



# Manual de Prácticas: Construcción de una Red CAN Automotriz con Sensores y Conexión a GUI

*Para estudiantes de Ingeniería en Sistemas Automotrices*

**Autor: Ing. Sergio Josué Ortiz Hernández**

Benemérita Universidad Autónoma de Puebla  
Facultad de Ciencias de la Electrónica

20 de junio de 2025

# Índice

<b>1 Introducción</b>	<b>3</b>
<b>2 Práctica 1: Introducción al Protocolo CAN</b>	<b>3</b>
2.1 Importancia en Vehículos . . . . .	3
2.2 ¿Qué aprenderás? . . . . .	4
2.3 Fundamentos . . . . .	4
2.3.1 Estructura de una Trama CAN . . . . .	4
2.3.2 Ventajas de CAN en Vehículos . . . . .	5
2.3.3 ¿Por qué aprender CAN? . . . . .	5
2.4 Actividad . . . . .	6
2.5 Referencias . . . . .	7
<b>3 Práctica 2: Configuración de Nodos CAN: CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230)</b>	<b>7</b>
3.1 Importancia en Vehículos . . . . .	7
3.2 ¿Qué aprenderás? . . . . .	7
3.3 ¿Qué son CAN Autónomo y CAN Integrado? . . . . .	8
3.3.1 CAN Autónomo (Arduino con MCP2515) . . . . .	8
3.3.2 CAN Integrado (ESP32 con SN65HVD230) . . . . .	8
3.3.3 ¿Por qué ofrecemos ambas soluciones? . . . . .	9
3.4 CAN Autónomo (Arduino con MCP2515) - Opción 1 . . . . .	9
3.5 CAN Integrado (ESP32 con SN65HVD230) - Opción 2 . . . . .	11
3.6 Actividad . . . . .	13
3.7 Referencias . . . . .	14
<b>4 Práctica 3: Sensor de Efecto Hall (A3144)</b>	<b>15</b>
4.1 Importancia en Vehículos . . . . .	15
4.2 ¿Qué aprenderás? . . . . .	15
4.3 Explicación . . . . .	16
4.4 Ecuación . . . . .	16
4.5 Conexión . . . . .	16
4.6 Implementación . . . . .	17
4.7 Actividad . . . . .	18
4.8 Referencias . . . . .	19
<b>5 Práctica 4: Termistor (NTC 10kΩ)</b>	<b>19</b>
5.1 Importancia en Vehículos . . . . .	19
5.2 ¿Qué aprenderás? . . . . .	20
5.3 Explicación . . . . .	20
5.4 Ecuación de Steinhart-Hart . . . . .	20
5.5 Caracterización . . . . .	21
5.6 Implementación . . . . .	23
5.7 Actividad . . . . .	24
5.8 Referencias . . . . .	25
<b>6 Práctica 5: Sensor de Nivel de Combustible</b>	<b>25</b>

6.1	Importancia en Vehículos . . . . .	26
6.2	¿Qué aprenderás? . . . . .	27
6.3	Explicación . . . . .	27
6.4	Ecuación . . . . .	27
6.5	Implementación . . . . .	28
6.6	Actividad . . . . .	29
6.7	Referencias . . . . .	30
<b>7</b>	<b>Práctica 6: Integración de Sensores en una Red CAN y Visualización en GUI</b>	<b>31</b>
7.1	Importancia en Vehículos . . . . .	31
7.2	¿Qué aprenderás? . . . . .	32
7.3	Explicación . . . . .	32
7.4	Conexión . . . . .	32
7.5	Implementación . . . . .	33
7.6	Actividad . . . . .	37
7.7	Referencias . . . . .	39
<b>8</b>	<b>Práctica 7: Análisis de Tramas CAN con Analizador Lógico Saleae Logic 2</b>	<b>39</b>
8.1	Importancia en Vehículos . . . . .	39
8.2	¿Qué aprenderás? . . . . .	40
8.3	Explicación . . . . .	40
8.4	Conexión . . . . .	41
8.5	Implementación . . . . .	41
8.6	Actividad . . . . .	42
8.7	Referencias . . . . .	44
<b>9</b>	<b>Conclusión</b>	<b>44</b>
<b>10</b>	<b>Referencias</b>	<b>45</b>

## 1. Introducción

### ¿Qué aprenderás?

Construirás una red CAN automotriz que conecta sensores (RPM, temperatura, nivel de combustible) a una interfaz gráfica preconfigurada que emula un cuadro de instrumentos de un Kia Sorento 2019. Usarás Arduino o ESP32, un analizador lógico Saleae, y software gratuito (Arduino IDE, GUI preinstalada). No necesitas experiencia previa. Podrás elegir entre dos soluciones CAN: CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230).

### Objetivos:

- Entender el protocolo CAN y su importancia en vehículos.
- Configurar nodos CAN con transceptores SN65HVD230 o MCP2515, eligiendo entre CAN Autónomo o CAN Integrado.
- Implementar y caracterizar sensores automotrices.
- Analizar tramas CAN con Saleae Logic.
- Conectar la red a una GUI preconfigurada para visualizar datos en tiempo real.

### Requisitos:

- **Hardware:** Arduino Uno, ESP32, SN65HVD230, MCP2515, sensor A3144, termistor NTC ( $10k\Omega$ ), sensor de nivel de combustible ( $100\text{--}900\Omega$ ), resistencias ( $120\Omega$ ,  $10k\Omega$ ), cables, analizador lógico Saleae.
- **Software:** Arduino IDE, bibliotecas CAN, MCP\_CAN, GUI preinstalada (GUI Cuadro de instrumentos.exe), Saleae Logic.

## 2. Práctica 1: Introducción al Protocolo CAN

**Objetivo:** Comprender los fundamentos del protocolo CAN, su estructura, y su importancia en sistemas automotrices mediante actividades prácticas y análisis de ejemplos.

