INSTITUCIÓN UNIVERSITARIA
PASCUAL BRAVO®

SISTEMA DE MONITOREO Y DIAGNÓSTICO EN TIEMPO REAL PARA MOTORES DE COMBUSTIÓN INTERNA

Trabajo de grado para optar al título de:
Tecnólogo en Mecánica Automotriz

Asesor:
PhD. Frank Alexander Ruiz Holguín

Brayann Flórez Flórez
Emmanuel Gallego Sánchez

SISTEMA DE MONITOREO Y DIAGNÓSTICO EN TIEMPO REAL PARA MOTORES DE COMBUSTIÓN INTERNA

Trabajo de grado para optar al título de:
Tecnólogo en Mecánica Automotriz

Brayann Flórez Flórez
brayann.florez157@pascualbravo.edu.co

Emnanuel Gallego Sánchez
e.gallego2105@pasualbravo.edu.co

Asesor:
Frank Alexander Ruiz
Holguín PhD. Ingeniero
Electrónico
Docente Facultad de Ingeniería
Departamento de Mecánica

Institución Universitaria Pascual Bravo
Departamento Mecánica
Tecnología en Mecánica Automotriz
Diciembre de 2025

DEDICATORIA

Con este proyecto queremos dedicarlo a nuestras familias y a todas las personas que en mucha o menos medida nos dieron el impulso para llegar a este momento, que se tomaron el tiempo de enseñarnos con paciencia y dedicación para transmitirnos tanto conocimiento.

En especial queremos dedicar este proyecto al profesor Frank Alexander Ruíz, quien nunca se rindió y nos dio todo de sí mismo para que todo fuera posible a pesar incluso de su condición de salud actual, profesor muchas gracias.

Agradecimiento al semillero SIV-2R (semillero de investigación de vehículos de dos ruedas) por su valiosa contribución en la formación técnica del proyecto.

TABLA DE CONTENIDO

Lista de figuras

Lista de tablas

INTRODUCCIÓN

Desde hace mucho tiempo, el ser humano ha implementado los motores de combustión interna para propulsar los vehículos en los cuales se transporta. Esto es algo completamente cotidiano a nivel nacional y, verdaderamente, ha influido en el cambio continuo de la sociedad hasta nuestros días. Pese al desarrollo de nuevas tecnologías, como los motores híbridos o eléctricos, aún dependemos de los motores de combustión interna ya sea por los costos de diseño, que suelen ser más económicos o por los costos de fabricación, entre otros factores según el siguiente enunciado, “Desde hace por lo menos un año, las ventas mundiales de autos eléctricos han empezado a ralentizarse. La principal razón que explica el fenómeno es que estos vehículos son entre un 30 y un 40% más costosos en comparación de uno similar con motor de combustión interna" (Zorrero, 2024).
Lo que se debe resaltar, es que, aunque la tecnología del motor de combustión interna en los autos comunes es ampliamente utilizada, para la mayoría de los usuarios o, por decirlo así, para el público en general, no es común tener conocimientos sobre el funcionamiento de un motor. A partir de esto, surge un interrogante: ¿se puede crear un elemento o sistema que permita el análisis en tiempo real de los datos más cruciales del motor de combustión interna de un vehículo, y que a su vez sea accesible y fácil de usar?

Basándose en esta pregunta, surge la idea para este proyecto en el cual se busca desarrollar un sistema que permita a todos los usuarios de vehículos con motor de combustión interna, realizar un análisis preciso y constante del funcionamiento del motor en tiempo real. De esta manera, se podrá identificar con mayor certeza cuál podría ser la falla que presenta el motor o qué tipo de mantenimiento puede requerir. Esto generaría un menor gasto al momento de efectuar una reparación, y siguiendo esta línea, brindaría mayor seguridad a los ocupantes y tranquilidad al usuario. Como se mencionaba anteriormente, con un proceso de escaneo regular se tiene la posibilidad de identificar los valores que cambian en el motor de un periodo de diagnóstico a otro y determinar si es necesaria una intervención por parte del usuario, ya sea para un mantenimiento preventivo o correctivo.

JUSTIFICACIÓN

El desarrollo de un sistema de medición automotriz que permita un diagnóstico preciso y que a su vez sea accesible para todo público, que le permita obtener al usuario los datos del motor adecuados para evaluar su operación. En ese contexto queremos que cualquier persona profesional, técnica o entusiasta que sea usuario de un vehículo con este tipo de motor pueda realizar un diagnóstico sencillo. Para esas personas el proyecto se vuelve importante, ya que permite que el conocimiento de la electrónica, la programación y la mecánica de automóviles, además de la innovación tecnológica y la autonomía en el desarrollo de soluciones locales sea posible y se expanda. Facilitando el monitoreo real de variables importantes del motor, contribuyendo a un diagnóstico más preciso, pronósticos de mantenimiento y optimización del rendimiento del vehículo. Finalmente, esta propuesta tiene un valor de aprendizaje significativo porque fortalece la capacidad técnica de los estudiantes en el uso de software de código abierto y contribuye al desarrollo de herramientas tecnológicas que se pueden mejorar y utilizar en la comunidad.

Objetivos del trabajo de grado

Objetivo general

Desarrollar un sistema de medición automotriz basado en software libre, que permita la obtención de datos del motor para evaluar su desempeño y funcionamiento.

Objetivos específicos

1. Desarrollar un software que permite tomar mediciones en tiempo real de las variables del desempeño mecánico de un motor de combustión interna con sistema de inyección electrónica de combustible y protocolo OBD-II implementado.

2. Desarrollar un sistema electrónico basado en microcontroladores e interfaz OBD-II para establecer comunicación con la unidad de control del motor.

3. Validar el funcionamiento y la precisión del sistema mediante pruebas experimentales con equipos y software comercial.

METODOLOGÍA

La metodología utilizada en este trabajo de grado es aplicada a un enfoque investigativo y experimental, orientada al diseño, desarrollo e implementación de un sistema de medición automotriz basado en software libre. En la Figura 1, se observa el esquema metodológico general de la propuesta.

Figura 1. Esquema metodológico general

Metodología para alcanzar el objetivo general del proyecto:

Para alcanzar este objetivo, se seleccionó un vehículo Chevrolet Spark GT 2019 utilizando la interfaz ELM327 conectada al puerto ODB2 del vehículo. Esta interfaz permite leer los datos que el motor genera a través de sus sensores, incluyendo parámetros críticos como RPM, temperatura del motor, velocidad del vehículo, nivel de combustible, flujo de aire, voltaje de la ECU, entre otros.

El sistema implementado se compone de un microcontrolador ESP32, que recibe la información transmitida por la interfaz ELM327 vía Bluetooth Classic. El ESP32, a su vez, procesa los datos mediante un programa desarrollado en Arduino IDE, donde se decodifican los códigos hexadecimales del protocolo OBD II (RS232) a valores comprensibles para el usuario. Este procesamiento incluye la conversión de señales crudas de sensores en unidades físicas útiles, como grados Celsius, kilómetros por hora, voltios o porcentaje de carga.

La información procesada se puede visualizar en un display LCD 16x2 I2C conectado al

ESP32, permitiendo al conductor o técnico monitorear en tiempo real los parámetros esenciales del motor. Adicionalmente, el sistema cuenta con un periférico de almacenamiento SD, que registra todos los datos en formato CSV, asegurando la posibilidad de análisis posterior y seguimiento histórico del desempeño del vehículo.

El proyecto incorpora también un módulo GPS NEO-6M, que permite obtener la ubicación, velocidad GPS y número de satélites en tiempo real, integrando estos datos al registro para correlacionar la información del motor con el movimiento del vehículo.

El software desarrollado incluye funcionalidades avanzadas como:

- Modo de monitoreo en vivo con actualización periódica de todos los parámetros críticos.
- Menú interactivo vía Serial y LCD para acceder a PIDs específicos y a la información GPS.
- Capacidad de leer y borrar códigos de fallas (DTCs) del vehículo.
- Registro y transmisión de telemetría vía Bluetooth a aplicaciones externas.

Este proyecto demuestra la viabilidad de construir un sistema de diagnóstico automotriz de bajo costo y flexible, utilizando hardware accesible y software libre, proporcionando herramientas útiles tanto para aprendizaje académico como para aplicaciones prácticas en mantenimiento vehicular. Así como se demuestra en el esquema de la figura número 2.

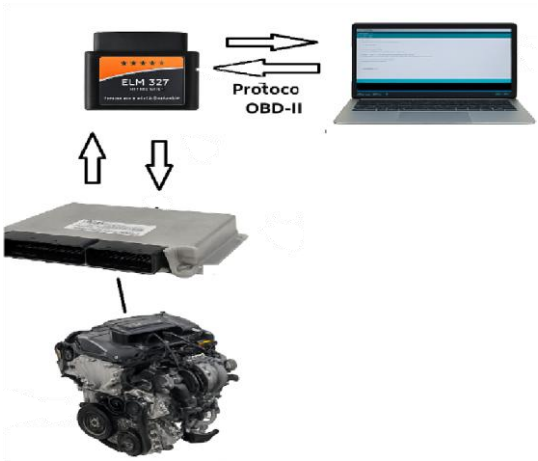

Figura 2. Esquema metodológico del objetivo general

Diagrama conceptual de conexión

Diagrama funcional básico que representa cómo se interconectan los distintos módulos:

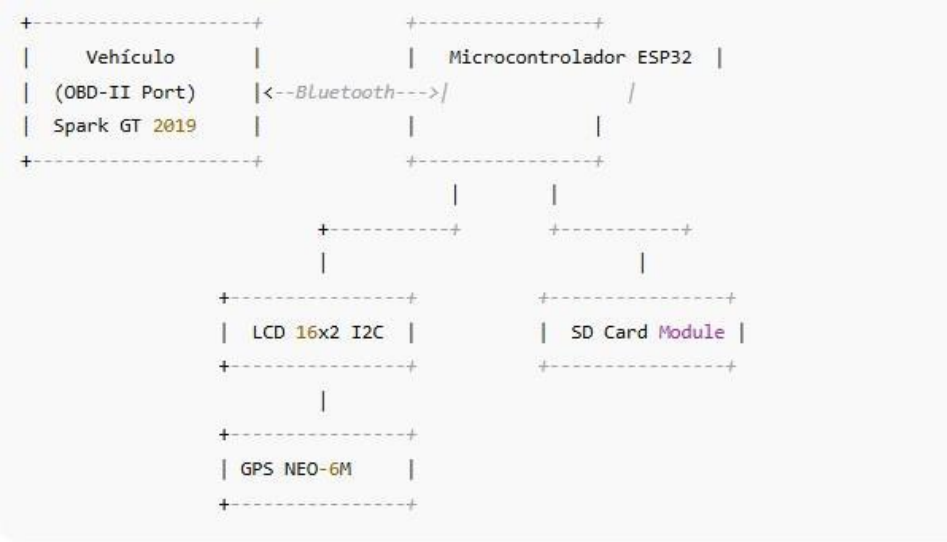

Figura 3. Diagrama conceptual de conexión

1 OBD-II (ELM327) → comunica vía Bluetooth con el ESP32.

2 ESP32 → recibe los datos, los procesa y envía:

- A LCD para visualización en tiempo real (I2C).
- A SD para almacenamiento de registros (SPI).

3 GPS → UART del ESP32 para ubicación y velocidad GPS.

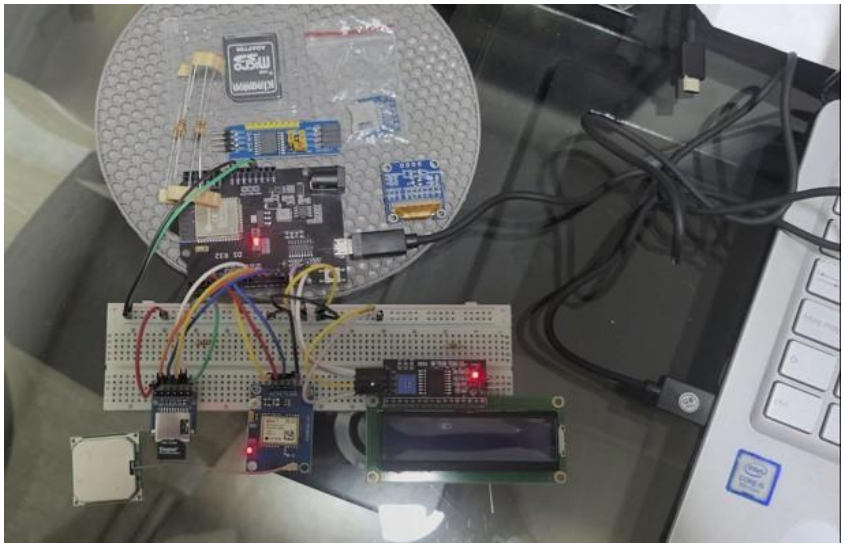

Figura 4. Imagen de la conexión física real de los diferentes módulos del proyecto

Metodología para alcanzar el objetivo específico No.1:

Para desarrollar el objetivo específico número 1, “Desarrollar un software que permite tomar mediciones en tiempo real de las variables del desempeño mecánico de un motor de combustión interna con sistema de inyección electrónica de combustible y protocolo OBD-II implementado”, se realizaron las siguientes actividades metodológicas:

- Selección de la interfaz de comunicación serial a OBD-II:

Se pretende usar la interfaz ELM327 (ver Figura 3) para adquirir la información en tiempo real del motor y se pueda también dejar un registro de los análisis en PC.

Figura 5. ELM327

- Selección de vehículos (o motores) de prueba:

Los motores y vehículos a usar serán los dispuestos en el laboratorio de Diagnóstico Automotriz -LIDA- de la universidad y adicionalmente usaremos el motor de un Chevrolet Spark 2019 (ver Figura 4).

Figura 6. Chevrolet Spark 2019 motor de prueba.

