_manual = 0;
float coolant_manual = 0;


void setup() {
    Serial.begin(9600);
    Serial3.begin(10400);
    pinMode(obdpin, INPUT);
    pinMode(LED_ready, OUTPUT);
    pinMode(LED_MIL, OUTPUT);
    pinMode(sw1, INPUT);
    pinMode(sw2, INPUT);
    pinMode(sw3, INPUT);
    pinMode(sw4, INPUT);
    pinMode(sw5, INPUT);
    pinMode(sw6, INPUT);
    pinMode(sw7, INPUT);
    pinMode(sw8, INPUT);
}
```

```
//Funcion que va testeando los flancos de la señal en la linea K del protocolo ISO9141-2 a 5Baud/s para comprobar que recibimos 0x33
int fun_ini5baud(){
  unsigned long currentMillis = millis();
  valor = digitalRead(obdpin);  //lectura digital de pin
  switch (estado_ini_5baud) {

    //Estado que comprueba si la línea pasa a nivel bajo
    case flanco_bajada:
      if (valor==LOW){
        estado_ini_5baud = bit_start;
        previousMillis = currentMillis; //Se guarda el instante de inicio de estado 1
      }
    break;

    //Estado que comprueba bit de start a 0, dura 200ms (ponemos un margen de 5ms para evitar errores
    case bit_start:
      if ((valor==HIGH) && (currentMillis-previousMillis > margen_inf_200) && (currentMillis-previousMillis < margen_sup_200)) {
        estado_ini_5baud = up_bits_1;
        previousMillis = currentMillis;
      }    else    if((valor==HIGH)    &&    (currentMillis-previousMillis    < margen_inf_200)) {
        estado_ini_5baud = flanco_bajada;
      }else{
        estado_ini_5baud = bit_start;
      }
    break;

    //Estado que comprueba 2 bits a nivel alto, dura 400ms (ponemos margen de 5ms para evitar errores)
    case up_bits_1:
      if ((valor==LOW) && (currentMillis-previousMillis > margen_inf_400) && (currentMillis-previousMillis < margen_sup_400)) {
        estado_ini_5baud = low_bits_1;
        previousMillis = currentMillis;
      } else if((valor==LOW) && (currentMillis-previousMillis < margen_inf_400)) {
        estado_ini_5baud = flanco_bajada;
      }else{
        estado_ini_5baud = up_bits_1;
```

```
      }
    break;


    //Estado que comprueba 2 bits a nivel bajo, dura 400ms (ponemos margen de
5ms para evitar errores)
    case low_bits_1:
      if ((valor==HIGH) && (currentMillis-previousMillis > margen_inf_400) &&
(currentMillis-previousMillis < margen_sup_400)) {
        estado_ini_5baud = up_bits_2;
        previousMillis = currentMillis;
      }    else    if((valor==HIGH)    &&    (currentMillis-previousMillis    <
margen_inf_400)) {
        estado_ini_5baud = flanco_bajada;
      }else{
        estado_ini_5baud = low_bits_1;
      }
    break;


    //Estado que comprueba 2 bits a nivel alto, dura 400ms (ponemos margen de
5ms para evitar errores)
    case up_bits_2:
      if ((valor==LOW) && (currentMillis-previousMillis > margen_inf_400) &&
(currentMillis-previousMillis < margen_sup_400)) {
        estado_ini_5baud = low_bits_2;
        previousMillis = currentMillis;
      } else if((valor==LOW) && (currentMillis-previousMillis < margen_inf_400))
{
        estado_ini_5baud = flanco_bajada;
      }else{
        estado_ini_5baud = up_bits_2;
      }
    break;


    //Estado que comprueba 2 bits a nivel bajo, dura 400ms (ponemos margen de
5ms para evitar errores)
    case low_bits_2:
        if ((valor==HIGH) && (currentMillis-previousMillis > margen_inf_400) &&
(currentMillis-previousMillis < margen_sup_400)) {
          ini5baud=true;
          delay(200);
        }    else    if((valor==HIGH)    &&    (currentMillis-previousMillis    <
margen_inf_400)) {
          estado_ini_5baud = flanco_bajada;
        }else{
```