### 2.1. Importancia en Vehículos

El protocolo CAN (Controller Area Network) es el **sistema nervioso** de los vehículos modernos. Permite que múltiples dispositivos, como sensores, unidades de control electrónico (ECUs), y el tablero, se comuniquen de forma rápida y confiable a través de un solo par de cables (bus CAN). Esto reduce la cantidad de cables, el peso, y los costos en un auto, mientras asegura una comunicación robusta incluso en entornos ruidosos, como cerca de un motor. Por ejemplo, CAN permite que un sensor de velocidad envíe datos al velocímetro en tiempo real o que un termistor informe la temperatura del motor a la ECU para ajustar el rendimiento.

## 2.2. ¿Qué aprenderás?

- Qué es el protocolo CAN y cómo funciona en vehículos.
- La estructura de una trama CAN y el propósito de cada campo.
- Cómo CAN gestiona la comunicación entre dispositivos sin colisiones.
- Ejemplos prácticos de uso de CAN en autos.
- Analizar y diseñar tramas CAN básicas.

## 2.3. Fundamentos

El protocolo CAN, desarrollado por Bosch en los años 80 (estandarizado en ISO 11898-2), es un sistema de comunicación en serie diseñado para entornos en tiempo real, como los vehículos. Permite que múltiples dispositivos (nodos) compartan datos a través de un **bus CAN**, que consta de dos líneas: **CAN H** (alto) y **CAN L** (bajo). Estas líneas transmiten señales diferenciales, lo que hace que CAN sea resistente al ruido eléctrico, algo común en autos.

Imagina CAN como una **conversación en grupo** en la que todos los dispositivos están conectados a un solo cable (el bus). Cada dispositivo puede **hablar** (enviar mensajes) o **escuchar** (recibir mensajes) sin necesidad de un controlador central. Lo especial de CAN es que, si dos dispositivos intentan hablar al mismo tiempo, el mensaje con mayor prioridad (basado en su ID) **gana** automáticamente, evitando colisiones, gracias a un mecanismo llamado **arbitraje**.

### 2.3.1. Estructura de una Trama CAN

Los datos en CAN se envían en \*\*tramas\*\* (mensajes) con una estructura específica. Cada trama tiene varios campos, como se muestra en la figura 1. Aquí explicamos cada uno con un ejemplo práctico (por ejemplo, enviar un valor de RPM de 1200):

- **SOF (Start of Frame)**: Un bit que marca el inicio de la trama (siempre 0).
- **ID (Identificador)**: Un número (11 o 29 bits) que indica el tipo de mensaje y su prioridad. Por ejemplo, ID 0x100 para RPM. Un ID más bajo tiene mayor prioridad.
- **RTR (Remote Transmission Request)**: Indica si la trama es un mensaje de datos (0) o una solicitud remota (1). Usamos 0 para datos.
- **DLC (Data Length Code)**: Especifica cuántos bytes de datos lleva la trama (0 a 8). Por ejemplo, 2 bytes para RPM.
- **Datos**: Hasta 8 bytes de información. Para RPM = 1200 (0x04B0 en hexadecimal), los datos serían [0x04, 0xB0].
- **CRC (Cyclic Redundancy Check)**: Un código para detectar errores en la transmisión.
- **ACK (Acknowledge)**: Un bit que confirma que al menos un nodo recibió la trama correctamente.
- **EOF (End of Frame)**: 7 bits que marcan el fin de la trama.

Cuadro 1: Ejemplo de una trama CAN para RPM = 1200.

Campo	Valor (Ejemplo)
SOF	0
ID	0x100
RTR	0
DLC	2
Datos	0x04 0xB0 (1200 en hexadecimal)
CRC	(Calculado automáticamente)
ACK	1 (Confirmado por receptor)
EOF	1111111

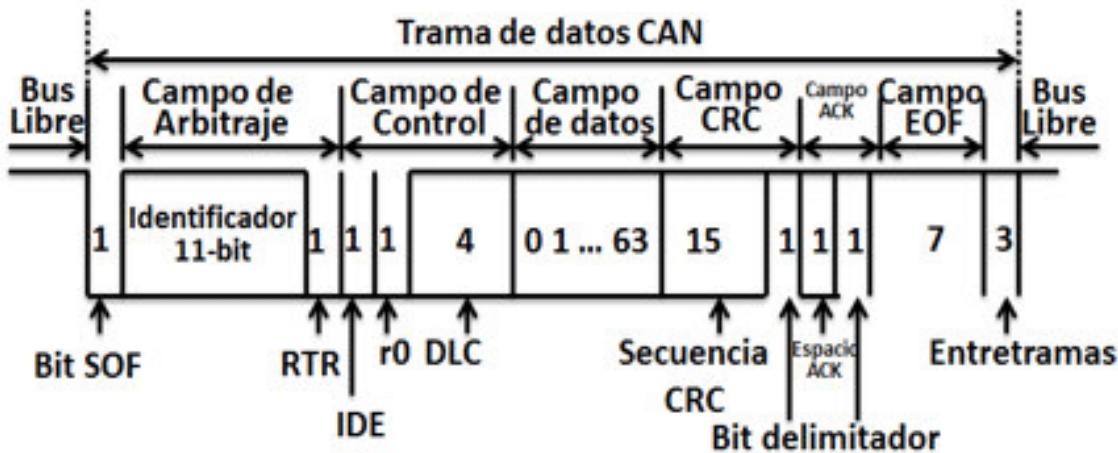


Figura 1: Estructura de una trama CAN con campos etiquetados (SOF, ID, RTR, DLC, Datos, CRC, ACK, EOF).

### 2.3.2. Ventajas de CAN en Vehículos

- Reducción de cableado:** Un solo bus CAN reemplaza docenas de cables, haciendo los autos más ligeros y económicos.
- Tolerancia a fallos:** CAN detecta errores (usando CRC) y reintenta transmisiones automáticamente.
- Prioridad:** Los mensajes importantes (como una advertencia de frenos) tienen prioridad sobre otros (como el nivel de combustible).
- Escalabilidad:** Puedes añadir nuevos sensores o ECUs al bus sin rediseñar el sistema.