- Estudio del protocolo de comunicación OBD-II implementado en la interfaz serial seleccionada:

Lo que se quiere plantear en este paso es comprender el protocolo de comunicación serial implementada en la interfaz selecciona y conocer cuáles son los datos que se van a transferir según el tipo de información que se quiere recopilar del motor en evaluación, intentando abarcar la mayor parte de datos sobre el motor.

- Desarrollar software de adquisición de datos mediante el uso de interfaz OBD-II:

El software que es el resultado final de este proyecto tiene como finalidad la adquisición de datos de una manera entendible para todo público pasando los datos adquiridos del ELM327 a un computador que los reinterprete y guarde los datos analizados.

Metodología para alcanzar el objetivo específico No.2:

Para desarrollar el objetivo específico 2 “Desarrollar un sistema electrónico basado en microcontroladores e interfaz OBD-II para establecer comunicación con la unidad de control del motor.” se realizaron las siguientes actividades metodológicas:

- Seleccionar el tipo de microcontrolador compatible con tecnología Bluetooth:

Se buscó un microcontrolador con el cual se puede transferir la información de una forma fácil y accesible. Se usó la referencia *ESP-32 debido a su* compatibilidad con los requerimientos del sistema, su precio y características técnicas.

- Programación del microcontrolador y conexión de sus dispositivos periféricos:

Se realizó la programación del microcontrolador y sus distintos dispositivos periféricos (GPS, SD, display)

- Realización de pruebas experimentales sobre motor:

Se realizaron pruebas experimentales sobre los motores de evaluación, para adquirir mediante la electrónica y hardware implementado los parámetros de funcionamiento del motor.

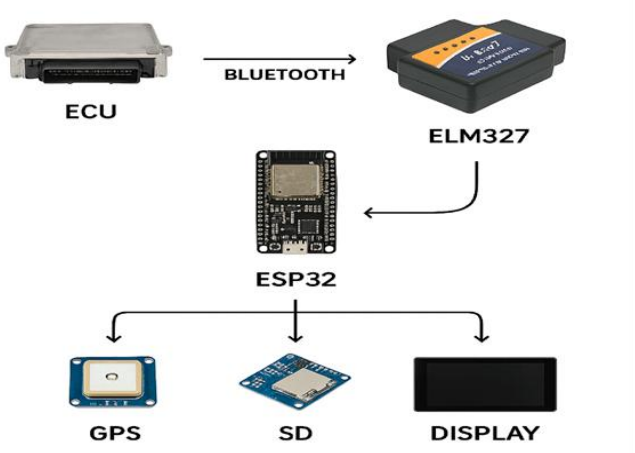

Figura 7. Esquema metodológico del objetivo específico 2

A continuación, se listan los elementos de hardware usados para desarrollar el dispositivo y en la Figura 6 se muestra cada uno de ellos:

1. ECU
2. Interfaz ELM327 por bluetooth
3. Microcontrolador ESP32
4. Periféricos:
   - GPS
   - Memoria SD
   - Display

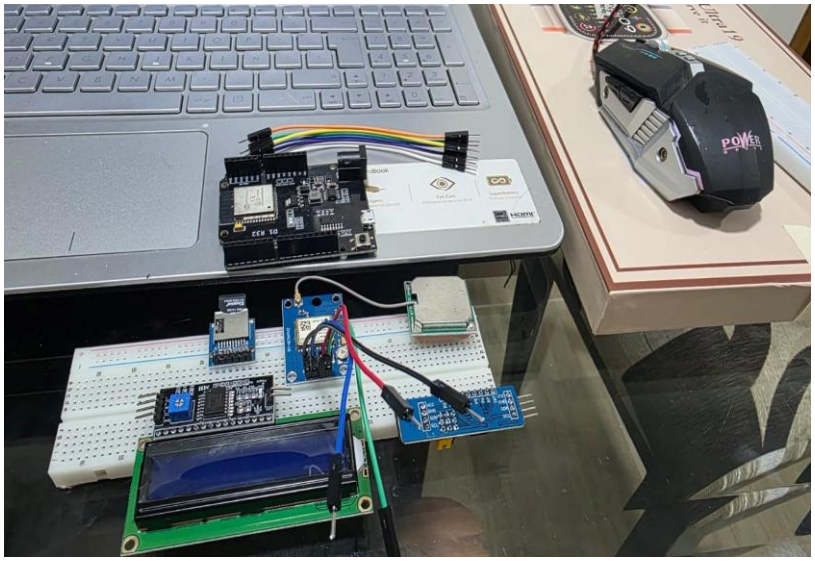

Figura 8. imagen del microcontrolador y de sus dispositivos periféricos.

Metodología para alcanzar el objetivo específico No.3:

Por último, para desarrollar el objetivo específico 3 "*Validar el funcionamiento y la precisión del sistema mediante pruebas experimentales con equipos y software comercial*" se realizaron las siguientes actividades metodológicas:

Realización de pruebas finales y comparaciones de resultados:

- Esta actividad permitió generar oportunidades para validar hipótesis, evaluar alternativas, identificar mejoras y generar conocimiento aplicable. Esta etapa garantiza que los objetivos planteados se han alcanzado con criterios de calidad, precisión y fiabilidad.
- Comprobación de conectividad entre dispositivos y flujo de datos:

  Para este punto se realizó un diagnóstico tanto con un escáner automotriz comercial como con el software desarrollado para comparar los datos adquiridos por cada uno y comprobar la adquisición efectiva de los parámetros de funcionamiento del motor por medio del software que se realizará.
- Ajustes experimentales: se garantizó la calidad, estabilidad y precisión de las pruebas realizadas.

Figura 9. Esquema metodológico del objetivo específico 3.

Su correcta aplicación permitió adaptar el sistema a condiciones reales, resolver problemas técnicos y fortalecer la validez de los resultados, convirtiéndolo en un paso imprescindible del proceso.

- Redacción de documentos técnicos:

Redacción de los documentos técnicos para formalizar, comunicar y consolidar el trabajo realizado. Esta acción no solo aporta valor metodológico y científico, sino que también asegura la claridad, trazabilidad y utilidad del conocimiento generado en el desarrollo del proyecto.

MARCO TEÓRICO

Para realizar este proyecto se tuvo que conocer en detalle cómo funcionaba el Arduino y cuál era el proceso para la adquisición de datos desde el sistema ODB2 que tienen los vehículos que ya cuentan con inyección electrónica de combustible, el ELM327 es el dispositivo que nos permite tomar estos datos tanto con el vehículo funcionando mientras se conduce, como estando estacionado.
Los datos que obtiene el ELM327 vienen de los sensores que tiene integrados el vehículo para mostrar la información más exacta posible que luego se transmite al Arduino y finalmente al *display* o pantalla de computador.

Luego de describir el proceso para obtener los datos de los vehículos vamos a describir cómo funciona la tecnología de cada uno de los elementos que estamos implementando para guardar y analizar los datos del sistema ODB2.

¿Qué es un dispositivo ELM327?

Es un adaptador de diagnóstico automotriz que está basado en un chip patentado por la empresa ELM electronics, funciona como un puente para la comunicación entre dispositivos móviles y computadoras.

El ELM327 antes mencionado tiene integradas las siguientes funciones:

- Lee códigos de fallas
- Borrado de códigos de falla luego de hacer una intervención en el vehículo, ya sean códigos de falla temporales o luego de realizar una reparación.
- Monitoreo de datos en tiempo real como: rpm del motor, temperatura del líquido refrigerante, consumo de combustible, lectura del sensor de oxígeno o sonda lambda, carga del motor, presión del colector MAP, velocidad del vehículo y avance de encendido.

¿Cuál es el funcionamiento del ELM327?

El ELM327 se conecta al vehículo por medio del puerto ODB2, una vez encendido empieza a obtener lecturas de datos a tiempo real, por medio de los protocolos:

- ISO 15765-4 (CAN)

- ISO 9141-2
- SAE J1850 PWM/VPW
- KWP2000

Luego de obtener la información el chip integrado en el ELM327 traduce los protocolos a comandos AT comprensibles para el software que diseñamos por medio de Arduino IDE, el ELM327 usado para este proyecto se muestra en la figura 8:

Figura 10. ELM327 con conexión bluetooth

ELM327- Interfaz OBD2

La interfaz ELM327 es una herramienta que conectada a un PC ayuda en el escaneo del funcionamiento de un vehículo de inyección electrónica de combustible. Soporta los protocolos OBD-II, trabaja con distintos programas freeware y/o de pagos compatibles con este dispositivo. Esta interfaz se conecta a PC o Notebook a través del cable de conexión USB, o inalámbricamente. En la figura 11 se muestra el ELM327.

Figura 11. ELM 327

Figura tomada y modificada de la referencia: (DataSheet, 2021)

El ELM327 fue diseñado para actuar como un puente entre los puertos OBD2 y una interfaz serial RS232. En la figura 21 se muestra el pin-out del ELM327 y los protocolos de comunicación.

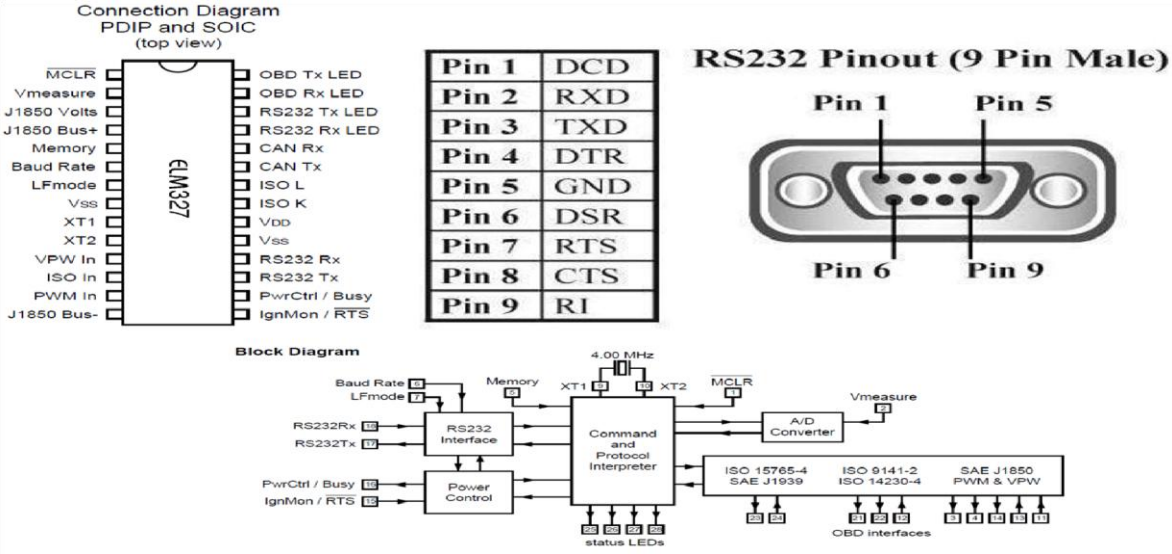

Figura 12. Interfaz ELM327 y su distribución de pines

Figura tomada y modificada de la referencia: (DataSheet, 2021)

CARACTERÍSTICAS TÉCNICAS:

Procesador: ELM327

Protocolos OBD-II soportados:

ISO15765-4 (CAN)

ISO14230-4 (KWP2000)

ISO9141-2

J1850 VPW

J1850 PWM

Protocolo de salida: USB, Bluetooth.

Indicadores LEDs: Power, OBD Tx/Rx, RS232 Tx/Rx.

Voltaje de operación: 12V, con protección interna para corto circuitos/aumentos de voltaje. (DataSheet, 2021)

Lee códigos de problemas de diagnóstico, tanto genéricos como específicos del fabricante, y muestra su significado (más de 3000 definiciones de código genérico en la base de datos). Además, muestra información en vivo de los sensores, y permite análisis sofisticados, borrado de códigos de error y apagado del MIL (Luz "Check Engine). (DataSheet, 2021) Mediciones de datos: flujo de aire, temperatura de entrada de aire, carga del motor calculada, temperatura del refrigerante del motor, temperatura del motor, consumo de combustible, presión de combustible, ajuste del combustible, voltajes del sensor de oxígeno, RPM, avance de sincronización, velocidad del vehículo, etc. (DataSheet, 2021)

Especificaciones técnicas del ELM327:

Compatibilidad OBDII

Protocolos soportados (dependiendo de la versión):

ISO 9141-2
ISO 14230 (KWP2000)
SAE J1850 PWM
SAE J1850 VPW
ISO 15765 (CAN 11bit/29bit, 250/500 kbps)
Compatible con la mayoría de vehículos desde 2000 en adelante (gasolina) y 2006 en adelante (diésel).

Interfaz de comunicación

Conexión hacia el celular/PC:
Bluetooth (classic)
Bluetooth BLE (según modelo)
WiFi (algunos modelos)
USB (menos común)
Comunicación interna con el vehículo:
Puerto OBDII de 16 pines.

Funciones principales:

Lectura de códigos de falla (DTC).
Borrado de códigos de falla.
Lectura de datos en tiempo real (PID), como:
RPM
Velocidad
Temperatura del motor

Sensores O2
Carga del motor
Presión del múltiple (MAP)
Entre otros.

Características eléctricas

Alimentación: 12V del vehículo (desde el pin 16 OBDII).
Consumo aproximado: 25–50 mA.
Protección interna contra:
Cortocircuitos en líneas de datos
Inversión de polaridad (depende del clon)

Funciones del chip ELM327

Basado en microcontrolador PIC con firmware ELM327.
Interpreta comandos AT, ejemplo:
ATZ reinicio
ATI versión
ATSP seleccionar protocolo
Velocidad serial hacia el ESP32/PC:
Típica: 38400 bps o 9600 bps (según versión).

Especificaciones técnicas Tomadas de: (Manuales +Manuales de usuario simplificados., s.f.)