```
          estado_ini_5baud = low_bits_2;
        }
    break;
  }
  return ini5baud;
}


//Función que recibe señal INI a 5Baud/s y manda señales de sincronización y keywords correspondientes al protocolo ISO9141-2
int fun_ini() {
    unsigned long final_W4 = millis();

    switch(estado_ini_rapida){

      case false:
        delay(20); //W1 [20-300ms]
        Serial3.write(SINC); //Sincronización a 10.4 kbps (0x55)
        delay(5); //W2 [5-20ms]
        Serial3.write(KW1); //KeyWord 1
        delay(5); //W3 [0-20ms]
        Serial3.write(KW2); //Al enviar el segundo KeyWord estamos asignando P2min=25ms;
        principio_W4=final_W4;
        estado_ini_rapida=true;
      break;

      case true:
        //Comprueba la señal recibida de KW2 invertida y manda señal de INI invertida (después acaba la inicialización)
        if (Serial3.available() > 0){
          byte_aux = Serial3.read();
          if(byte_aux==KW2_NEG){
            delay(25);
            Serial3.write(DIR_NEG); //0x33 (INI) invertido = 0xCC
            ini=true; //Listo para comunicarse
            estado_ini_rapida=false;
          }else{
            //si pasan 5 seg de inactividad o cree que es el protocolo KWP2000
            if((final_W4 - principio_W4 > 5000)||(byte_aux==0x16)){
              estado=INICIALIZACION_1;
            }
            ini=false;
```

```
          }
        }
      break;
    }
    return ini;
  }

void manda_resp(){
   trama_tx[0]=trama_rx[0]-0x20;
   trama_tx[1]=0x6B;
   trama_tx[2]=0xD1; //dirección ECU 1
   trama_tx[3]=mode  + 0x40;
   //Hacemos el checksum de la trama a enviar
   for (int cont=0; cont<length_tx-1; cont++){
     sum=sum+trama_tx[cont];
   }
   trama_tx[length_tx-1]=sum;
   delay(25);//Esperamos tiempo P2
   //Enviamos trama con tiempo P1=0s;
   for (int cont2=0; cont2<length_tx; cont2++){
     Serial3.write(trama_tx[cont2]);
   }
}

void fun_mode1(){
  PID=trama_rx[4];
  if (manual_mode==false){
    Serial.println("PID=");
    Serial.println(trama_rx[4],HEX);
  }
  trama_tx[4]=PID;
  //Marcamos la longitud del mensaje a enviar para el cálculo del checksum
  length_tx=7; //Por defecto para un solo byte de dato
  switch(PID){

    case PIDs_supported:
      Serial.println("PIDs soportados [1-20]\n");
      length_tx=10;
      trama_tx[5]=0x88;  //Con  estado  MIL  y  num.  DTCs  y  temperatura  del
refrigerante
      trama_tx[6]=0x18; //Con rpm y velocidad
```

```
      trama_tx[7]=0x80; //Con posicion acelerador
      trama_tx[8]=0x13;   //Con   estándar   OBD,   tiempo   desde   arranque   y
PIDs_supported_40
      manda_resp();
    break;

    case MILyDTC:
      Serial.println("Estado MIL y num. DTCs\n");
      length_tx=10;
      if (MIL==true){
        trama_tx[5]=0x80 + num_DTC; //MIL ON y número de DTCs
      }else{
        trama_tx[5]=0x00 + num_DTC;
      }
      trama_tx[6]=0x00;
      trama_tx[7]=0x00;
      trama_tx[8]=0x00;
      manda_resp();
    break;

    case coolant:
      if (manual_mode==true){
        trama_tx[5]=coolant_manual+40;
      }else{
        Serial.println("Temperatura del refrigerante\n");
        A=analogRead(coolant_pin);
        A=A*(200.0/1023.0);
        trama_tx[5]=A+40;
      }
      manda_resp();
    break;

    case rpm:
      length_tx=8;
      if (manual_mode==true){
        trama_tx[5]=rpm_manual*4/256;
        trama_tx[6]=rpm_manual-(A*4/256);
      }else{
        Serial.println("Revoluciones por minuto\n");
        A=analogRead(rpm_pin);
        A=A*(8000.0/1023.0);
        trama_tx[5]=A*4/256;
```