Ejemplo: En un Kia Sorento 2019, el sensor de velocidad de las ruedas envía datos por CAN al sistema ABS (ID 0x200) para evitar derrapes, mientras el termistor del motor envía la temperatura (ID 0x101) a la ECU para ajustar la inyección de combustible.

### 2.3.3. ¿Por qué aprender CAN?

Como estudiante de sistemas automotrices, entender CAN es clave porque es el estándar de comunicación en casi todos los vehículos modernos. En este manual, usarás CAN para

conectar sensores (RPM, temperatura, nivel de combustible) a una red y visualizar datos en una GUI, simulando un cuadro de instrumentos real. Aprenderás a configurar nodos CAN (Práctica 2), leer sensores (Prácticas 3–5), y analizar datos (Prácticas 6–8), habilidades esenciales para diagnosticar y diseñar sistemas automotrices.

## 2.4. Actividad

### Actividades

#### 1. Analizar una trama CAN:

- Observa la figura 1 y la tabla 1. Identifica cada campo de la trama (SOF, ID, RTR, DLC, Datos, CRC, ACK, EOF) y explica brevemente su propósito en una tabla (por ejemplo, “ID: Define el tipo de mensaje y prioridad”).
- Supón que recibes una trama con ID 0x101, DLC = 2, y datos [0x00, 0x19]. ¿Qué valor representa si es una temperatura en °C? (Pista: Convierte [0x00, 0x19] a decimal).

#### 2. Investigación guiada:

- Investiga un ejemplo real de uso de CAN en un vehículo (por ejemplo, cómo el sensor de velocidad informa al ABS o cómo el termistor envía datos al tablero). Responde:
  - ¿Qué dispositivos se comunican?
  - ¿Qué tipo de datos envían?
  - ¿Por qué es importante esta comunicación?
- Usa fuentes confiables, como manuales técnicos o sitios web de fabricantes (por ejemplo, Bosch, Texas Instruments).

#### 3. Diseñar una trama CAN:

- Escribe un pseudocódigo en Arduino IDE (sin hardware) que envíe una trama CAN con:
  - ID = 0x103
  - Datos = Nivel de combustible de 75 % (1 byte, 0x4B en hexadecimal).
  - DLC = 1
- Usa la estructura de los códigos de la Práctica 2 como referencia (por ejemplo, CAN.beginPacket() para ESP32 o CAN.sendMsgBuf() para Arduino). Explica qué hace cada línea.

#### 4. Opcional: Explorar un log CAN:

- Si tienes acceso a un simulador CAN online (por ejemplo, <https://www.csselectronics.com/pages/can-bus-simulator>) o a un log CAN proporcionado por tu instructor, analiza una captura de tramas. Identifica al menos dos IDs diferentes y describe qué podrían representar (por ejemplo, RPM, temperatura).

## 2.5. Referencias

- ISO 11898-2:2003, *Controller Area Network (CAN) - Part 2: High-speed medium access unit*, International Organization for Standardization, 2003.
- Robert Bosch GmbH, *CAN Specification Version 2.0*, 1991, <https://www.nxp.com/docs/en/reference-manual/CAN2spec.pdf>.
- Lawrenz, W., *CAN System Engineering: From Theory to Practical Applications*, Springer, 2013, ISBN: 978-1447154181.
- *Introduction to the Controller Area Network (CAN)*, Texas Instruments, 2016, <https://www.ti.com/lit/an/sloa101b/sloa101b.pdf>.

## 3. Práctica 2: Configuración de Nodos CAN: CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230)

**Objetivo:** Configurar nodos CAN implementando una solución a elegir entre CAN Autónomo (usando Arduino con MCP2515 como controlador externo) o CAN Integrado (usando ESP32 con SN65HVD230), programando un transmisor y un receptor para observar la comunicación CAN bidireccional. Puedes implementar una solución o ambas según el hardware disponible.

### 3.1. Importancia en Vehículos

Un nodo CAN es un dispositivo (como un sensor o una ECU) conectado al bus CAN, que envía o recibe datos para formar la red de comunicación del vehículo. En esta práctica, implementarás una solución CAN a tu elección: **CAN Autónomo** (usando Arduino Uno con el módulo MCP2515 como controlador CAN externo) o **CAN Integrado** (usando el controlador CAN integrado del ESP32 con el transceptor SN65HVD230). Ambas soluciones son equivalentes en funcionalidad para las prácticas de este manual, pero difieren en hardware y enfoque técnico, lo que permite adaptarse a diferentes niveles de experiencia y disponibilidad de equipo. Configurar un transmisor y un receptor te ayudará a entender cómo los dispositivos intercambian datos en tiempo real, una habilidad clave para diseñar y diagnosticar sistemas CAN automotrices. Consulta la siguiente subsección para entender las diferencias entre ambas soluciones.

### 3.2. ¿Qué aprenderás?

- Conectar hardware CAN usando Arduino con MCP2515 (CAN Autónomo) o ESP32 con SN65HVD230 (CAN Integrado).
- Programar un nodo transmisor para enviar mensajes CAN.
- Programar un nodo receptor para leer y mostrar mensajes CAN en el monitor serie.
- Verificar la comunicación bidireccional entre nodos en la solución elegida.

### 3.3. ¿Qué son CAN Autónomo y CAN Integrado?

En esta práctica, tienes la opción de elegir entre dos formas de implementar nodos CAN: **CAN Autónomo** y **CAN Integrado**. Ambas cumplen con el estándar CAN (ISO 11898-2) y son compatibles con las prácticas posteriores, pero usan diferentes hardware y enfoques técnicos. A continuación, te explicamos cada una para que entiendas sus diferencias y por qué se ofrecen ambas opciones.

#### 3.3.1. CAN Autónomo (Arduino con MCP2515)

En la solución **CAN Autónomo**, usas un microcontrolador **Arduino Uno** junto con un módulo externo **MCP2515**, que actúa como controlador CAN. El módulo MCP2515 incluye tanto el controlador CAN (que gestiona el protocolo) como un transceptor (que convierte señales digitales en señales diferenciales para el bus CAN). El Arduino se comunica con el MCP2515 a través de una interfaz **SPI** (Serial Peripheral Interface), enviando y recibiendo datos CAN.