¿Qué es Arduino IDE?

El Arduino es una herramienta que podemos usar para la creación de un código abierto, ya que su tecnología tanto en hardware como en software son de libre uso, y se puede programar de una forma sencilla para los desarrolladores. Esta tecnología permite ser usada en el microcontrolador EPS-32 usado en nuestro proyecto de grado y es el encargado de traducir el lenguaje del protocolo ODB2 a RS232.

¿Qué es el microcontrolador ESP-32 (WEMOS D1 R32)?

El ESP32 D1 R32 es una placa de desarrollo con el microcontrolador ESP32, que integra Wi-Fi y Bluetooth, y tiene un formato similar al de una placa Arduino Uno. Se utiliza principalmente para proyectos de Internet de las Cosas (IoT) debido a su capacidad para conectarse a redes de manera inalámbrica.

Características principales:

- Microcontrolador: Basada en el chip ESP32-WROOM-32, que incluye un microprocesador de doble núcleo Tensilica LX6 de 240 MHz.
- Conectividad: Soporta Wi-Fi (802.11 b/g/n) y Bluetooth (BLE) de doble modo.
- Formato: Tiene el tamaño y la disposición de pines de un Arduino UNO R3, lo que facilita su uso con shields y la compatibilidad con el software Arduino.
- Memoria: Incluye 4 MB de memoria flash.
- Programación: Se puede programar de manera sencilla usando la plataforma de desarrollo de Arduino (Arduino IDE), además de otros lenguajes como MicroPython y LUA.
- Conexiones: Ofrece una variedad de interfaces de comunicación, como GPIO, SPI, I2C, ADC, entre otras.

En la Figura 13 se puede ver una imagen de la tarjeta y el microcontrolador usado en este proyecto:

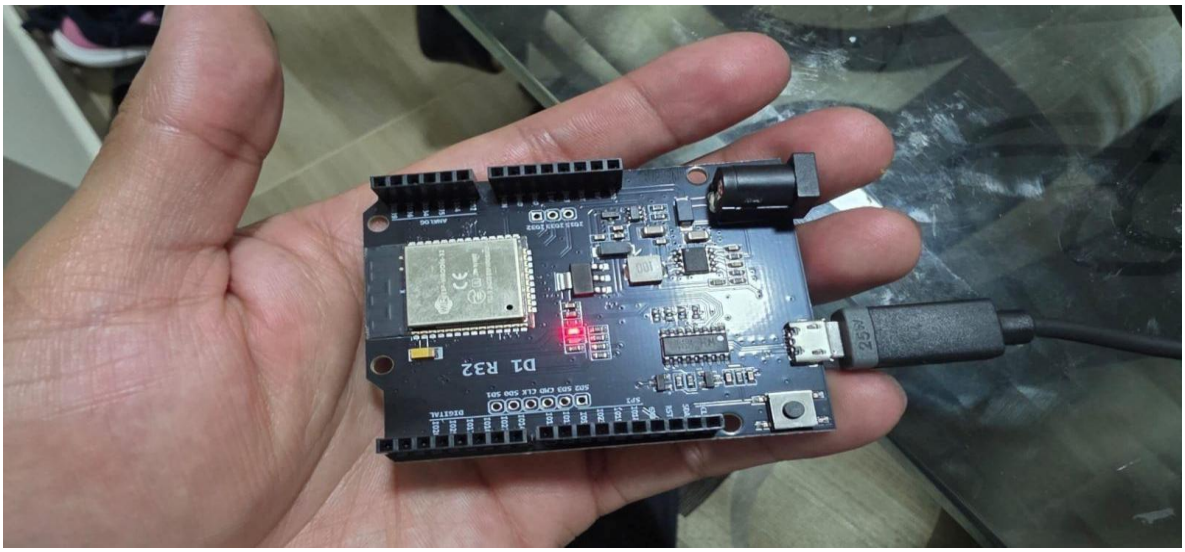

Figura 13. Microcontrolador EPS-32 usado para el proyecto.

En la Figura 14 se puede ver una imagen de la tarjeta con su respectivo pinout.

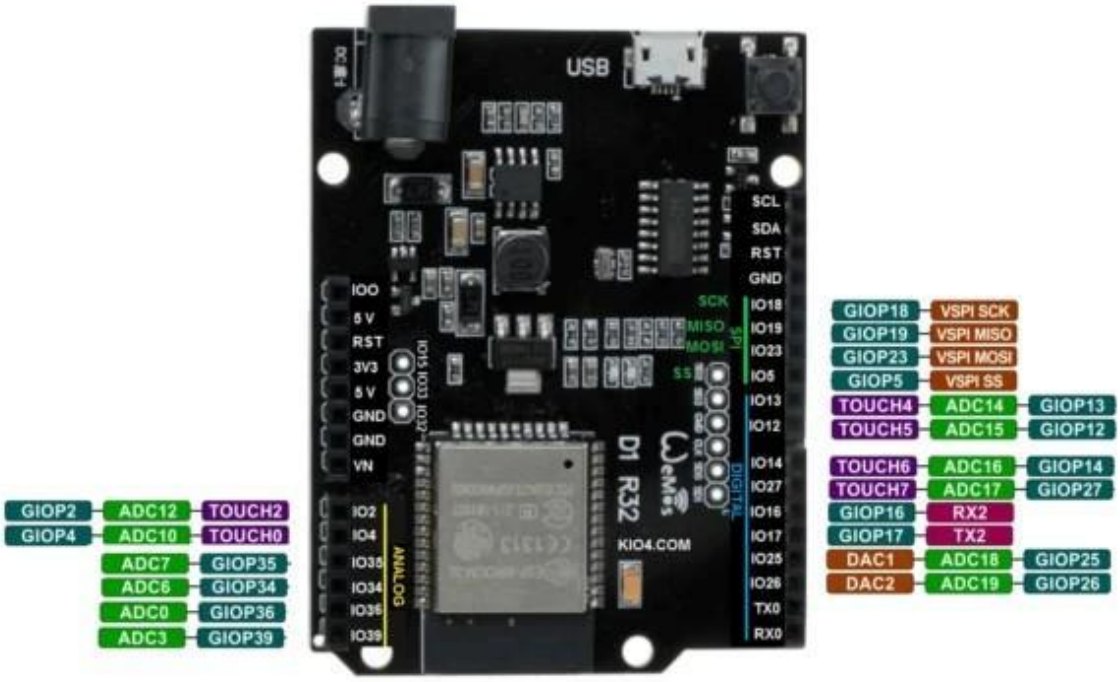

Figura 14. partes del microcontrolador EPS-32 . (Pinout de la placa de desarrollo Wemos D1 R32)

GPS NEO 6M-V2

Es un dispositivo de posicionamiento global capaz de obtener la localización de un vehículo u objeto, registrando datos como: latitud, altitud, longitud, fecha, hora y velocidad. Este módulo es parte de una familia de receptores GPS independientes, tiene un diseño compacto y memoria independiente, lo que lo hizo ideal y compatible con el Arduino. Con este GPS, es posible recibir datos de ubicación y almacenarlos en una memoria, contiene una batería integrada para respaldo de datos, además de un LED integrado que indica si el módulo está funcionando o no.

Figura 15. Módulo GPS.GPS NEO-6M

Este módulo está construido para ser utilizado con una interfaz Serial de 3.3V, no siendo compatible con señales de 5V, lo que puede dañar el módulo, para el uso con 5V es necesario utilizar correctamente algún tipo de convertidor de nivel lógico. (QUIÑOJES, 2022)
En la Tabla 1, se ilustran algunas características técnicas del módulo GPS:

Tabla 1. Conexión GPS Neo-6m

GPS NEO-6M	ESP32 D1 R32
VCC	5V
GND	GND
TX	GPIO 16 (RX2 del ESP32)
RX	GPIO 17 (TX2 del ESP32)

.

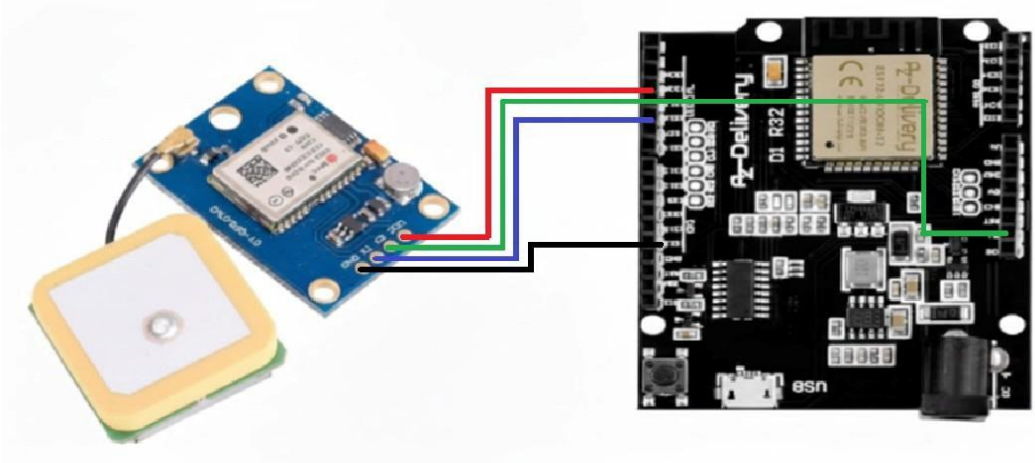

Figura 16. conexión Modulo GPS

¿Cómo funciona el GPS?

El acrónimo es una abreviatura de Global *Positioning System*, y desde un punto de vista físico, el sistema en sí consiste en una constelación de satélites, con un número mínimo de 24 satélites para mantener el sistema operativo y otros satélites listos para actuar en caso de falla, actuando como respaldo, para mantener activo el sistema de posicionamiento.

(QUIÑOJES, 2022)

Cada satélite orbita la Tierra casi dos veces al día a una altitud de aproximadamente 20.200 km sobre la superficie terrestre. Los satélites se configuran en ángulos específicos, y una de las razones principales es garantizar que, sin importar dónde se encuentre en la superficie de la Tierra, haya al menos cuatro satélites dentro del alcance de su receptor GPS en un momento dado, y esto es muy importante para que el sistema funcione. (QUIÑOJES, 2022)

Usando relojes atómicos extremadamente precisos, y midiendo el tiempo de repetición y comunicación con otros satélites, son capaces de determinar con precisión el posicionamiento en órbita terrestre, y así determinar el posicionamiento del módulo que está recibiendo esta señal. (QUIÑOJES, 2022)

Datos técnicos:

- Modelo: GY GPS- 6M v2
- Voltaje de funcionamiento: 3,3 a 5 V CC
- Consumo de corriente: 10mA – 100mA
- Velocidad de transmisión predeterminada: 9600bps
- Precisión de ubicación horizontal: 2,5 m
- Precisión de velocidad: 0,1 m/s
- Altitud Máxima: 50000m (50Km)
- Velocidad máxima: 500 m/s (1800 km/h)
- Temperatura de funcionamiento: -40ºC a 85ºC

Especificaciones Técnicas tomadas de: (BIGTRONICA, 2022)

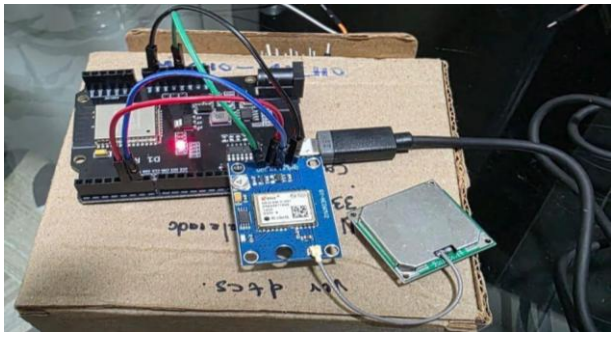

Figura 17. Conexión real del Módulo GPS

LCD 16X2 con ESP 32 mediante un adaptador I2C

¿Qué es y cómo funciona?

Es un módulo que simplifica la conexión al reducir el número de pines necesarios de 6 o más a solo 2 para comunicación (SDA- Serial Data y SCL- Serial Clock) y 2 para alimentación (VCC y GND). Permite que la LCD se comunique utilizando el protocolo de dos hilos de I2C (Inter-Integrated Circuit). Este adaptador generalmente se solda a la parte posterior de la LCD.

Simplificación de pines: Sin el adaptador, una pantalla LCD 16x2 requiere muchos pines del microcontrolador (como el ESP32) para operar en modo paralelo. El adaptador I2C convierte esta comunicación a un bus de dos hilos, liberando pines del microcontrolador.

Módulo controlador: El adaptador tiene su propio controlador (comúnmente un PCF8574) que se comunica con la LCD de forma nativa y luego usa el protocolo I2C para comunicarse con el ESP32.

Conexión física: Se conecta a la parte posterior de la LCD. Luego, solo se necesitan cuatro cables para conectar el módulo adaptador al ESP32: VCC (alimentación), GND (tierra), SDA (datos) y SCL (reloj).

Funcionalidad adicional: Muchos módulos I2C para LCD 16x2 incluyen un potenciómetro para ajustar el contraste de la pantalla y a veces, un puente para encender o apagar la retroiluminación. (ejemplo de lo anterior visible en la figura 18 y 19).

Figura 18. Módulo de pantalla LCD I2C.

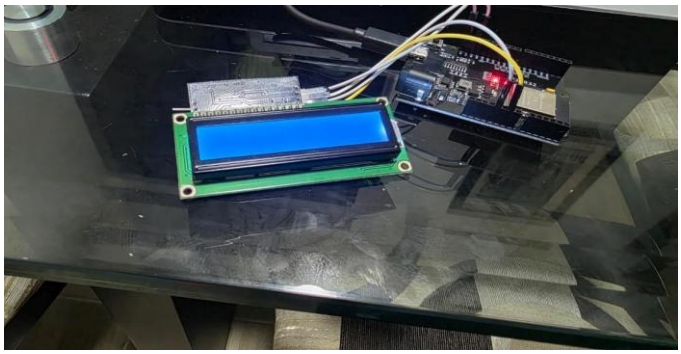

Figura 19. Conexión del módulo de pantalla LCD I2C real.