```
    trama_tx[6]=A-(A*4/256);
  }
  manda_resp();
break;

case vehicle_speed:
  if (manual_mode==true){
    trama_tx[5]=speed_manual;
  }else{
    Serial.println("Velocidad del vehiculo\n");
    A=analogRead(speed_pin);
    trama_tx[5]=A*(255.0/1023.0);
  }
  manda_resp();
break;

case throttle_position:
  if (manual_mode==true){
    trama_tx[5]=throttle_manual*(255.0/100.0);
  }else{
    Serial.println("Posicion del acelerador\n");
    A=analogRead(throttle_pin);
    trama_tx[5]=A*(255.0/1023.0);
  }
  manda_resp();
break;

case OBD_standard:
  Serial.println("Standard OBD\n");
  trama_tx[5]=0x09; //Soporta OBD, OBD2 y EOBD
  manda_resp();
break;

case total_time:
  Serial.println("Tiempo desde el arranque\n");
  length_tx=8;
  A=millis()/1000; //Pasamos de milisegundos a segundos
  trama_rx[5]=A/256;
  trama_rx[6]=A-A/256;
  manda_resp();
break;
```

```
case PIDs_supported_20:
  Serial.println("PIDs soportados [21-40]\n");
  length_tx=10;
  trama_tx[5]=0x00;
  trama_tx[6]=0x02; //Con nivel combustible
  trama_tx[7]=0x00;
  trama_tx[8]=0x01; //Con PIDs_supported_40
  manda_resp();
break;

case fuel_level:
  if (manual_mode==true){
    trama_tx[5]=fuel_manual*(255.0/100.0);
  }else{
    Serial.println("Nivel de combustible\n");
    A=analogRead(fuel_pin);
    trama_tx[5]=A*(255.0/1023.0);
  }
  manda_resp();
break;

case PIDs_supported_40:
  Serial.println("PIDs soportados [41-60]\n");
  length_tx=10;
  trama_tx[5]=0x04; //Con temperatura ambiente
  trama_tx[6]=0x00;
  trama_tx[7]=0x80; //Con tipo de combustible
  trama_tx[8]=0x00;
  manda_resp();
break;

case air_temp:
  if (manual_mode==true){
    trama_tx[5]=temp_manual+40;
  }else{
    Serial.println("Temperatura ambiente\n");
    A=analogRead(temp_pin);
    A=A*(40.0/1023.0);
    trama_tx[5]=A+40;
  }
  manda_resp();
```

```
    break;

    case fuel_type:
      Serial.println("Tipo de combustible\n");
      trama_tx[5]=0x04; //Combustible diesel
      manda_resp();
    break;
  }
}

void fun_mode3(){
  //Por simplicidad se contemplarán 4 DTCs a la vez como máximo
  Serial.println("Numero de DTCs: ");
  Serial.println(num_DTC);
  if (num_DTC==0){
    MIL=false;
  }else if (num_DTC < 3){
    trama_tx[4]=num_DTC;
    trama_tx[5]=pos_H[0];
    trama_tx[6]=pos_L[0];
    if (num_DTC==2){
      trama_tx[7]=pos_H[1];
      trama_tx[8]=pos_L[1];
      manda_resp();
    }else{
      trama_tx[7]=0x00;
      trama_tx[8]=0x00;
      manda_resp();
    }
  }else{
    if (iter==false){
      trama_tx[4]=num_DTC;
      trama_tx[5]=pos_H[0];
      trama_tx[6]=pos_L[0];
      trama_tx[7]=pos_H[1];
      trama_tx[8]=pos_L[1];
      manda_resp();
      iter=true;
    }else{
      trama_tx[4]=num_DTC;
      trama_tx[5]=pos_H[2];
```