##### Ventajas:

- **Fácil de usar:** El Arduino Uno es una plataforma conocida y sencilla, ideal para principiantes en electrónica.
- **Económico:** El módulo MCP2515 es asequible (100–200 MXN) y fácil de conseguir.
- **Ampliamente soportado:** La biblioteca MCP\_CAN es robusta y está bien documentada, con muchos ejemplos disponibles.

##### Desventajas:

- **Dependencia de SPI:** La comunicación SPI entre el Arduino y el MCP2515 puede ser un cuello de botella en aplicaciones muy rápidas.
- **Limitaciones del Arduino:** El Arduino Uno tiene menos potencia de procesamiento y carece de funciones avanzadas como Wi-Fi o Bluetooth.

**Cuándo elegirlo:** Si eres principiante, tienes un Arduino Uno disponible, o prefieres una configuración sencilla con hardware común en laboratorios educativos.

#### 3.3.2. CAN Integrado (ESP32 con SN65HVD230)

En la solución **CAN Integrado**, usas un microcontrolador **ESP32**, que tiene un controlador CAN integrado (llamado TWAI, "Two-Wire Automotive Interface"). Este controlador gestiona el protocolo CAN directamente en el chip del ESP32, pero necesitas un transceptor externo, como el **SN65HVD230**, para conectar el ESP32 al bus CAN, ya que el transceptor convierte las señales digitales del ESP32 en señales diferenciales (CAN H y CAN L).

##### Ventajas:

- **Eficiencia:** Al tener el controlador CAN integrado, no depende de SPI, lo que reduce la latencia y simplifica el hardware.
- **Potencia del ESP32:** El ESP32 es más rápido y versátil, con soporte para Wi-Fi y Bluetooth, útil para proyectos avanzados.
- **Moderna:** Refleja sistemas CAN más recientes usados en la industria automotriz.

**Desventajas:**

- **Configuración más técnica:** Conectar el transceptor SN65HVD230 requiere un poco más de cuidado (por ejemplo, usar 3.3V en lugar de 5V).
- **Costo ligeramente mayor:** El ESP32 ( 150–300 MXN) más el transceptor ( 50–100 MXN) puede ser más caro que un módulo MCP2515.

**Cuándo elegirlo:** Si tienes experiencia previa, acceso a un ESP32, o quieres explorar una solución más moderna con potencial para funciones adicionales (como conexión inalámbrica en futuros proyectos).

### 3.3.3. ¿Por qué ofrecemos ambas soluciones?

Ofrecemos **CAN Autónomo** y **CAN Integrado** para darte flexibilidad según tu nivel de experiencia, el hardware disponible en tu laboratorio, y tus objetivos de aprendizaje. Ambas soluciones:

- Son compatibles con el estándar CAN (500 kbps, High Speed).
- Funcionan con las prácticas posteriores (3 a 8), ya que los códigos y conexiones están diseñados para ser equivalentes.
- Permiten interoperabilidad: Puedes usar un nodo Arduino con MCP2515 y otro ESP32 con SN65HVD230 en la misma red CAN.

Por ejemplo, si solo tienes Arduinos, elige **CAN Autónomo** por su simplicidad. Si tienes ESP32 y quieres una solución más eficiente o planeas proyectos más complejos, elige **CAN Integrado**. Si tienes ambos, puedes experimentar con las dos y compararlas, o incluso combinarlas en la misma red (por ejemplo, un transmisor en Arduino y un receptor en ESP32). La elección es tuya, y este manual proporciona códigos y conexiones para ambas opciones para que no te limite.

## 3.4. CAN Autónomo (Arduino con MCP2515) - Opción 1

Esta solución utiliza un Arduino Uno con el módulo MCP2515 como controlador CAN externo, conectado vía SPI, junto con un transceptor (integrado en el módulo) para comunicarse con el bus CAN.

### Conexión con Arduino y MCP2515

- Conecta el módulo MCP2515 al Arduino Uno (para transmisor y receptor):
  - VCC a 5V, GND a tierra.
  - CS a pin 10, MOSI a pin 11, MISO a pin 12, SCK a pin 13.
  - CAN H y CAN L al bus CAN (conecta CAN H y CAN L de ambos nodos).
- Coloca resistencias de terminación ( $120\Omega$ ) en los extremos del bus.

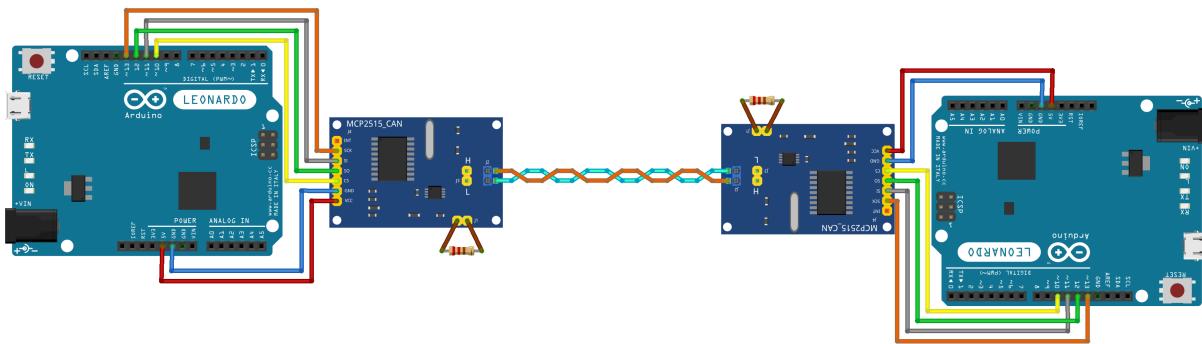


Figura 2: Esquemático de conexión del MCP2515 con Arduino.