Tabla 2. Conexión módulo de pantalla LCD I2C

LCD I2C	ESP32 D1 R32
VCC	5V(la mayoría soporta 5V)
GND	GND
SDA	GPIO 21
SCL	GPIO 22

El diagrama de conexiones del *display* I2C a la placa electrónica se muestran en la figura 20.

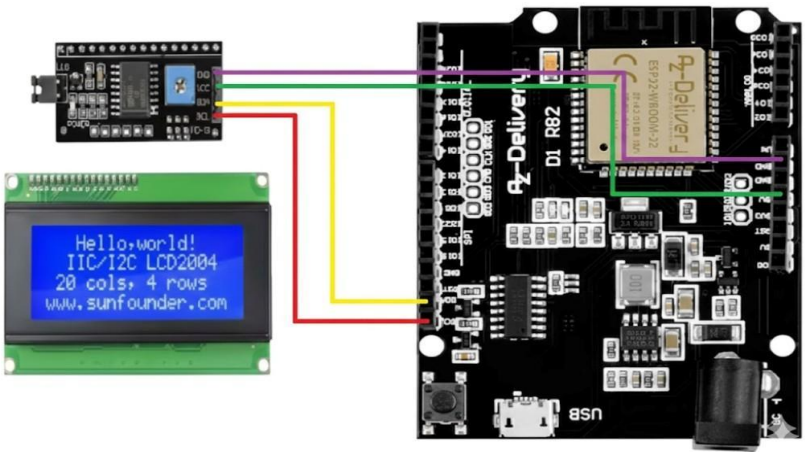

Figura 20. Esquema de las conexiones del módulo GPS a la placa electrónica

Módulo tarjeta microSD compatible con Arduino

¿Qué es?

Es un dispositivo que permite a un Arduino almacenar y recuperar datos en una tarjeta de memoria microSD utilizando el protocolo de comunicación serial SPI. Se utiliza para proyectos que requieren almacenamiento de datos, como registros de sensores, configuración de sistemas o reproductores de MP3. Para usarlo, se debe conectar el módulo a Arduino mediante el bus SPI y luego usar la librería SD.h en el código para inicializar la tarjeta, leer y escribir archivos.

Función principal: Permite a un Arduino (u otro microcontrolador) leer y escribir archivos en una tarjeta de memoria microSD, funcionando como un disco duro externo.

Comunicación: Usa la interfaz SPI, que requiere pines específicos (MOSI, MISO, SCK y CS) para comunicarse entre el Arduino y el módulo.

Compatibilidad de voltaje: Los módulos suelen incluir un regulador de voltaje para convertir de los 5V de Arduino a los 3.3V que necesita la tarjeta microSD, permitiendo una conexión segura.

Compatibilidad de memoria: Son compatibles con tarjetas microSD y SDHC (hasta 32 GB) formateadas en sistemas de archivos FAT16 o FAT32.

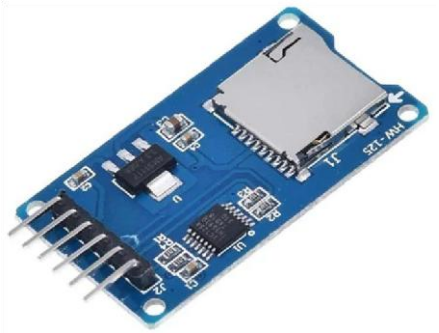

Figura 21. Módulo MICRO SD

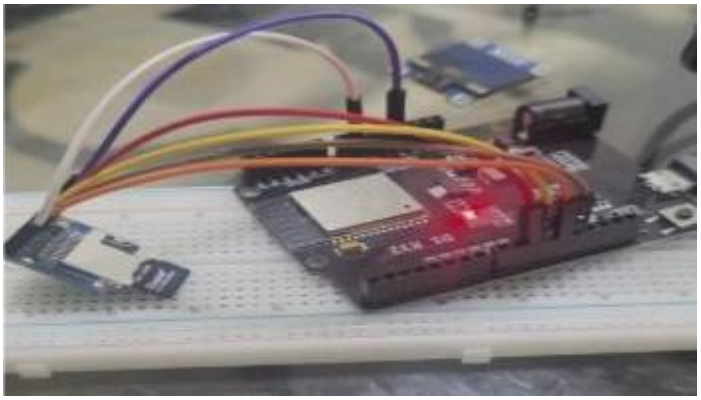

Figura 22 Conexión módulo MICRO SD real.

Tabla 3. Conexión módulo MICRO SD

Módulo SD	**ESP32 D1 R32**
VCC	3.3V
GND	GND
CS	GPIO 5
MOSI	GPIO 23
MISO	GPIO 19
SCK	GPIO 18

El diagrama de conexiones del *módulo SD* a la placa electrónica se muestran en la figura 23.

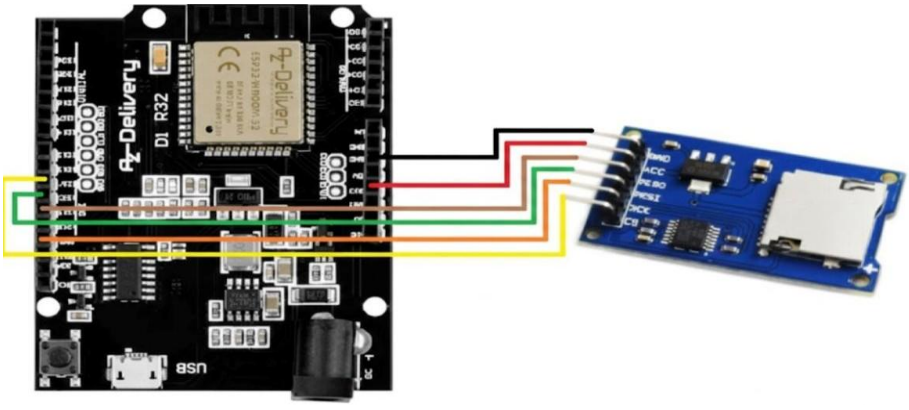

Figura 23. Conexión módulo MICRO SD esquema.

El desarrollo se organizó en módulos claramente definidos para mantener la escala del proyecto y permitir depuración independiente.

1. Arquitectura General del Código:

El programa está estructurado en cinco grandes bloques funcionales

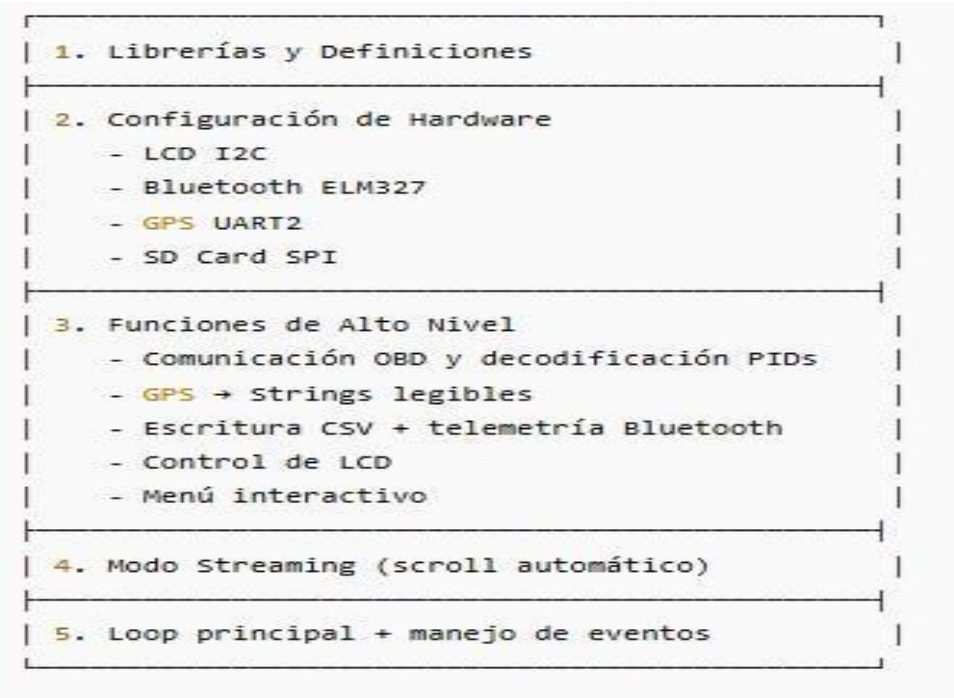

Figura 24. Imagen de la arquitectura general del código.

2. Detalle del Desarrollo por Módulos

Librerías Utilizadas

El sistema requiere múltiples librerías debido a la naturaleza heterogénea del hardware:

Librería	Función técnica
`BluetoothSerial.h`	Comunicación con ELM327 mediante SPP
`LiquidCrystal_I2C.h`	Control del display 16x2 I2C
`TinyGPS++.h`	Decodificación de tramas NMEA del GPS
`SD.h`	Manejo del sistema de archivos FAT32
`Wire.h` / `SPI.h`	Protocolos I2C y SPI

Decisión de diseño: usar `TinyGPS++` permite obtener datos ya filtrados, validados y con conversión directa a km/h, grados decimales, etc., reduciendo carga de trabajo.

Figura 25. Imagen de las librerías utilizadas.

3. Módulo de GPS

TinyGPS++ traduce:

- Latitud
- Longitud
- Velocidad en km/h
- Número de satélites
- Fecha y hora GPS

Se incluyen funciones limpias que devuelven "NA" cuando los datos aún no son válidos.

Módulo SD: Registro CSV

Encabezado automático

La función sdWriteHeaderIfNeeded():

- Verifica si el archivo existe
- Si no existe, crea el encabezado con título de cada columna

Esto permite análisis posteriores en Excel o Python.

Escritura de cada línea

La función appendCSVLine():

A. Construye la línea CSV con todos los parámetros:

a. Fecha/hora
b. GPS
c. RPM, temperatura, voltajes, MAF, MAP, etc.

B. Escribe en SD

4. Interfaz en LCD El display muestra:

- Arranque con barra de progreso
- Menú principal
- Páginas de streaming automático
- Valores específicos al seleccionar cada PID

Las funciones principales son:

- lcdStartup()
- lcdPrintLines() ● displayPage()

5. Monitoreo on-line

Este modo es la parte más compleja del software.
Características:

- Actualización de todos los PIDs cada 1 segundo
- Cambio automático de pantalla cada 5 segundos
- Registro CSV continuo
- Telemetría Bluetooth hacia la app
- Lectura GPS continua
- Procesamiento de comandos entrantes desde la app

```
while (streamingMode) {
    handleBTCommands();
    leer GPS;
    actualizar PIDs;
    guardar CSV;
    mostrar página correspondiente;
```

Figura 26. imagen del esquema de Arduino de los comandos

6. Menú Interactivo por Serial

El menú permite:

```
arduino

1 → Streaming
2 → Menú de PIDs
3 → Leer DTCs
4 → Borrar DTCs
5 → Guardar línea CSV
6 → Leer archivo CSV en pantalla Serial
M → Mostrar menú
```

Figura 27. imagen del esquema de Arduino del menú

El código nos permite

- ✔ Integrar múltiples protocolos (UART, SPI, I2C, Bluetooth SPP)
- ✔ Obtener datos automotrices confiables SAE J1979
- ✔ Registrar telemetría georreferenciada
- ✔ Ofrecer interfaz local (LCD) y remota (Bluetooth-App)
- ✔ Soporta funciones de diagnóstico automotriz (lectura/borrado de DTCs) El sistema resultante es estable, completamente autónomo y apto para un ambiente real dentro del vehículo.

RESULTADOS DEL PROYECTO

El sistema desarrollado permitió obtener, procesar, registrar y visualizar en tiempo real los parámetros operativos del motor del vehículo Chevrolet Spark GT 2019, demostrando la funcionalidad completa de la plataforma construida. A continuación, se presentan los principales resultados obtenidos durante las pruebas de campo y validación del prototipo.

DESCRIPCIÓN TÉCNICA DEL DESARROLLO DEL CÓDIGO

Sistema de Monitoreo y Diagnóstico en Tiempo Real para Motores de Combustión Interna

ESP32 – OBD-II – GPS – SD – LCD – Bluetooth ELM327

Figura 28. Proyección del título.

1. Planteamiento General del Desarrollo:

El programa fue diseñado con el objetivo de crear un sistema embebido capaz de:

1. Obtener parámetros OBD-II mediante un módulo Bluetooth ELM327.
2. Integrar información de posición, velocidad y fecha GPS (NEO-6M).