```
      trama_tx[6]=pos_L[2];
      if (num_DTC==4){
        trama_tx[7]=pos_H[3];
        trama_tx[8]=pos_L[3];
        manda_resp();
        iter=false;
      }else{
        trama_tx[7]=0x00;
        trama_tx[8]=0x00;
        manda_resp();
        iter=false;
      }
    }
  }
}

void comprueba_switch(){
  int pos=0;
  if (digitalRead(sw1) == 1){
    manual_mode=true;
    comprueba_manual();
  }else{  //Si  se  desactiva  se  resetean  las  banderas  de  los  PIDs  usados
manualmente
    manual_mode=false;
    manual_flag=0;
    throttle_manual=0;
    speed_manual=0;
    rpm_manual=0;
    fuel_manual=0;
    temp_manual=0;
    coolant_manual=0;
  }
  if (digitalRead(sw2) == 1){
    pos_H[pos]=0x03;
    pos_L[pos]=0x01;
    pos=pos+1;
  }
  if (digitalRead(sw3) == 1){
    pos_H[pos]=0x03;
    pos_L[pos]=0x40;
    pos=pos+1;
  }
```

```
  if (digitalRead(sw4) == 1){
    pos_H[pos]=0x02;
    pos_L[pos]=0x17;
    MIL=true;
    digitalWrite(LED_MIL,HIGH);
    pos=pos+1;
  }
  if (digitalRead(sw5) == 1){
    pos_H[pos]=0x01;
    pos_L[pos]=0x71;
    pos=pos+1;
  }
  if (digitalRead(sw6) == 1){
    pos_H[pos]=0x05;
    pos_L[pos]=0x00;
    pos=pos+1;
  }
  if (digitalRead(sw7) == 1){
    pos_H[pos]=0x04;
    pos_L[pos]=0x20;
    MIL=true;
    digitalWrite(LED_MIL,HIGH);
    pos=pos+1;
  }
  if (digitalRead(sw8) == 1){
    pos_H[pos]=0x00;
    pos_L[pos]=0x68;
    MIL=true;
    digitalWrite(LED_MIL,HIGH);
    pos=pos+1;
  }
  num_DTC=pos;
}

 void comprueba_manual(){
   char c = ' ';
   String mensaje = "";
   String mensaje2 = "";
   if (Serial.available()){
     while((c != '=')) //Leemos hasta el igual
     {
```

```
      mensaje = mensaje + c;
      c = Serial.read();
      delay(25);
    }
    mensaje.trim(); //Elimina los espacios en blanco
    mensaje.toLowerCase(); //Pasa a minusculas
    if (mensaje == "throttle"){manual_flag=1;}
    else if (mensaje == "speed"){manual_flag=2;}
    else if (mensaje == "rpm"){manual_flag=3;}
    else if (mensaje == "fuel"){manual_flag=4;}
    else if (mensaje == "temp"){manual_flag=5;}
    else if (mensaje == "coolant"){manual_flag=6;}
    else{
      manual_flag=0;
    }
    mensaje = ""; //Reseteo la cadena de caracteres
  }
  if ((Serial.available())&&(c=='=')){
    c = Serial.read(); //Elimino el '='
    delay(25);
   while((c=='0')||(c=='1')||(c=='2')||(c=='3')||(c=='4')||(c=='5')||(c=='6')||
(c=='7')||(c=='8')||(c=='9')||(c==' ')||(c=='-')) //Leemos la cantidad
    {
      mensaje2 = mensaje2 + c;
      c = Serial.read();
      delay(25);
    }
    mensaje.trim(); //Elimina los espacios en blanco
    switch(manual_flag){ //Asignamos el valor en tipo entero para su cambio en
fun_mode1

      case 1:
        if ((mensaje2.toInt()<0)||(mensaje2.toInt()>100)){
          Serial.println("Throttle range = [0 - 100]");
        }else{
          throttle_manual=mensaje2.toInt();
        }
      break;

      case 2:
        if ((mensaje2.toInt()<0)||(mensaje2.toInt()>255)){
          Serial.println("Speed range = [0 - 255]");
```

```
    }else{
      speed_manual=mensaje2.toInt();
    }
  break;

  case 3:
    if ((mensaje2.toInt()<0)||(mensaje2.toInt()>16000)){
      Serial.println("Rpm range = [0 - 16000]");
    }else{
      rpm_manual=mensaje2.toInt();
    }
  break;

  case 4:
    if ((mensaje2.toInt()<0)||(mensaje2.toInt()>100)){
      Serial.println("Fuel range = [0 - 100]");
    }else{
      fuel_manual=mensaje2.toInt();
    }
  break;

  case 5:
    if ((mensaje2.toInt()<-40)||(mensaje2.toInt()>215)){
      Serial.println("Temp range = [-40 - 215]");
    }else{
      temp_manual=mensaje2.toInt();
    }
  break;

  case 6:
    if ((mensaje2.toInt()<-40)||(mensaje2.toInt()>215)){
      Serial.println("Coolant range = [-40 - 215]");
    }else{
      coolant_manual=mensaje2.toInt();
    }
  break;

  case 0:
    Serial.println("PID erroneo o no soportado en modo manual");
  break;
}
```