## Código del Transmisor (Arduino con MCP2515)

1. Instala la biblioteca MCP\_CAN en Arduino IDE (Tools >Manage Libraries).
2. Carga este código en un Arduino (nodo transmisor):

```

1 #include <SPI.h>
2 #include <mcp_can.h>
3 MCP_CAN CAN(10); // Pin CS
4 void setup() {
5     Serial.begin(9600);
6     while (CAN_OK != CAN.begin(CAN_500KBPS)) {
7         Serial.println("Error al iniciar CAN, reintentando...");
8         delay(100);
9     }
10    Serial.println("Transmisor CAN iniciado");
11 }
12 void loop() {
13     unsigned char data[8] = {0x01, 0x02, 0x03, 0x04, 0x05, 0
14         x06, 0x07, 0x08};
15     CAN.sendMsgBuf(0x100, 0, 8, data);
16     Serial.println("Mensaje CAN enviado");
17     delay(1000);
18 }
```

## Código del Receptor (Arduino con MCP2515)

1. Carga este código en otro Arduino (nodo receptor):

```

1 #include <SPI.h>
2 #include <mcp_can.h>
3 MCP_CAN CAN(10); // Pin CS
4 void setup() {
5     Serial.begin(9600);
6     while (CAN_OK != CAN.begin(CAN_500KBPS)) {
```

```

7     Serial.println("Error al iniciar CAN, reintentando...");
8     delay(100);
9 }
10 Serial.println("Receptor CAN iniciado");
11 }
12 void loop() {
13     unsigned long rxId;
14     unsigned char len = 0;
15     unsigned char rxBuf[8];
16     if (CAN_MSGAVAIL == CAN.checkReceive()) {
17         CAN.readMsgBuf(&len, rxBuf);
18         rxId = CAN.getCanId();
19         if (rxId == 0x100) {
20             Serial.print("Mensaje recibido, ID: 0x");
21             Serial.print(rxId, HEX);
22             Serial.print(", Datos:");
23             for (int i = 0; i < len; i++) {
24                 Serial.print(rxBuf[i], HEX);
25                 Serial.print(" ");
26             }
27             Serial.println();
28         }
29     }
30 }
```

### 3.5. CAN Integrado (ESP32 con SN65HVD230) - Opción 2

Esta solución utiliza el controlador CAN integrado (TWAI) del ESP32, junto con el transceptor SN65HVD230 para convertir las señales digitales en señales diferenciales para el bus CAN.

#### Conexión con ESP32 y SN65HVD230

- Conecta el SN65HVD230 al ESP32 (para transmisor y receptor):
  - VCC a 3.3V, GND a tierra.
  - TX a GPIO 5, RX a GPIO 4.
  - CAN H y CAN L al bus CAN (conecta CAN H y CAN L de ambos nodos).
- Coloca resistencias de terminación ( $120\Omega$ ) en los extremos del bus.

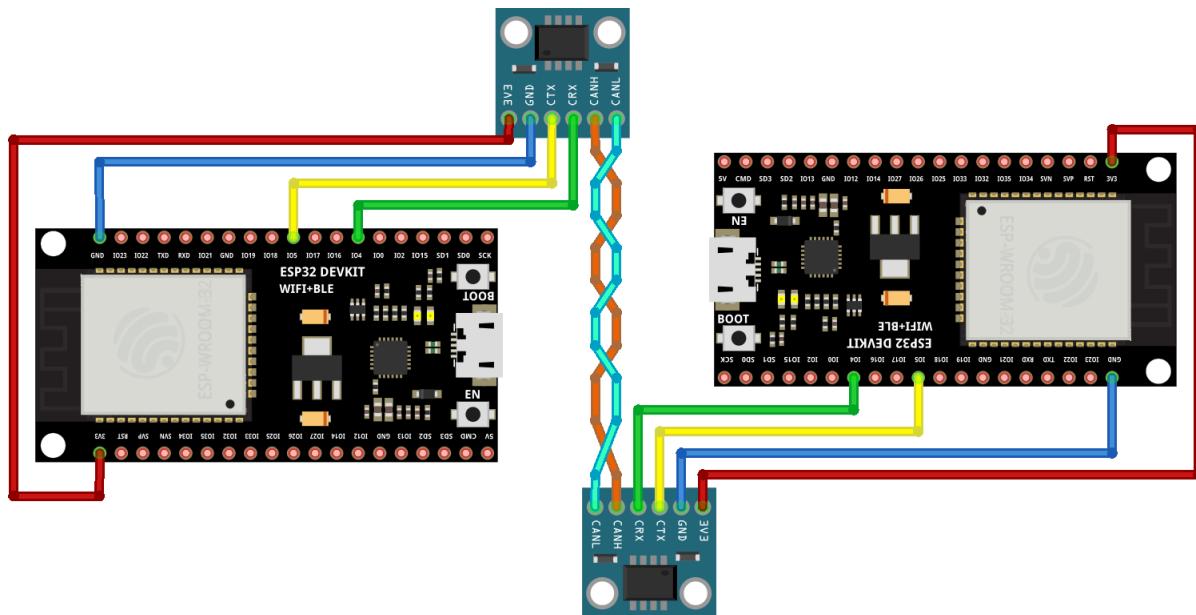


Figura 3: Esquemático de conexión del SN65HVD230 con ESP32.