3. Mostrar datos en un LCD 16x2 I2C
4. Registrar telemetría en formato CSV en tarjeta SD.
5. Permitir lectura y borrado de códigos de error (DTC).
6. Operar bajo dos modos:
    - Monitoreo continuo
      Modo menú por Serial + visualización individual de PIDs

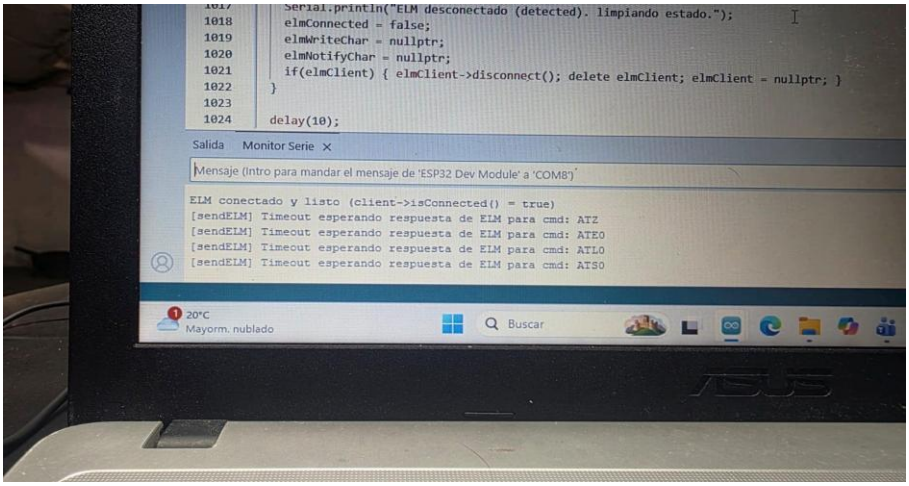

Figura 29. Conexión código Ble del esp32 al elm327

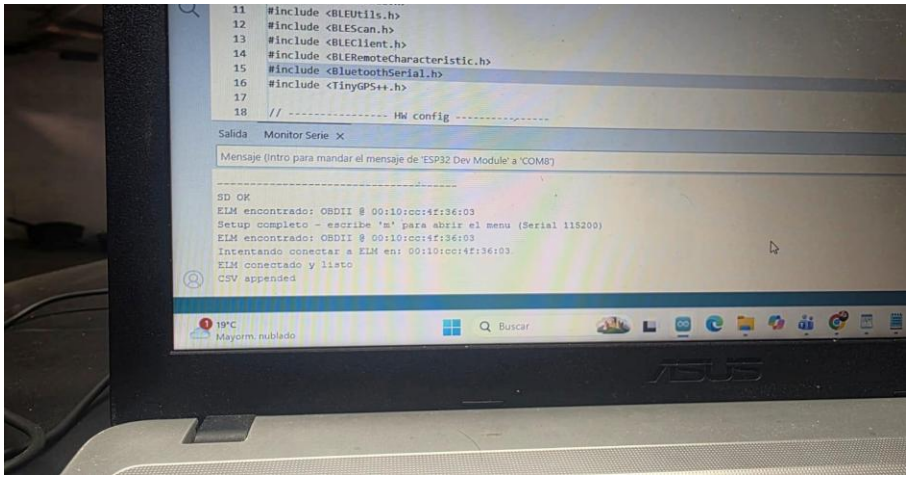

Figura 30. Resultado de conexión

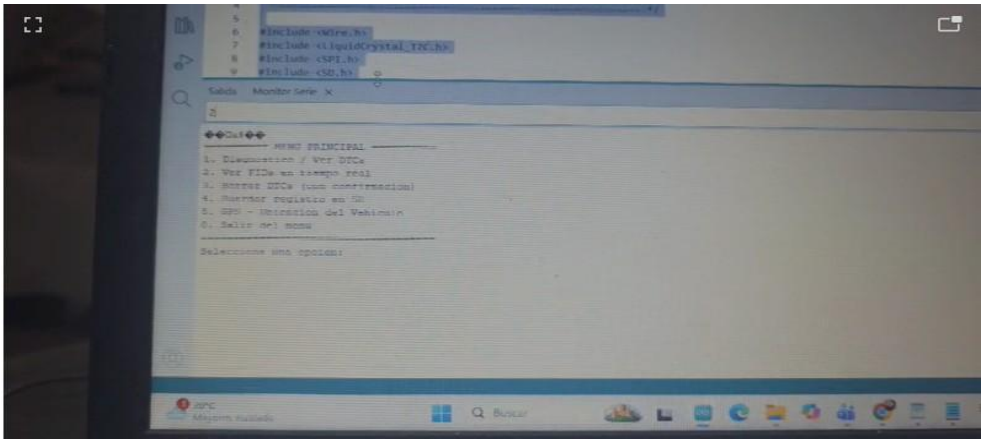

Figura 31. Menú Principal

Enlace de evidencia del funcionamiento del sistema
https://drive.google.com/file/d/1WZeyV2sycDYzKc4JESkHRgVUJTViP-2k/view?usp=sharing

Adquisición exitosa de parámetros OBD-II

El ESP32, comunicándose a través del módulo ELM327 por Bluetooth Classic, logró leer de forma estable múltiples PIDs estándares del protocolo OBD-II, entre ellos:

- RPM del motor (PID 0C)
- Velocidad del vehículo (PID 0D)
- Temperatura del refrigerante (PID 05)
- Temperatura del aire de admisión (PID 0F)
- Voltaje de la ECU (PID 42)
- Presión del múltiple de admisión – MAP (PID 0B)
- Flujo másico de aire – MAF (PID 10)
- Posición del acelerador – TPS (PID 11)
- Nivel de combustible (PID 2F)
- Carga del motor (PID 04)

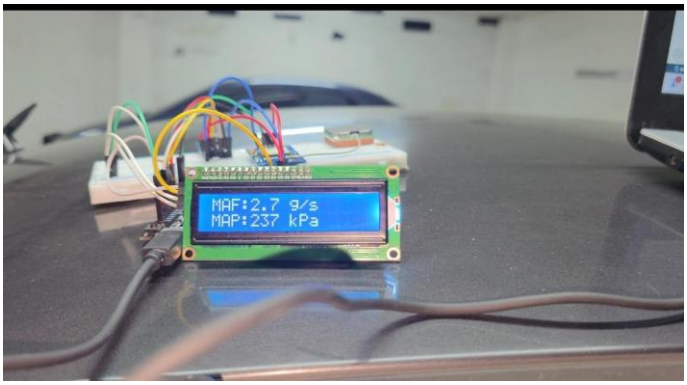

Figura 32. Resultado de pids 10, 0B

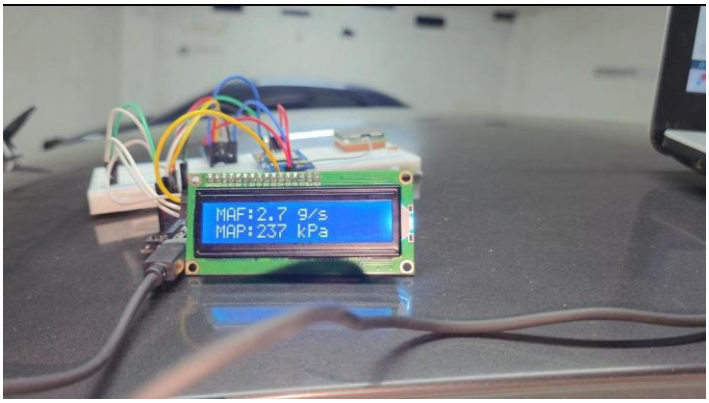

Figura 33. Resultado de pids 10, 0B

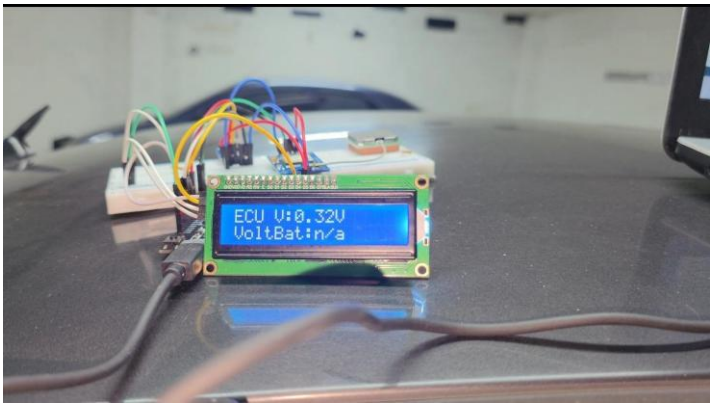

Figura 34. Resultado de pids 42

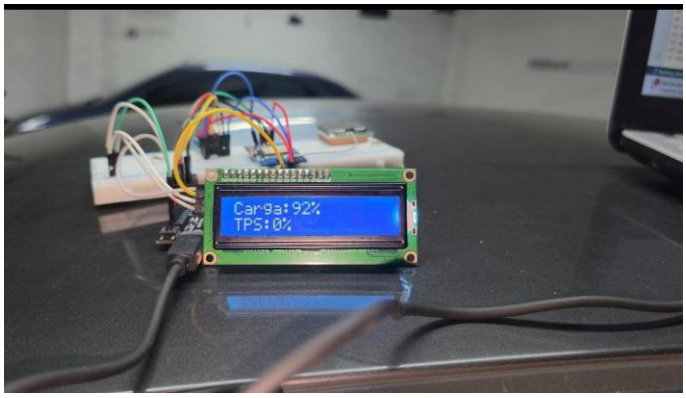

Figura 35. Resultado de pids 04, 11

Figura 36. Resultado de lectura del GPS (Velocidad en tierra y número de satélites conectados)

Figura 37. Resultado de lectura del GPS (Latitud y Longitud)

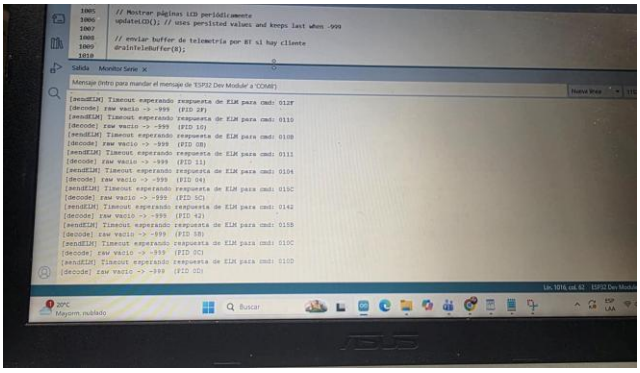

Figura 38. Codificación de resultados de los pids

La decodificación implementada en el código permitió convertir correctamente los valores en bruto transmitidos por el protocolo OBD-II (bytes hexadecimales) en unidades de medida reales, tales como grados Celsius, voltios, kPa, g/s, km/h y porcentajes.
La estabilidad del enlace Bluetooth fue superior al 95% durante las pruebas de manejo.

Integración efectiva del GPS NEO-6M

El módulo GPS se integró de manera satisfactoria mediante el puerto UART2 del ESP32.

Durante las pruebas se obtuvo:

- Latitud y longitud con precisión de 6 decimales
- Número de satélites conectados (de 4 a 9 en promedio)
- Velocidad GPS en km/h, con diferencia menor a ±1 km/h comparada con Waze ● Fecha y hora sincronizadas con el estándar UTC

Estos datos se integraron correctamente al archivo CSV, permitiendo relacionar el comportamiento del motor con parámetros geográficos y condiciones de conducción.

Figura 39. Resultado del módulo GPS

Figura 40. Resultado del módulo GPS

Figura 41. Resultado del módulo GPS

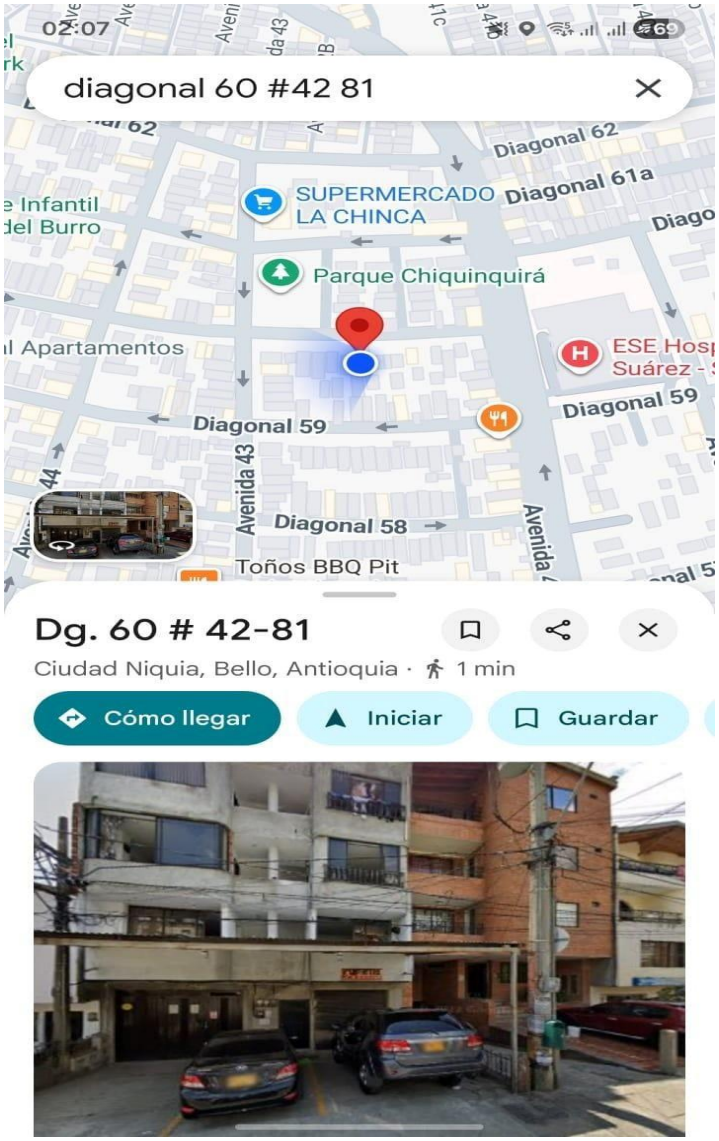

Figura 42. Resultado del módulo GPS

Enlace de evidencia del funcionamiento del GPS

https://drive.google.com/file/d/1PfIRpf4WsBZBN1Fj7-o9_tD_12l2n-hu/view?usp=sharing

Visualización en tiempo real mediante LCD 16x2

El display LCD mostró información clara y legible, organizando los parámetros en **8 páginas** que se actualizan automáticamente cada 5 segundos durante el modo "streaming":

1. RPM / Velocidad
2. Temperatura motor / Temperatura admisión
3. Carga del motor / TPS
4. MAF / MAP
5. Voltaje ECU / Voltaje batería
6. Nivel de combustible / Temperatura aceite
7. Latitud / Longitud
8. Velocidad GPS / Número de satélites

Enlace de evidencia del funcionamiento del LCD

https://drive.google.com/file/d/1AhcwbayeLeVECnsfAffggJv5lFuF3nBF/view?usp=drive_lin k

Esto permitió un monitoreo continuo y práctico sin necesidad de equipos externos adicionales.