```
     mensaje2 = "";
    }
  }

void loop(){
  comprueba_switch();
  unsigned long tiempo_actual = millis();
  switch (estado){
    //Estado de la fase de inicialización del protocolo
    case INICIALIZACION_1:
      digitalWrite(LED_ready,LOW);
      ini5baud=fun_ini5baud();
      if (ini5baud==true){
        estado=INICIALIZACION_2;
        estado_ini_5baud=0;
        ini5baud=false;
      }
    break;

    case INICIALIZACION_2:
      ini=fun_ini();
      if (ini==true) {
        digitalWrite(LED_ready,HIGH); //Se enciende el LED verde cuando se esta preparado para la comunicacion
        estado=PETICION_1;
        length_rx=0;
        tiempo_parado=tiempo_actual;
        Serial3.flush();
        ini=false;
      }
    break;

    //Estado que espera hasta que le llegue el principio de la trama de peticion y lo guarda
    case PETICION_1:
      if(Serial3.available()>0){
        byte_aux=Serial3.read();
        if(byte_aux==0x68){
          trama_rx[length_rx]=byte_aux;
          length_rx = length_rx + 1;
          estado=PETICION_2;
        } else if (tiempo_actual - tiempo_parado > 5000){ //Si hay dato, no es el principio de trama y han pasado mas de 5 segundos
```

```
          estado=INICIALIZACION_1;
        }
      } else if (tiempo_actual - tiempo_parado > 10000){ //Si han pasado 10 segundos de inactividad
          estado=INICIALIZACION_1;
          ini5baud=false;
          ini=false;
          estado_ini_5baud=0;
          estado_ini_rapida=false;
      }
    break;

    //Estado que recibe y guarda la peticion del tester hasta el checksum
    case PETICION_2:
        if(length_rx<12){ //Si cumple la longitud maxima de trama
          if(Serial3.available()>0){
            byte_aux=Serial3.read();
            trama_rx[length_rx]=byte_aux;
            length_rx=length_rx+1;
            if(length_rx>4){  //Cuando ya tenemos el primer byte de dato empezamos a comprobar el checksum
              for (int cont3=0; cont3<length_rx-1; cont3++){
                sum=sum+trama_rx[cont3];
              }
              //Si hemos llegado al checksum y es correcto paso a responder a la peticion
              if(sum==trama_rx[length_rx-1]){
                estado=RESPUESTA;
                sum=0x00;
              }else{
                sum=0x00;
              }
            }
          }else if (tiempo_actual - tiempo_parado > 5000){ //Si no hay dato y han pasado más de 5 segundos
            estado=INICIALIZACION_1;
          }
        }else{ //En el caso que se corrompan los datos recibidos
          estado=PETICION_1;
          tiempo_parado=tiempo_actual;
          length_rx=0;
          sum=0x00;
```

```
        }
    break;

    //Estado que comprueba que la trama es correcta y actúa según el modo requerido
    case RESPUESTA:
      mode=trama_rx[3];
      if(manual_mode==false){
        Serial.println("MODO=");
        Serial.println(trama_rx[3],HEX);
      }
      switch(mode){

        //Funcion que procesa peticiones de modo 1
        case mode1:
          fun_mode1();
          estado=PETICION_1;  //Despues de interpretar la peticion vuelvo a esperar otra
          tiempo_parado=tiempo_actual; //Guardo el momento desde que empiezo a escuchar si hay trama de peticion
        break;

        //Funcion que procesa peticiones de modo 3
        case mode3:
          fun_mode3();
          estado=PETICION_1;
          length_rx=0;
          tiempo_parado=tiempo_actual;
        break;