## Código del Transmisor (ESP32 con SN65HVD230)

1. Instala la biblioteca CAN en Arduino IDE (Tools >Manage Libraries).
2. Carga este código en un ESP32 (nodo transmisor):

```

1 #include <CAN.h>
2 void setup() {
3     Serial.begin(9600);
4     CAN.setPins(4, 5); // RX = GPIO4, TX = GPIO5
5     if (CAN.begin(500000)) { // 500 kbps
6         Serial.println("Transmisor CAN iniciado");
7     } else {
8         Serial.println("Error al iniciar CAN");
9     }
10 }
11 void loop() {
12     uint8_t data[8] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0
13         x07, 0x08};
14     CAN.beginPacket(0x100); // ID 0x100
15     CAN.write(data, 8);
16     CAN.endPacket();
17     Serial.println("Mensaje CAN enviado");
18     delay(1000);
19 }
```

## Código del Receptor (ESP32 con SN65HVD230)

1. Carga este código en otro ESP32 (nodo receptor):

```

1 #include <CAN.h>
2 void setup() {
3     Serial.begin(9600);
4     CAN.setPins(4, 5); // RX = GPIO4, TX = GPIO5
5     if (CAN.begin(500000)) { // 500 kbps
6         Serial.println("Receptor CAN iniciado");
7     } else {
8         Serial.println("Error al iniciar CAN");
9     }
10 }
11 void loop() {
12     if (CAN.parsePacket()) {
13         long unsigned int rxId = CAN.packetId();
14         int len = CAN.available();
15         uint8_t rxBuf[8];
16         for (int i = 0; i < len; i++) {
17             rxBuf[i] = CAN.read();
18         }
19         if (rxId == 0x100) {
20             Serial.print("Mensaje recibido, ID: 0x");
21             Serial.print(rxId, HEX);
22             Serial.print(", Datos: ");
23             for (int i = 0; i < len; i++) {
24                 Serial.print(rxBuf[i], HEX);
25                 Serial.print(" ");
26             }
27             Serial.println();
28         }
29     }
30 }
```

### 3.6. Actividad

## Actividades

1. Escoge una solución CAN:

- Selecciona CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230) según el hardware disponible o tu preferencia.
- Si tienes ambas plataformas, considera implementar las dos soluciones para compararlas o combinarlas (por ejemplo, un transmisor en Arduino y un receptor en ESP32).

2. Configura los nodos:

- Conecta un nodo transmisor y un nodo receptor al bus CAN, siguiendo las instrucciones de conexión de la solución elegida (figura 2 para CAN Autónomo o 3 para CAN Integrado).
- Asegúrate de usar resistencias de terminación de  $120\Omega$  en los extremos del bus CAN.

### 3. Carga los códigos:

- Carga el código de transmisor en un Arduino o ESP32 y el código de receptor en otro, según la solución seleccionada.
- Verifica que el transmisor envíe mensajes con ID 0x100 y datos [0x01, 0x02, ..., 0x08].

### 4. Verifica la comunicación:

- Abre el monitor serie del receptor (9600 baudios) y confirma que los mensajes CAN se reciben correctamente (ID 0x100, datos correctos).
- Si no se reciben mensajes, revisa las conexiones, las resistencias de terminación y la inicialización del CAN.

### 5. Dibuja el esquemático:

- Representa la conexión realizada en un diagrama claro, basado en la figura 2 (CAN Autónomo) o 3 (CAN Integrado).
- Etiqueta los pines y componentes clave (por ejemplo, CAN H, CAN L, VCC, GND).

### 6. Opcional: Prueba interoperabilidad:

- Si tienes ambas plataformas, configura un transmisor en Arduino con MCP2515 y un receptor en ESP32 con SN65HVD230 (o viceversa).
- Verifica que los mensajes se reciban correctamente, demostrando la compatibilidad entre soluciones CAN.
- Reflexiona: ¿Qué ventajas tiene combinar diferentes plataformas en una red CAN?

## 3.7. Referencias

- Texas Instruments, *SN65HVD230 CAN Transceiver Datasheet*, 2020, <https://www.ti.com/lit/ds/symlink/sn65hvd230.pdf>.
- Microchip Technology, *MCP2515 Stand-Alone CAN Controller Datasheet*, 2010, <https://ww1.microchip.com/downloads/en/DeviceDoc/20001801J.pdf>.
- *ESP32-CAN Library*, GitHub, <https://github.com/miwagner/ESP32-Arduino-CAN>.
- *MCP\_CAN Library*, GitHub, [https://github.com/coryjfowler/MCP\\_CAN\\_lib](https://github.com/coryjfowler/MCP_CAN_lib).

- *Controller Area Network (CAN) Tutorial*, National Instruments, [https://www.ni.com/docs/en-US/bundle/labview/page/lvhowto/can\\_tutorial.html](https://www.ni.com/docs/en-US/bundle/labview/page/lvhowto/can_tutorial.html).

## 4. Práctica 3: Sensor de Efecto Hall (A3144)

**Objetivo:** Configurar un sensor de efecto Hall A3144 para medir RPM y enviar datos por CAN, usando la solución CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230) elegida en la Práctica 2.

### 4.1. Importancia en Vehículos

Los sensores de efecto Hall son esenciales en los vehículos modernos porque miden la velocidad de rotación (RPM) de componentes clave, como el cigüeñal, el árbol de levas, o las ruedas. Estos datos son fundamentales para varios sistemas del auto:

- **Control del motor:** El sensor de cigüeñal mide las RPM del motor y envía esta información a la unidad de control electrónico (ECU). La ECU usa estos datos para ajustar la sincronización del encendido y la inyección de combustible, asegurando que el motor funcione de manera eficiente y sin fallos. Por ejemplo, en un Kia Sorento 2019, el sensor de cigüeñal envía datos por CAN (ID 0x102) para que la ECU optimice el rendimiento.
- **Sistema de frenos ABS:** Los sensores de efecto Hall en las ruedas miden la velocidad de rotación de cada una. Si una rueda gira más lento (indicando un posible bloqueo), el sistema ABS usa estos datos para liberar presión en el freno, evitando derrapes y mejorando la seguridad.
- **Velocímetro:** La velocidad de rotación de las ruedas se convierte en velocidad del vehículo, que se muestra en el tablero. Esto permite al conductor saber a qué velocidad viaja.
- **Transmisión:** Los datos de RPM ayudan a la transmisión automática a decidir cuándo cambiar de marcha, mejorando la suavidad y el consumo de combustible.