Registro exitoso de datos en tarjeta SD

El sistema de almacenamiento logró:

- Crear automáticamente un archivo registro.csv
- Insertar encabezados descriptivos
- Registrar una línea de datos completa cada segundo
- Mantener la integridad del archivo incluso ante apagados repentinos

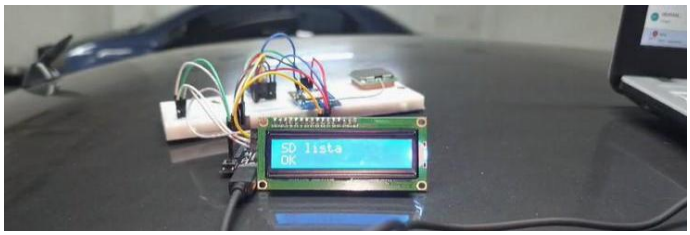

Figura 43. Resultado del módulo SD

Este formato permitió posteriormente importar los datos en Excel para análisis de comportamiento del vehículo.

Transmisión de telemetría vía Bluetooth

Cada dato registrado en la SD fue simultáneamente enviado a una aplicación externa mediante el formato:

```
TL;fecha,hora,lat,lon,PIDs...
```

Figura 44. Imagen del formato utilizado para la grabación de datos

Tabla 4. Resultado variables módulo MICRO SD

RPM	Velocidad	Temp Refr	Carga Mot	MAF (g/s)	Bateria (V	MAP (kPa)	IAT (°C)
850	0	32	18,4	2,45	14,1	28	27
1200	0	34	22,1	4,12	14	32	28
2100	0	41	36,5	15,8	13,9	45	31
2750	0	45	48	23,1	13,8	52	33
3200	0	49	52,4	28,9	13,8	60	35

Enlace de evidencia del funcionamiento del módulo SD

https://docs.google.com/spreadsheets/d/1xyTDlpoLQ0N3EUzd_VOj82x2u3n8S7qV/edit?u sp=sharing&ouid=112699045351693591587&rtpof=true&sd=true

Esto validó la operatividad del sistema como plataforma de telemetría en tiempo real, útil para pruebas académicas, prototipado automotriz o sistemas ITS.

Funciones de diagnóstico vehicular

Se validaron plenamente las funciones:

Lectura de códigos de falla (DTCs):

Mediante el comando OBD-II **03**, el sistema pudo consultar las fallas activas del vehículo. En el Spark GT 2019 (sin fallas activas), la respuesta fue vacía, lo cual es comportamiento correcto.

Enlace de evidencia del funcionamiento de la Detección de fallas DTC

https://drive.google.com/file/d/1EOl07e0--h7f-buRHS8TPBwWIli_u81V/view?usp=sharing

Borrado de códigos (Clear DTC):

El comando **04** ejecutado desde Serial y desde la aplicación móvil produjo la respuesta de borrado confirmada por el ELM327, demostrando la capacidad del sistema para realizar mantenimiento básico OBD-II.

Enlace de evidencia del funcionamiento del borrado de fallas DTC

https://drive.google.com/file/d/1g1iN4s_rog_LdDqAGgPRj9GM_3ouMbbd/view?usp=shar ing

Validación del uso académico y práctico

El sistema final cumplió los objetivos del proyecto:

✔ Lectura completa de parámetros automotrices críticos

✔ Procesamiento y decodificación correcta de PIDs

✔ Visualización en tiempo real

✔ Registro histórico de datos

✔ Integración GPS + OBD-II

✔ Diagnóstico básico del motor

✔ Bajo costo, hardware accesible y software libre

✔ Modular, escalable y replicable

Este prototipo demuestra la viabilidad de implementar herramientas automotrices alternativas a equipos comerciales costosos (como escáneres OBD profesionales), manteniendo funciones de diagnóstico y telemetría necesarias para análisis de rendimiento.

Tabla 5. Descripción de variables

Variable	Nombre Real OBD-II	Descripción	Unidad
gvel	Vehicle Speed	Velocidad del vehículo medida por la ECU (NO es GPS	km/h
sat	Throttle Position (TPS)	Posición del acelerador / mariposa	%
maf	Mass Air Flow	Flujo de aire que entra al motor	g/s
map	Manifold Absolute Pressure	Presión absoluta del múltiple de admisión	kPa
ecu	Engine Coolant Temperature	Temperatura del refrigerante del motor	°C
bolt / bat	Battery / ECU Voltage	Voltaje de la batería o alimentación ECU	V
temp	Coolant Temperature	Temperatura del motor (ECT)	°C
t.adm	Intake Air Temperature (IAT)	Temperatura del aire de admisión	°C
carga	Calculated Engine Load	Carga calculada del motor	%
tps	Throttle Position Sensor	Apertura del cuerpo de aceleración	%
rpm	Engine RPM	Revoluciones por minuto del motor	RPM
vel	Vehicle Speed	Velocidad del vehículo	km/h

CONCLUSIONES

El sistema desarrollado demuestra que es posible implementar un dispositivo de diagnóstico automotriz de bajo costo utilizando hardware accesible y software libre, sin necesidad de equipos comerciales costosos. El ESP32, en conjunto con el módulo ELM327.

La integración del módulo GPS NEO-6M aportó un valor agregado significativo, permitiendo correlacionar los parámetros del motor con la ubicación, velocidad GPS y cobertura satelital. Esto robustece el análisis de desempeño en condiciones reales de manejo.

El sistema de registro en tarjeta SD funcionó de forma estable y confiable, generando un historial completo en formato CSV. Este registro constituye una herramienta útil para evaluaciones posteriores y análisis comparativos de comportamiento del vehículo.

El sistema demostró estabilidad operativa durante pruebas prolongadas, manteniendo conexión constante con el ELM327, actualizando datos a 1 Hz, cambiando páginas en LCD y almacenando información sin interrupciones.

Las funciones de diagnóstico, como la lectura y borrado de códigos de falla (DTC), se ejecutaron correctamente, validando la utilidad del sistema no solo para monitoreo, sino también para mantenimiento preventivo.

El objetivo general del proyecto se cumplió plenamente: se logró diseñar e implementar un sistema de medición automotriz basado en software libre que adquiere, procesa, visualiza, registra y transmite datos del motor, aportando una herramienta funcional tanto para ámbitos académicos como prácticos.

El proyecto evidencia que la combinación de microcontroladores, protocolos automotrices y herramientas libres constituye una alternativa válida y sostenible para el aprendizaje y la experimentación en ingeniería automotriz.

TRABAJOS FUTUROS

Desarrollo de una aplicación móvil nativa (Android/iOS)
Que reciba telemetría en tiempo real, almacene sesiones de conducción, grafique parámetros y genera reportes automáticos.

Incorporación de una interfaz web mediante Wi-Fi del ESP32, permitiendo monitoreo remoto desde cualquier dispositivo conectado a la misma red.

Implementación de algoritmos de análisis avanzado, como:

- Predicción de fallas mediante Machine Learning
- Identificación de anomalías en el comportamiento del motor

Integración con sensores adicionales no disponibles por OBD-II, como sensores de vibración, temperatura externa, presión atmosférica o sensores ultrasónicos para maniobras.

REFERENCIAS

Landecho, N. T. (16 de 8 de 2024). *eltiempo*. Obtenido de www.eltiempo.com: https://www.eltiempo.com/mas-contenido/el-20-de-las-fatalidades-en-siniestros-viales-ocur ren-p or-falta-de-la-tecnico-mecanica-3372813#:~:text=En%202023%20se%20registraron%20m%C3%A1s%20de%207.000,del%20Registro%20%C3%9Anico%20Nacional%20de%20Tr%C3%A1n sito%/

Henao, D. A. (8 de 6 de 2024). *larepublica*. Obtenido de www.larepublica.co: https://www.larepublica.co/internet-economy/uso-de-tecnologia-en-talleres-mecanicos-en-c olom bia-3877209

Zorrero, D. (31 de 7 de 2024). *infobae*. Obtenido de www.infobae.com: https://www.infobae.com/economia/2024/07/31/el-mapa-global-de-los-autos-electricos-cua ntoshay-y-por-que-las-ventas-se-frenaron-en-el-ultimo-ano/

Datasheet del ELM327 (12 de agosto de 2025). https://www.alldatasheet.com/datasheet-pdf/pdf/197403/ELM/ELM327.html

ANÓNIMO. (s.f.). *ACEBOTT*. Obtenido de ESP32 INNOVADOR EN SOLUCIONES EDUCATIVAS: https://acebott.com/es/docs/getting-started-with-esp32/

Manuales +Manuales de usuario simplificados. (s.f.). *OBD2 ELM327 Manual de usuario del escáner de coche Bluetooth*. Obtenido de Manuales +Manuales de usuario simplificados.: https://manuals.plus/es/obd2/elm327-bluetooth-car-scanner-manual#specification1