        //Funcion que procesa peticiones de modo 4
        case mode4:
          MIL=false;
          num_DTC=0;
          digitalWrite(LED_MIL,LOW);
          estado=PETICION_1;
          length_rx=0;
          tiempo_parado=tiempo_actual;
        break;

        //Si nos pide algun modo no implementado vuelvo a esperar trama de peticion
        default:
```

```
          estado=PETICION_1;
          length_rx=0;
          tiempo_parado=tiempo_actual;
        break;


      }
      break;
  }
    }
```

ANEXO B. LISTA DE PARÁMETROS PID

PID	Bytes	Descripción	Min.	Max.	Uds.	Fórmula
00	4	PIDs supported [01 - 20]				Bit encoded [A7..D0] == [PID $01..PID $20]
01	4	Monitor status since DTCs cleared. (Includes malfunction indicator lamp (MIL) status and number of DTCs.)				Bit encoded.
02	2	Freeze DTC				
03	2	Fuel system status				Bit encoded.
04	1	Calculated engine load	0	100	%	$\frac{100}{255}A$
05	1	Engine coolant temperature	-40	215	°C	A – 40
06	1	Short term fuel trim—Bank 1	-100	99.2	%	$\frac{100}{128}A-100$
07	1	Long term fuel trim—Bank 1				
08	1	Short term fuel trim—Bank 2				
09	1	Long term fuel trim—Bank 2				
0A	1	Fuel pressure	0	765	kPa	3 A
0B	1	Intake manifold absolute pressure	0	255	kPa	A
0C	2	Engine RPM	0	16,383.75	rpm	$\frac{256A+B}{4}$
0D	1	Vehicle speed	0	255	km/h	A
0E	1	Timing advance	-64	63.5	°	$\frac{A}{2}-64$
0F	1	Intake air temperature	-40	215	°C	A – 40
10	2	MAF air flow rate	0	655.35	grams/sec	$\frac{256A+B}{100}$
11	1	Throttle position	0	100	%	$\frac{100}{255}A$
12	1	Commanded secondary air status				Bit encoded.
13	1	Oxygen sensors present (in 2 banks)				[A0..A3] == Bank 1, Sensors 1-4. [A4..A7] == Bank 2...
14	2	Oxygen Sensor 1 A: Voltage B: Short term fuel trim	0 -100	1.275 99.2	Volts %	$\frac{A}{200}$ $\frac{100}{128}B-100$ (if B==$FF, sensor is not used in trim calculation)
15	2	Oxygen Sensor 2 A: Voltage B: Short term fuel trim				
16	2	Oxygen Sensor 3 A: Voltage B: Short term fuel trim				
17	2	Oxygen Sensor 4 A: Voltage B: Short term fuel trim				
18	2	Oxygen Sensor 5 A: Voltage				

		B: Short term fuel trim				
19	2	Oxygen Sensor 6 A: Voltage B: Short term fuel trim				
1A	2	Oxygen Sensor 7 A: Voltage B: Short term fuel trim				
1B	2	Oxygen Sensor 8 A: Voltage B: Short term fuel trim				
1C	1	OBD standards this vehicle conforms to				Bit encoded.
1D	1	Oxygen sensors present (in 4 banks)				Similar to PID 13, but [A0..A7] == [B1S1, B1S2, B2S1, B2S2, B3S1, B3S2, B4S1, B4S2]
1E	1	Auxiliary input status				A0 == Power Take Off (PTO) status (1 == active) [A1..A7] not used
1F	2	Run time since engine start	0	65,535	seconds	256A + B
20	4	PIDs supported [21 - 40]				Bit encoded [A7..D0] == [PID $21..PID $40]
21	2	Distance traveled with malfunction indicator lamp (MIL) on	0	65,535	km	256A + B
22	2	Fuel Rail Pressure (relative to manifold vacuum)	0	5177.265	kPa	0.079(256A + B)
23	2	Fuel Rail Gauge Pressure (diesel, or gasoline direct injection)	0	655,35	kPa	10(256A + B)
24	4	Oxygen Sensor 1 AB: Fuel–Air Equivalence Ratio CD: Voltage				
25	4	Oxygen Sensor 2 AB: Fuel–Air Equivalence Ratio CD: Voltage				