Imagina el sensor de efecto Hall como un "contador de giros" que detecta un imán pegado a un componente giratorio, como el cigüeñal o una rueda. Cada vez que el imán pasa por el sensor, genera un pulso eléctrico. Al contar estos pulsos, el sensor calcula las RPM. Su diseño simple y robusto lo hace ideal para entornos automotrices, donde debe soportar vibraciones, temperaturas extremas (de -40°C a 150°C), y ruido eléctrico. Sin estos sensores, sistemas críticos como el motor o el ABS no funcionarían correctamente, lo que podría causar fallos mecánicos o accidentes.

### 4.2. ¿Qué aprenderás?

- Cómo conectar un sensor de efecto Hall A3144 para medir RPM.
- Contar pulsos magnéticos para calcular RPM.
- Enviar datos de RPM por CAN usando la solución elegida (Autónomo o Integrado).

### 4.3. Explicación

El sensor A3144 detecta un imán en un componente giratorio, como un eje. Cuando el imán pasa frente al sensor, genera un pulso eléctrico (de alto a bajo). Contando estos pulsos en un segundo, puedes calcular las revoluciones por minuto (RPM). En esta práctica, usarás un imán en un eje giratorio (por ejemplo, un motor pequeño o un ventilador manual) y asumirás que un pulso equivale a una revolución.

### 4.4. Ecuación

$$\text{RPM} = \text{Pulsos por segundo} \times 60$$

Ejemplo: Si cuentas 10 pulsos en un segundo con un solo imán, las RPM son  $10 \times 60 = 600$ .

### 4.5. Conexión

#### Conexión del Sensor A3144

- **Materiales:** Sensor A3144, imán, Arduino Uno o ESP32, cables.
- **Conexión:**
  - VCC del A3144 a 5V (Arduino) o 3.3V (ESP32).
  - GND del A3144 a tierra.
  - Salida del A3144 a pin 2 (Arduino) o GPIO 5 (ESP32).
- **Configuración:** Pega un imán en un eje giratorio (por ejemplo, un motor pequeño o un ventilador girado a mano).
- **Nota:** Asegúrate de que el imán pase cerca del sensor (menos de 1 cm) en cada giro.

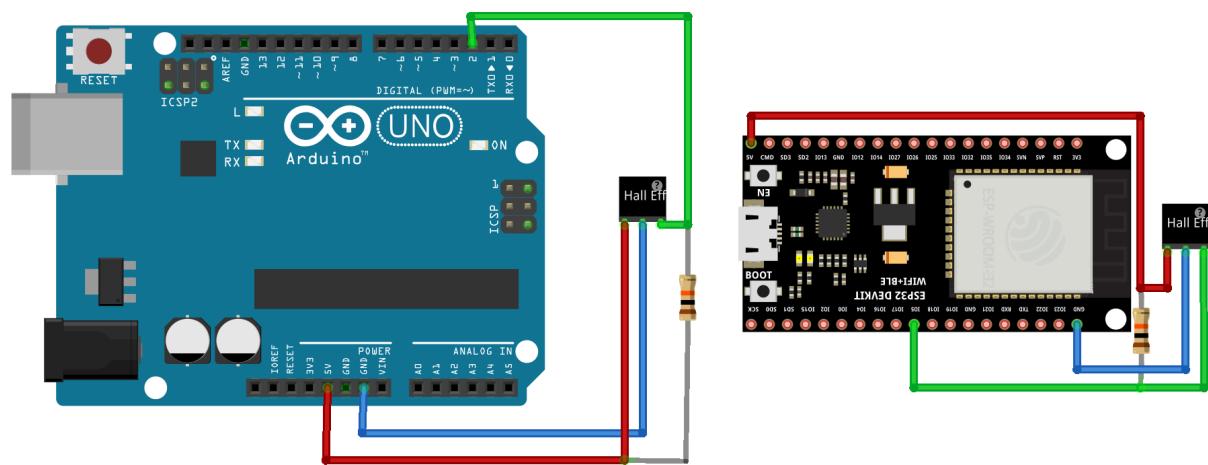


Figura 4: Conexión simplificada del sensor de efecto Hall A3144 con Arduino o ESP32.

## 4.6. Implementación

Código (ESP32 con SN65HVD230)

```

1 #include <CAN.h>
2 const int hallPin = 5;
3 volatile int pulses = 0;
4 void IRAM_ATTR countPulse() {
5     pulses++;
6 }
7 void setup() {
8     pinMode(hallPin, INPUT);
9     attachInterrupt(digitalPinToInterrupt(hallPin),
10                     countPulse, RISING);
11    Serial.begin(9600);
12    CAN.setPins(4, 5); // RX = GPIO4, TX = GPIO5
13    if (CAN.begin(500000)) {
14        Serial.println("Nodo RPM iniciado");
15    } else {
16        Serial.println("Error al iniciar CAN");
17    }
18 }
19 void loop() {
20     pulses = 0;
21     delay(1000); // Contar pulsos durante 1 segundo
22     int rpm = pulses * 60; // 1 iman
23     Serial.print("S:");
24     uint8_t data[2] = {(uint8_t)(rpm >> 8), (uint8_t)(rpm & 0xFF)};
25     CAN.beginPacket(0x102);
26     CAN.write(data, 2);
27     CAN.endPacket();
}

```

Código (Arduino con MCP2515)

```

1 #include <SPI.h>
2 #include <mcp_can.h>
3 MCP_CAN CAN(10); // Pin CS
4 const int hallPin = 2;
5 volatile int pulses = 0;
6 void countPulse() {
7     pulses++;
8 }
9 void setup() {
10     pinMode(hallPin, INPUT);
11     attachInterrupt(digitalPinToInterrupt(hallPin),
12                     countPulse, RISING);
13     Serial.begin(9600);
14     while (CAN_OK != CAN.begin(CAN_500KBPS)) {
}

```

```

4     Serial.println("Error al iniciar CAN, reintentando...");
5         );
6     delay(100);
7 }
8 Serial.println("Nodo RPM iniciado");
9 }
10 void loop() {
11     pulses = 0;
12     delay(1000); // Contar pulsos durante 1 segundo
13     int rpm = pulses * 60; // 1 iman
14     Serial.print("S:");
15     Serial.println(rpm);
16     unsigned char data[2] = {(unsigned char)(rpm >> 8), (
17         unsigned char)(rpm & 0xFF)};
18     CAN.sendMsgBuf(0x102, 0, 2, data);
19 }
20 }
```

## 4.7. Actividad

### Actividades

#### 1. Configura el sensor A3144:

- Conecta el sensor A3144 al Arduino o ESP32 según la solución CAN elegida (Autónomo o Integrado), siguiendo el esquemático de la figura 4.
- Pega un imán a un eje giratorio (por ejemplo, un motor pequeño o un ventilador girado manualmente) y asegúrate de que pase cerca del sensor ( $\pm 1$  cm) en cada giro.