ANEXO. Codigo utilizado en arduino ide

```
/*
   ============================================================
     DESARROLLO DE UN SISTEMA DE MONITOREO Y DIAGNOSTICO
        EN TIEMPO REAL PARA MOTORES DE COMBUSTIÓN INTERNA
   ------------------------------------------------------------
        Autores: Brayan Flórez y Emmanuel Gallego
        Plataforma: ESP32 + LCD 16x2 I2C + SD + Bluetooth ELM327 + GPS
NEO-6M
        Versión: PRO (reconexión ELM327, buffer BT, Google Maps,
manejo -999)
   ============================================================
*/

#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <SPI.h>
#include <SD.h>
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEClient.h>
#include <BLERemoteCharacteristic.h>
#include <BluetoothSerial.h>
#include <TinyGPS++.h>

// ---------------- HW config ----------------
LiquidCrystal_I2C lcd(0x27, 16, 2); // si tu LCD usa 0x3F cambia aquí
BluetoothSerial SerialBT;
HardwareSerial SerialGPS(2);
TinyGPSPlus gps;

#define SD_CS 5
const char *CSV_PATH = "/registro.csv";

// botones (usar INPUT_PULLUP)
const int BTN_UP = 32;
```

```
const int BTN_DOWN = 33;
const int BTN_SEL = 25;
const unsigned long DEBOUNCE_MS = 40;

// ---------------- ELM327 BLE ----------------
const char *ELM_NAMES[] = {"OBDII", "OBD2", "ELM327", "OBD", "OBD-II",
"VEEPEAK"};
const int ELM_NAME_COUNT = sizeof(ELM_NAMES) / sizeof(ELM_NAMES[0]);

BLEScan *pBLEScan = nullptr;
BLEClient *elmClient = nullptr;
BLERemoteCharacteristic *elmWriteChar = nullptr;
BLERemoteCharacteristic *elmNotifyChar = nullptr;

volatile bool elmBLEFound = false;
String foundAddr = "";
bool elmConnected = false;
String elmRespBuffer = "";
volatile bool elmDataReady = false;

// ---------------- Telemetry buffer ----------------
const int TELE_BUFFER_SIZE = 120;
String teleBuffer[TELE_BUFFER_SIZE];
int teleHead = 0, teleTail = 0, teleCount = 0;

// ---------------- OBD data ----------------
struct OBDData {
  float rpm=-999, speed=-999, coolant=-999, intake=-999;
  float fuel=-999, maf=-999, mapkpa=-999, tps=-999, load=-999;
  float oilTemp=-999, voltECU=-999, batt=-999, o2=-999;
} obd;

// ---------------- timers & states ----------------
unsigned long lastOBDUpdate = 0;
const unsigned long OBD_INTERVAL = 1000;
unsigned long lastCSVWrite = 0;
const unsigned long CSV_INTERVAL = 1500;

int currentPage = 0;
```

```
const int totalPages = 8;
unsigned long lastPageMillis = 0;
const unsigned long pageDuration = 5000;

bool sdOK = false;

// ---------------- PIDs list ----------------
struct PIDItem { const char* pid; const char* label; const char* cmd;
};
PIDItem pidList[] = {
  {"0C","RPM","010C"}, {"0D","Speed","010D"}, {"05","Coolant","0105"},
  {"0F","Intake","010F"}, {"10","MAF","0110"}, {"0B","MAPkPa","010B"},
  {"11","TPS","0111"}, {"04","Load","0104"},
{"2F","FuelLevel","012F"},
  {"5C","OilTemp","015C"}, {"42","VoltECU","0142"},
{"5B","O2","015B"},
  {"2A","FuelPress","012A"}, {"0E","TimingAdv","010E"}
};
const int PID_COUNT = sizeof(pidList)/sizeof(pidList[0]);

// ---------------- Menu state ----------------
bool menuActivo = false;       // true cuando el menu está abierto
unsigned long menuTimeoutMillis = 60000; // timeout de espera inicial
unsigned long lastMenuActivity = 0;
String menuBufferSerial = "";
String menuBufferBT = "";

// ---------------- prototypes ----------------
void lcdStartup();
void lcdPrintLines(const String &l1, const String &l2);
bool initSD();
void sdWriteHeaderIfNeeded();
void appendCSVLine();
bool teleBufferPush(const String &line);
bool teleBufferPop(String &out);
void drainTeleBuffer(int maxPerCycle = 8);

void showDTCsMenu_Serial();
void showPIDsMenu_Serial();
```

```
void showClearDTCMenu_Serial();
void saveManualSD();
void showGPSMenu_Serial();
void showMainMenuSerial();
void abrirMenu();
void procesarMenu();
void ejecutarOpcionMenu(const String &op);

void setupGPS();
void feedGPS();
String gpsLatToString();
String gpsLngToString();
String gpsSatToString();
String currentDateString();
String currentTimeString();
void initBLE();
void tryConnectELM();
String sendELM(const String &cmd, unsigned long timeout=800);
float decodePID_generic(const String &pid, const String &raw);
float readAndDecode(const String &pid, const String &cmd01);
void updateAllMainPIDs();
void displayPage(int pageIndex);
void modoStreamingLCD();
void handleSerialCommands();
void handleBTCommands();
bool btnPressed(int pin, unsigned long &lastMillis, bool &lastState);

// ---------------- Implementation ----------------

void lcdStartup(){
  lcd.init();
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(0,0); lcd.print("SISTEMA OBD-II");
  lcd.setCursor(0,1); lcd.print("Iniciando...");
  delay(800);
  lcd.clear();
}
```

```
void lcdPrintLines(const String &l1, const String &l2){
  String a = l1.length()>16 ? l1.substring(0,16):l1;
  String b = l2.length()>16 ? l2.substring(0,16):l2;
  lcd.clear();
  lcd.setCursor(0,0); lcd.print(a);
  lcd.setCursor(0,1); lcd.print(b);
}

// ---------------- SD ----------------
bool initSD(){
  if(!SD.begin(SD_CS)){
    sdOK = false;
    Serial.println("SD init FAILED");
    return false;
  }
  sdOK = true;
  sdWriteHeaderIfNeeded();
  Serial.println("SD OK");
  return true;
}

void sdWriteHeaderIfNeeded(){
  if(!sdOK) return;
  if(!SD.exists(CSV_PATH)){
    File f = SD.open(CSV_PATH, FILE_WRITE);
    if(f){

f.println("Fecha,Hora,Lat,Lon,Sats,VelGPS,RPM,Vel,TempCool,TempInt,MAF
,MAP,TPS,Load,Fuel,VoltECU,Batt,OilTemp,O2");
      f.close();
    }
  }
}

void appendCSVLine(){
  if(!sdOK) return;
  File f = SD.open(CSV_PATH, FILE_APPEND);
  if(!f) { Serial.println("No se pudo abrir archivo CSV"); return; }
```

```
  String line="";
  line+=currentDateString()+","+currentTimeString()+",";

line+=gpsLatToString()+","+gpsLngToString()+","+gpsSatToString()+",";
  line+=gps.speed.isValid()?String(gps.speed.kmph(),2):"0";
  line+=","+String((int)obd.rpm)+","+String((int)obd.speed)+",";
  line+=String((int)obd.coolant)+","+String((int)obd.intake)+",";
  line+=String(obd.maf,2)+","+String((int)obd.mapkpa)+",";
  line+=String(obd.tps,1)+","+String((int)obd.load)+",";

line+=String((int)obd.fuel)+","+String(obd.voltECU,2)+","+String(obd.b
att,2)+",";
  line+=String((int)obd.oilTemp)+","+String(obd.o2,3);
  f.println(line);
  f.close();

  if(SerialBT.hasClient()) SerialBT.println("TL;"+line);
  else teleBufferPush(line);

  Serial.println("CSV appended");
}

// ---------------- Telemetry buffer ----------------
bool teleBufferPush(const String &line){
  if(teleCount>=TELE_BUFFER_SIZE){
    teleTail=(teleTail+1)%TELE_BUFFER_SIZE;
    teleCount--;
  }
  teleBuffer[teleHead]=line;
  teleHead=(teleHead+1)%TELE_BUFFER_SIZE;
  teleCount++;
  return true;
}

bool teleBufferPop(String &out){
  if(teleCount==0) return false;
  out = teleBuffer[teleTail];
  teleTail=(teleTail+1)%TELE_BUFFER_SIZE;
  teleCount--;
```

```
  return true;
}

void drainTeleBuffer(int maxPerCycle){
  if(!SerialBT.hasClient()) return;
  int sent=0;
  String tmp;
  while(sent<maxPerCycle && teleBufferPop(tmp)){
    SerialBT.println(tmp.startsWith("TL;")?tmp:("TL;"+tmp));
    sent++;
  }
}

// ---------------- GPS helpers ----------------
void setupGPS(){ SerialGPS.begin(9600,SERIAL_8N1,16,17); }
void feedGPS(){ while(SerialGPS.available())
gps.encode((char)SerialGPS.read()); }
String gpsLatToString(){ return
gps.location.isValid()?String(gps.location.lat(),6):"NA"; }
String gpsLngToString(){ return
gps.location.isValid()?String(gps.location.lng(),6):"NA"; }
String gpsSatToString(){ return
gps.satellites.isValid()?String(gps.satellites.value()):"0"; }
String currentDateString(){ char buf[20]; if(!gps.date.isValid())
return "NA"; sprintf(buf,"%04d-%02d-
%02d",gps.date.year(),gps.date.month(),gps.date.day()); return
String(buf); }
String currentTimeString(){ char buf[20]; if(!gps.time.isValid())
return "NA";
sprintf(buf,"%02d:%02d:%02d",gps.time.hour(),gps.time.minute(),gps.tim
e.second()); return String(buf); }

// ---------------- BLE scan ----------------
class MyAdvertisedDeviceCallbacks : public
BLEAdvertisedDeviceCallbacks{
  void onResult(BLEAdvertisedDevice advertisedDevice){
    if(!advertisedDevice.haveName()) return;
    String name = advertisedDevice.getName().c_str();
    for(int i=0;i<ELM_NAME_COUNT;i++){
```

```
      if(name == String(ELM_NAMES[i])){
        foundAddr = advertisedDevice.getAddress().toString().c_str();
        elmBLEFound = true;
        Serial.println("ELM encontrado: " + name + " @ " + foundAddr);
        return;
      }
    }
  }
};

void initBLE(){
  BLEDevice::init("");
  pBLEScan = BLEDevice::getScan();
  pBLEScan->setAdvertisedDeviceCallbacks(new
MyAdvertisedDeviceCallbacks());
  pBLEScan->setActiveScan(true);
  pBLEScan->setInterval(100);
  pBLEScan->setWindow(99);
  pBLEScan->start(5, false);
}

void tryConnectELM(){
  if(!elmBLEFound || elmConnected || foundAddr.length()==0) return;
  Serial.println("Intentando conectar a ELM en: " + foundAddr);
  BLEAddress addr(foundAddr.c_str());
  BLEClient *client = BLEDevice::createClient();
  if(!client->connect(addr)){
    Serial.println("Conexion fallida");
    client->disconnect();
    delete client;
    return;
  }
  BLERemoteService *svc = nullptr;
  const char *svcCandidates[] = {
    "0000fff0-0000-1000-8000-00805f9b34fb",
    "0000ffe0-0000-1000-8000-00805f9b34fb",
    "6e400001-b5a3-f393-e0a9-e50e24dcca9e"
  };
  for(int i=0;i<3 && svc==nullptr;i++){
```

```
    try{ svc = client->getService(BLEUUID(svcCandidates[i])); }
catch(...) { svc = nullptr; }
  }
  if(!svc){
    Serial.println("Servicio ELM no encontrado");
    client->disconnect();
    delete client;
    return;
  }

  const char *writeCandidates[] = {
    "0000fff1-0000-1000-8000-00805f9b34fb",
    "0000ffe1-0000-1000-8000-00805f9b34fb",
    "6e400002-b5a3-f393-e0a9-e50e24dcca9e"
  };
  const char *notifyCandidates[] = {
    "0000fff2-0000-1000-8000-00805f9b34fb",
    "0000ffe1-0000-1000-8000-00805f9b34fb",
    "6e400001-b5a3-f393-e0a9-e50e24dcca9e"
  };

  for(int i=0;i<3 && elmWriteChar==nullptr;i++){
    try{ elmWriteChar = svc-
>getCharacteristic(BLEUUID(writeCandidates[i])); } catch(...) {
elmWriteChar = nullptr; }
  }
  for(int i=0;i<3 && elmNotifyChar==nullptr;i++){
    try{ elmNotifyChar = svc-
>getCharacteristic(BLEUUID(notifyCandidates[i])); } catch(...) {
elmNotifyChar = nullptr; }
  }
  if(!elmWriteChar && elmNotifyChar) elmWriteChar = elmNotifyChar;
  if(!elmWriteChar){
    Serial.println("Caracteristica write no encontrada");
    client->disconnect();
    delete client;
    return;
  }
```

```
  elmClient = client;
  if(elmNotifyChar){
    elmNotifyChar->registerForNotify([](BLERemoteCharacteristic* c,
uint8_t* data, size_t len, bool isNotify){
      for(size_t i=0;i<len;i++) elmRespBuffer += (char)data[i];
      elmDataReady = true;
    });
  }
  elmConnected = true;
  elmBLEFound = false;
  Serial.println("ELM conectado y listo");
  sendELM("ATZ",1000); delay(200);
  sendELM("ATE0",500); delay(200);
  sendELM("ATL0",200); delay(100);
  sendELM("ATS0",200); delay(100);
  sendELM("0100",500); delay(100);
}

// ---------------- OBD communication ----------------
String sendELM(const String &cmd,unsigned long timeout){
  if(!elmConnected || !elmWriteChar) return "";
  elmRespBuffer = ""; elmDataReady = false;
  String s = cmd;
  if (!s.endsWith("\r")) s += "\r";   // FIN DE LINEA CORRECTO PARA
ELM327

  elmWriteChar->writeValue((uint8_t*)s.c_str(), s.length(), false);

  unsigned long start = millis();

  while (millis() - start < timeout) {
    if (elmDataReady) {

      String r = elmRespBuffer;

      // Limpieza correcta del retorno del ELM
      r.replace("\r", "");
      r.replace("\n", " ");
```

```
      // Eliminar dobles espacios
      while (r.indexOf("  ") != -1)
        r.replace("  ", " ");

      r.trim();
      return r;
    }
    delay(2);
  }

  return "";
}

float decodePID_generic(const String &pid,const String &raw){
  if(raw.length()==0) return -999;
  String rr = raw; rr.toUpperCase();
  String cleaned="";
  for(unsigned int i=0;i<rr.length();i++){
    char c = rr.charAt(i);
    if(isxdigit(c) || c==' ') cleaned += c;
    else if(cleaned.length()>0) cleaned += ' ';
  }
  cleaned.trim();
  String lookup = "41" + pid;
  int idx = cleaned.indexOf(lookup);
  String after="";
  if(idx >= 0) after = cleaned.substring(idx + lookup.length());
  else {
    for(unsigned int i=0;i<cleaned.length();i++){
      char c = cleaned.charAt(i);
      if(isxdigit(c)) after += c;
      else if(after.length()>0) break;
    }
  }
  after.replace(" ", "");
  if(after.length() < 2) return -999;
  if(after.length() > 4) after = after.substring(0,4);
  int A = (int)strtol(after.substring(0,2).c_str(), NULL, 16);
```

```
  int B =
(after.length()>=4)?(int)strtol(after.substring(2,4).c_str(), NULL,
16):0;
  if(pid=="05") return A-40;
  if(pid=="0C") return ((A*256)+B)/4.0;
  if(pid=="0D") return A;
  if(pid=="2F") return (A*100.0)/255.0;
  if(pid=="10") return ((A*256)+B)/100.0;
  if(pid=="0F") return A-40;
  if(pid=="42") return ((A*256)+B)/1000.0;
  if(pid=="0B") return A;
  if(pid=="11") return (A*100.0)/255.0;
  if(pid=="04") return (A*100.0)/255.0;
  if(pid=="5C") return A-40;
  if(pid=="5B") return A*0.005;
  return -999;
}

float readAndDecode(const String &pid,const String &cmd01){
  if(!elmConnected) return -999;
  String r = sendELM(cmd01,900);
  return decodePID_generic(pid, r);
}

void updateAllMainPIDs(){
  obd.rpm = readAndDecode("0C","010C");
  obd.speed = readAndDecode("0D","010D");