26	4	Oxygen Sensor 3 AB: Fuel–Air Equivalence Ratio CD: Voltage				
27	4	Oxygen Sensor 4 AB: Fuel–Air Equivalence Ratio CD: Voltage	0 0	< 2 < 8	ratio V	$\frac{2}{65536}(256A + B)$ $\frac{8}{65536}(256C + D)$
28	4	Oxygen Sensor 5 AB: Fuel–Air Equivalence Ratio CD: Voltage				
29	4	Oxygen Sensor 6 AB: Fuel–Air Equivalence Ratio CD: Voltage				
2A	4	Oxygen Sensor 7				

		AB: Fuel–Air Equivalence Ratio CD: Voltage				
2B	4	Oxygen Sensor 8 AB: Fuel–Air Equivalence Ratio CD: Voltage				
2C	1	Commanded EGR	0	100	%	$\frac{100}{255}A$
2D	1	EGR Error	-100	99.2	%	$\frac{100}{128}A - 100$
2E	1	Commanded evaporative purge	0	100	%	$\frac{100}{255}A$
2F	1	Fuel Tank Level Input	0	100	%	$\frac{100}{255}A$
30	1	Warm-ups since codes cleared	0	255	count	A
31	2	Distance traveled since codes cleared	0	65,535	km	256A + B
32	2	Evap. System Vapor Pressure	-8,192	8191.75	Pa	$\frac{256A + B}{4}$
33	1	Absolute Barometric Pressure	0	255	kPa	A
34	4	Oxygen Sensor 1 AB: Fuel–Air Equivalence Ratio CD: Current	0 -128	< 2 < 128	ratio mA	$\frac{2}{65536}(256A + B)$ $\frac{256C + D}{256} - 128$ or $C + \frac{D}{256} - 128$
35	4	Oxygen Sensor 2 AB: Fuel–Air Equivalence Ratio CD: Current				
36	4	Oxygen Sensor 3 AB: Fuel–Air Equivalence Ratio CD: Current				
37	4	Oxygen Sensor 4 AB: Fuel–Air Equivalence Ratio CD: Current				
38	4	Oxygen Sensor 5 AB: Fuel–Air Equivalence Ratio CD: Current				
39	4	Oxygen Sensor 6 AB: Fuel–Air Equivalence Ratio CD: Current				
3A	4	Oxygen Sensor 7 AB: Fuel–Air Equivalence Ratio CD: Current				
3B	4	Oxygen Sensor 8 AB: Fuel–Air Equivalence Ratio				

		CD: Current				
3C	2	Catalyst Temperature: Bank 1, Sensor 1	-40	6,513.5	°C	$\frac{256A+B}{10}-40$
3D	2	Catalyst Temperature: Bank 2, Sensor 1				
3E	2	Catalyst Temperature: Bank 1, Sensor 2				
3F	2	Catalyst Temperature: Bank 2, Sensor 2				
40	4	PIDs supported [41 - 60]				Bit encoded [A7..D0] == [PID $41..PID $60]
41	4	Monitor status this drive cycle				Bit encoded.
42	2	Control module voltage	0	65.535	V	$\frac{256A+B}{1000}$
43	2	Absolute load value	0	25,700	%	$\frac{100}{255}(256A+B)$
44	2	Fuel–Air commanded equivalence ratio	0	< 2	ratio	$\frac{2}{65536}(256A+B)$
45	1	Relative throttle position	0	100	%	$\frac{100}{255}A$
46	1	Ambient air temperature	-40	215	°C	A − 40
47	1	Absolute throttle position B	0	100	%	$\frac{100}{255}A$
48	1	Absolute throttle position C				
49	1	Accelerator pedal position D				
4A	1	Accelerator pedal position E				
4B	1	Accelerator pedal position F				
4C	1	Commanded throttle actuator				
4D	2	Time run with MIL on	0	65,535	minutes	256A + B
4E	2	Time since trouble codes cleared				
4F	4	Maximum value for Fuel–Air equivalence ratio, oxygen sensor voltage, oxygen sensor current, and intake manifold absolute pressure	0, 0, 0, 0	255, 255, 255, 2550	ratio, V, mA, kPa	A, B, C, D*10
50	4	Maximum value for air flow rate from mass air flow sensor	0	2550	g/s	A*10, B, C, and D are reserved for future use
51	1	Fuel Type				From fuel type table
52	1	Ethanol fuel %	0	100	%	$\frac{100}{255}A$
53	2	Absolute Evap system Vapor Pressure	0	327.675	kPa	$\frac{256A+B}{200}$