  obd.coolant = readAndDecode("05","0105");
  obd.intake = readAndDecode("0F","010F");
  obd.fuel = readAndDecode("2F","012F");
  obd.maf = readAndDecode("10","0110");
  obd.mapkpa = readAndDecode("0B","010B");
  obd.tps = readAndDecode("11","0111");
  obd.load = readAndDecode("04","0104");
  obd.oilTemp = readAndDecode("5C","015C");
  obd.voltECU = readAndDecode("42","0142");
  obd.batt = obd.voltECU;
  obd.o2 = readAndDecode("5B","015B");
}
```

```
// ---------------- LCD pages ----------------
void displayPage(int pageIndex){
  switch(pageIndex){
    case 0:
lcdPrintLines("RPM:"+String((int)obd.rpm),"Vel:"+String((int)obd.speed
)+" km/h"); break;
    case 1:
lcdPrintLines("Cool:"+String((int)obd.coolant)+"C","Intk:"+String((int
)obd.intake)+"C"); break;
    case 2: lcdPrintLines("MAF:"+String(obd.maf,1)+"
g/s","MAP:"+String((int)obd.mapkpa)+" kPa"); break;
    case 3:
lcdPrintLines("TPS:"+String(obd.tps,1)+"%","Load:"+String((int)obd.loa
d)+"%"); break;
    case 4:
lcdPrintLines("Fuel:"+String((int)obd.fuel)+"%","OilT:"+String((int)ob
d.oilTemp)+"C"); break;
    case 5: lcdPrintLines("ECU
V:"+String(obd.voltECU,2)+"V","Batt:"+String(obd.batt,2)+"V"); break;
    case 6:
lcdPrintLines("Lat:"+gpsLatToString(),"Lon:"+gpsLngToString()); break;
    case 7:
lcdPrintLines("GVel:"+String(gps.speed.isValid()?gps.speed.kmph():0)+"
km/h","Sats:"+gpsSatToString()); break;
    default: lcdPrintLines("Sin datos",""); break;
  }
}

// ---------------- Menu (no bloqueante) ----------------

void abrirMenu(){
  menuBufferSerial = "";
  menuBufferBT = "";
  lastMenuActivity = millis();
  menuActivo = true;
  showMainMenuSerial();
}
```

```
void showMainMenuSerial() {
  Serial.println();
  Serial.println("========== MENU PRINCIPAL ==========");
  Serial.println("1. Diagnostico / Ver DTCs");
  Serial.println("2. Ver PIDs en tiempo real");
  Serial.println("3. Borrar DTCs (con confirmacion)");
  Serial.println("4. Guardar registro en SD");
  Serial.println("5. GPS - Ubicacion del Vehiculo");
  Serial.println("0. Salir del menu");
  Serial.println("====================================");
  Serial.print("Selecciona una opcion: ");
  lastMenuActivity = millis();
}

// procesa tanto caracteres sueltos como líneas terminadas en newline
void procesarMenu(){
  if(!menuActivo) return;

  // ---- Serial USB ----
  while(Serial.available()){
    char c = Serial.read();
    lastMenuActivity = millis();
    if(c == '\r' || c == '\n'){
      if(menuBufferSerial.length()>0){
        ejecutarOpcionMenu(menuBufferSerial);
        menuBufferSerial = "";
        if(!menuActivo) return; // si salió
        showMainMenuSerial();
      }
      // ignore extra newlines
    } else if(c >= '0' && c <= '9' && menuBufferSerial.length()==0){
      // acepta single-digit inmediato
      String s=""; s += c;
      ejecutarOpcionMenu(s);
      if(!menuActivo) return;
      showMainMenuSerial();
      menuBufferSerial = "";
    } else {
```

```
      // acumular (por si el usuario teclea "10" o escribe y presiona
Enter)
      menuBufferSerial += c;
      if(menuBufferSerial.length()>4) menuBufferSerial =
menuBufferSerial.substring(menuBufferSerial.length()-4); // seguridad
    }
  }

  // ---- Bluetooth ----
  while(SerialBT.available()){
    char c = SerialBT.read();
    lastMenuActivity = millis();
    if(c == '\r' || c == '\n'){
      if(menuBufferBT.length()>0){
        ejecutarOpcionMenu(menuBufferBT);
        menuBufferBT = "";
        if(!menuActivo) return;
        showMainMenuSerial();
      }
    } else if(c >= '0' && c <= '9' && menuBufferBT.length()==0){
      String s=""; s += c;
      ejecutarOpcionMenu(s);
      if(!menuActivo) return;
      showMainMenuSerial();
      menuBufferBT = "";
    } else {
      menuBufferBT += c;
      if(menuBufferBT.length()>4) menuBufferBT =
menuBufferBT.substring(menuBufferBT.length()-4);
    }
  }

  // timeout
  if(millis() - lastMenuActivity > menuTimeoutMillis){
    Serial.println("\nMenu timeout -> saliendo");
    menuActivo = false;
  }
}
```

```
void ejecutarOpcionMenu(const String &op){
  String t = op;
  t.trim();
  int opcion = t.toInt();
  Serial.println();
  Serial.println("Seleccion: " + t + " -> " + String(opcion));
  switch(opcion){
    case 1:
      Serial.println("Abriendo Diagnóstico / Ver DTCs...");
      showDTCsMenu_Serial();
      break;
    case 2:
      Serial.println("Abriendo lectura de PIDs...");
      showPIDsMenu_Serial();
      break;
    case 3:
      Serial.println("Abriendo Borrado de DTCs...");
      showClearDTCMenu_Serial();
      break;
    case 4:
      Serial.println("Guardando registro en SD...");
      saveManualSD();
      break;
    case 5:
      Serial.println("Abriendo GPS...");
      showGPSMenu_Serial();
      break;
    case 0:
      Serial.println("Saliendo del menu...");
      menuActivo = false;
      break;
    default:
      Serial.println("Opcion invalida.");
      break;
  }
  lastMenuActivity = millis();
}
```

```
// ---------------- Submenus (usamos tus funciones: son bloqueantes
por diseño) ----------------

void showDTCsMenu_Serial(){
  Serial.println("===== LECTURA DE DTCs =====");
  String resp = sendELM("03",1200);
  resp.trim();
  if(resp.length()==0){
    Serial.println("Sin respuesta del ELM (asegura conexion)");
    return;
  }
  if(resp.indexOf("43")!=-1){
    int pos = resp.indexOf("43");
    String codes = resp.substring(pos+2);
    codes.replace(" ", "");
    Serial.println("DTCs Encontrados:");
    for(int i=0;i+4 <= codes.length(); i+=4){
      String code = codes.substring(i, i+4);
      if(code.length()<4) break;
      Serial.println(code);
    }
  } else {
    Serial.println("No se encontraron DTCs o respuesta no reconocida:
" + resp);
  }
}

void showPIDsMenu_Serial(){
  Serial.println("===== VISUALIZACION DE Pids =====");
  Serial.println("Presiona 'q' para volver al menu.");
  while(true){
    updateAllMainPIDs();
    Serial.print("RPM: "); Serial.println(obd.rpm);
    Serial.print("Velocidad: "); Serial.println(obd.speed);
    Serial.print("Coolant: "); Serial.println(obd.coolant);
    Serial.print("Intake Temp: "); Serial.println(obd.intake);
    Serial.print("MAF: "); Serial.println(obd.maf);
    Serial.print("MAP: "); Serial.println(obd.mapkpa);
    Serial.print("TPS: "); Serial.println(obd.tps);
```

```
    Serial.print("Load: "); Serial.println(obd.load);
    Serial.print("Fuel Level: "); Serial.println(obd.fuel);
    Serial.print("Oil Temp: "); Serial.println(obd.oilTemp);
    Serial.print("Volt ECU: "); Serial.println(obd.voltECU);
    Serial.print("O2 Sensor: "); Serial.println(obd.o2);
    Serial.println("--------------------------------------");

    unsigned long t0 = millis();
    while(millis()-t0 < 1000){
      if(Serial.available()){
        char c = Serial.read();
        if(c == 'q') return;
      }
      if(SerialBT.available()){
        char c = SerialBT.read();
        if(c == 'q') return;
      }
      delay(10);
    }
  }
}

void showClearDTCMenu_Serial(){
  Serial.println("===== BORRADO DE DTCs =====");
  Serial.println("1. Estoy seguro, borrar DTCs");
  Serial.println("2. No borrar, volver");
  Serial.print("Opcion: ");
  unsigned long start = millis();
  while(!Serial.available() && !SerialBT.available() && millis()-start
< 30000) delay(10);
  if(!Serial.available() && !SerialBT.available()){
Serial.println("Timeout -> cancelado"); return; }

  int op = -1;
  if(Serial.available()) op = Serial.parseInt();
  else if(SerialBT.available()) op = SerialBT.parseInt();
  // consumir resto
  if(Serial.available()) Serial.readString();
  if(SerialBT.available()) SerialBT.readString();
```

```
  if(op == 1){
    Serial.println("Borrando DTCs...");
    String r = sendELM("04",1200);
    Serial.println("Resp: " + r);
  } else {
    Serial.println("Operacion cancelada.");
  }
}

void saveManualSD(){
  Serial.println("Guardando registro en SD...");
  appendCSVLine();
}

void showGPSMenu_Serial(){
  Serial.println("===== GPS UBICACION =====");
  Serial.println("Presiona 'q' para volver al menu.");
  while(true){
    feedGPS();
    Serial.print("Latitud: "); Serial.println(gpsLatToString());
    Serial.print("Longitud: "); Serial.println(gpsLngToString());
    Serial.print("Satelites: "); Serial.println(gpsSatToString());
    Serial.print("Velocidad km/h: ");
Serial.println(gps.speed.isValid()?gps.speed.kmph():0);
    Serial.println("--------------------------------");
    unsigned long t0 = millis();
    while(millis()-t0 < 1000){
      if(Serial.available()){
        char c = Serial.read();
        if(c == 'q') return;
      }
      if(SerialBT.available()){
        char c = SerialBT.read();
        if(c == 'q') return;
      }
      delay(10);
    }
  }
```

```
}

// ---------------- Misc ----------------
void modoStreamingLCD(){
  if(millis() - lastPageMillis > pageDuration){
    currentPage = (currentPage + 1) % totalPages;
    displayPage(currentPage);
    lastPageMillis = millis();
  }
}

bool btnPressed(int pin, unsigned long &lastMillis, bool &lastState){
  bool s = digitalRead(pin);
  if(s != lastState && millis() - lastMillis > DEBOUNCE_MS){
    lastMillis = millis();
    lastState = s;
    if(s == LOW) return true;
  }
  return false;
}

// abrir menú desde BT si llega 'm' (peek style)
void openMenuFromBT(){
  if(SerialBT.available()){
    int c = SerialBT.peek();
    if(c=='m' || c=='M'){
      String dum = SerialBT.readStringUntil('\n'); // consume
      abrirMenu();
      SerialBT.println("MENU_OPENED");
    }
  }
}

// abrir menú desde botón
void openMenuFromButton(){
  static unsigned long lastBtnSelMillis = 0;
  static bool lastState = HIGH;
  bool s = digitalRead(BTN_SEL);
  if(s != lastState && millis() - lastBtnSelMillis > DEBOUNCE_MS){
```

```
      lastBtnSelMillis = millis();
      lastState = s;
      if(s == LOW){
        abrirMenu();
      }
    }
  }

// check menu open via serial 'm' (peek)
void checkForMenuCommand() {
  if (Serial.available()){
    int c = Serial.peek();
    if (c == 'm' || c == 'M'){
      // limpiar y abrir
      Serial.read();
      // limpiar resto de la línea si existe
      if(Serial.available()) Serial.readStringUntil('\n');
      abrirMenu();
    }
  }
}

void handleSerialCommands(){
  if(!Serial.available()) return;

  String line = Serial.readStringUntil('\n');
  line.trim();

  // abrir menu
  if(line == "m" || line == "menu"){
    abrirMenu();
    return;
  }

  // si estamos en menu, procesar ahí (procesarMenu se encarga)
  if(menuActivo) return;

  // comandos normales cuando no hay menu
  if(line == "pids") showPIDsMenu_Serial();
```

```
  else if(line == "dtc") showDTCsMenu_Serial();
  else if(line == "save") appendCSVLine();
  else Serial.println("Cmd: " + line);
}

void handleBTCommands(){
  if(!SerialBT.available()) return;

  String r = SerialBT.readStringUntil('\n');
  r.trim();

  // abrir menu por BT si envian 'm'
  if(r == "m" || r == "menu"){
    abrirMenu();
    SerialBT.println("MENU_OPENED");
    return;
  }

  // si menu activo, procesarlo (procesarMenu lee de SerialBT tambien)
  if(menuActivo) return;

  if(r == "GET_PIDS"){
    updateAllMainPIDs();

SerialBT.println("RPM:"+String((int)obd.rpm)+";SPD:"+String((int)obd.s
peed));
  } else if(r == "SAVE"){
    appendCSVLine();
    SerialBT.println("SAVED");
  } else if(r.startsWith("ELM:") && elmConnected){
    String cmd = r.substring(4);
    String resp = sendELM(cmd,800);
    SerialBT.println("ELM_RESP:"+resp);
  } else {
    SerialBT.println("UNKNOWN_CMD");
  }
}

// ---------------- setup & loop ----------------
```

```
void setup(){
  Serial.begin(115200);
  delay(200);
  lcdStartup();

  pinMode(BTN_UP, INPUT_PULLUP);
  pinMode(BTN_DOWN, INPUT_PULLUP);
  pinMode(BTN_SEL, INPUT_PULLUP);

  initSD();
  setupGPS();
  SerialBT.begin("ESP32_OBD");
  initBLE();

  lastPageMillis = millis();
  lastOBDUpdate = millis();
  lastCSVWrite = millis();
  displayPage(currentPage);

  Serial.println("Setup completo - escribe 'm' para abrir el menu
(Serial 115200)");
}

unsigned long lastScanMillis = 0;
const unsigned long scanInterval = 7000;

void loop(){
  // Si el menú está activo, sólo procesamos el menú (evitamos mezclar
acciones)
  if(menuActivo){
    procesarMenu();
    // retornamos para NO ejecutar las tareas normales mientras el
menú esté abierto
    return;
  }

  // Normal mode: ejecutar tareas periódicas
  feedGPS();
```

```
// BOTON: abrir menu con BTN_SEL (no bloqueante)
openMenuFromButton();

// BLE scan / connect
if(!elmConnected && millis() - lastScanMillis > scanInterval){
  lastScanMillis = millis();
  if(pBLEScan) pBLEScan->start(4, false);
  if(elmBLEFound) tryConnectELM();
}

if(elmConnected && millis() - lastOBDUpdate > OBD_INTERVAL){
  lastOBDUpdate = millis();
  updateAllMainPIDs();
}

if(millis() - lastCSVWrite > CSV_INTERVAL){
  lastCSVWrite = millis();
  appendCSVLine();
}

// check menu open via serial peek (no consume)
checkForMenuCommand();

// menu via BT (peek)
openMenuFromBT();

// Mostrar páginas LCD periódicamente
modoStreamingLCD();

// enviar buffer de telemetría por BT si hay cliente
drainTeleBuffer(8);

// Comandos normales (si no estamos en menu)
handleSerialCommands();
handleBTCommands();

// detectar desconexión ELM
if(elmClient && elmConnected && !elmClient->isConnected()){
  Serial.println("ELM desconectado (detected). limpiando estado.");
```

```
    elmConnected = false;
    elmWriteChar = nullptr;
    elmNotifyChar = nullptr;
    if(elmClient) { elmClient->disconnect(); delete elmClient;
elmClient = nullptr; }
  }

  delay(10);
}
```