54	2	Evap system vapor pressure	-32,767	32,768	Pa	((A*256)+B)-32767
55	2	Short term secondary oxygen sensor trim, A: bank 1, B: bank 3	-100	99.2	%	$\frac{100}{128}A-100$
56	2	Long term secondary oxygen sensor trim, A: bank 1, B: bank 3				$\frac{100}{128}B-100$
57	2	Short term secondary oxygen sensor trim, A: bank 2, B:				

		bank 4				
58	2	Long term secondary oxygen sensor trim, A: bank 2, B: bank 4				
59	2	Fuel rail absolute pressure	0	655,350	kPa	10 (256A + B)
5A	1	Relative accelerator pedal position	0	100	%	$\frac{100}{255}A$
5B	1	Hybrid battery pack remaining life	0	100	%	$\frac{100}{255}A$
5C	1	Engine oil temperature	-40	210	°C	A − 40
5D	2	Fuel injection timing	-210.00	301.992	°	$\frac{256A+B}{128} - 210$
5E	2	Engine fuel rate	0	3276.75	L/h	$\frac{256A+B}{20}$
5F	1	Emission requirements to which vehicle is designed				Bit Encoded
60	4	PIDs supported [61 - 80]				Bit encoded [A7..D0] == [PID $61..PID $80]
61	1	Driver's demand engine - percent torque	-125	125	%	A-125
62	1	Actual engine - percent torque	-125	125	%	A-125
63	2	Engine reference torque	0	65,535	Nm	256 A + B
64	5	Engine percent torque data	-125	125	%	A-125 Idle B-125 Engine point 1 C-125 Engine point 2 D-125 Engine point 3 E-125 Engine point 4
65	2	Auxiliary input / output supported				Bit Encoded
66	5	Mass air flow sensor				
67	3	Engine coolant temperature				
68	7	Intake air temperature sensor				
69	7	Commanded EGR and EGR Error				
6A	5	Commanded Diesel intake air flow control and relative intake air flow position				
6B	5	Exhaust gas recirculation temperature				
6C	5	Commanded throttle actuator control and relative throttle position				
6D	6	Fuel pressure control system				
6E	5	Injection pressure control system				
6F	3	Turbocharger compressor inlet pressure				
70	9	Boost pressure control				
71	5	Variable Geometry turbo (VGT) control				
72	5	Wastegate control				
73	5	Exhaust pressure				

74	5	Turbocharger RPM				
75	7	Turbocharger temperature				
76	7	Turbocharger temperature				
77	5	Charge air cooler temperature (CACT)				
78	9	Exhaust Gas temperature (EGT) Bank 1				Special PID.
79	9	Exhaust Gas temperature (EGT) Bank 2				Special PID.
7A	7	Diesel particulate filter (DPF)				
7B	7	Diesel particulate filter (DPF)				
7C	9	Diesel Particulate filter (DPF) temperature				
7D	1	NOx NTE control area status				
7E	1	PM NTE control area status				
7F	13	Engine run time				
80	4	PIDs supported [81 - A0]				Bit encoded [A7..D0] == [PID $81..PID $A0]
81	21	Engine run time for Auxiliary Emissions Control Device(AECD)				
82	21	Engine run time for Auxiliary Emissions Control Device(AECD)				
83	5	NOx sensor				
84		Manifold surface temperature				
85		NOx reagent system				
86		Particulate matter (PM) sensor				
87		Intake manifold absolute pressure				
A0	4	PIDs supported [A1 - C0]				Bit encoded [A7..D0] == [PID $A1..PID $C0]
C0	4	PIDs supported [C1 - E0]				Bit encoded [A7..D0] == [PID $C1..PID $E0]
C3						Returns numerous data, including Drive Condition ID and Engine Speed
C4						B5 is Engine Idle Request B6 is Engine Stop Request

Tabla 26 – Lista de parámetros PID