#### 2. Carga el código:

- Carga el código correspondiente (ESP32 o Arduino) en tu microcontrolador.
- Verifica que el código inicialice correctamente el bus CAN y configure la interrupción para contar pulsos.

#### 3. Mide las RPM:

- Gira el eje con el imán a diferentes velocidades (por ejemplo, lento, medio, rápido).
- Observa los valores de RPM en el monitor serie (9600 baudios, formato S:valor).
- Asegúrate de que los datos se envíen por CAN con ID 0x102 (esto se verificará en prácticas posteriores).

#### 4. Valida los cálculos:

- Usa la ecuación  $RPM = \text{Pulsos por segundo} \times 60$  para verificar manualmente las RPM medidas. Por ejemplo, si cuentas 5 pulsos en 1 segundo, calcula:  $5 \times 60 = 300$  RPM.

- Compara los valores calculados manualmente con los mostrados en el monitor serie.

### 5. Reflexiona:

- ¿Cómo usaría la ECU los datos de RPM para controlar el motor o el sistema ABS en un vehículo real?
- ¿Qué factores (como la distancia del imán o el ruido eléctrico) podrían afectar la precisión del sensor A3144 en un entorno automotriz?

### 6. Opcional: Experimenta con el sensor:

- Si tienes acceso a un motor con velocidad controlada, varía la velocidad y registra las RPM medidas.
- Analiza cómo el número de imanes en el eje afecta los cálculos de RPM. Por ejemplo, si usas 2 imanes, ¿cómo ajustarías la ecuación?

## 4.8. Referencias

- *Hall Effect Sensor A3144 Datasheet*, Allegro MicroSystems, <https://www.allegromicro.com/en/products/sense/magnetic-sensors/hall-effect-sensors/a3144>.
- *Using Hall Effect Sensors for RPM Measurement*, SparkFun, <https://learn.sparkfun.com/tutorials/hall-effect-sensor-hookup-guide>.
- *Hall Effect Sensors in Automotive Applications*, Allegro MicroSystems, <https://www.allegromicro.com/en/insights-and-innovations/technical-documents/hall-effect-sensor-ics/automotive-applications>.

## 5. Práctica 4: Termistor (NTC 10kΩ)

**Objetivo:** Configurar y caracterizar un termistor NTC de 10kΩ para medir temperatura y enviar datos por CAN, usando la solución CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230) elegida en la Práctica 2.

### 5.1. Importancia en Vehículos

Los termistores de coeficiente de temperatura negativo (NTC) son sensores clave en los vehículos modernos debido a su alta sensibilidad, bajo costo y capacidad para operar en entornos extremos (de -40°C a 150°C). En automóviles, los termistores NTC, comúnmente con una resistencia nominal de 10kΩ a 25°C (aunque pueden variar de 100Ω a 100kΩ según el fabricante y la aplicación), se utilizan para monitorear temperaturas críticas en varios sistemas:

- **Control del motor:** Los termistores miden la temperatura del refrigerante o del aceite del motor. Por ejemplo, en un Kia Sorento 2019, el sensor de temperatura del refrigerante (ID 0x101 en CAN) envía datos a la unidad de control electrónico (ECU) para ajustar la inyección de combustible y la sincronización del encendido, optimizando la combustión

y reduciendo emisiones. Si el refrigerante está a 90°C, la ECU puede activar el ventilador de enfriamiento para evitar el sobrecalentamiento, que podría dañar el motor.

- **Vehículos eléctricos (EV):** En los EVs, los termistores monitorean la temperatura de las baterías (generalmente entre 20°C y 40°C) para prevenir el sobrecalentamiento durante la carga o descarga, lo que prolonga la vida útil de la batería y evita fallos peligrosos como el **thermal runaway**.
- **Sistemas de climatización (HVAC):** Los termistores regulan la temperatura del habitáculo, asegurando el confort de los pasajeros con un consumo energético mínimo, especialmente en EVs donde la eficiencia es crítica.
- **Otros usos:** Monitorean la temperatura del aire de admisión, el combustible, los asientos calefactados o el convertidor catalítico (que puede alcanzar 600°C), asegurando un rendimiento óptimo y la seguridad del vehículo.

Imagina el termistor como un **termómetro electrónico** que traduce la temperatura en cambios de resistencia eléctrica. Su precisión y robustez lo hacen ideal para entornos automotrices, donde debe soportar vibraciones, humedad y temperaturas extremas. Sin termistores, sistemas como el motor, las baterías o el climatizador no podrían operar de manera eficiente ni segura, lo que podría resultar en fallos mecánicos, mayor consumo de combustible o incomodidad para los pasajeros.

## 5.2. ¿Qué aprenderás?

- Cómo funciona un termistor NTC de 10kΩ en aplicaciones automotrices.
- Caracterizar la resistencia del termistor para calcular temperatura.
- Usar la ecuación de Steinhart-Hart para obtener mediciones precisas.
- Enviar datos de temperatura por CAN usando la solución elegida (Autónomo o Integrado).

## 5.3. Explicación

Un termistor NTC reduce su resistencia al aumentar la temperatura. Por ejemplo, un termistor de 10kΩ a 25°C puede tener 2kΩ a 75°C o 30kΩ a 0°C. Esta relación no lineal entre resistencia y temperatura requiere una ecuación precisa, como la de Steinhart-Hart, para convertir la resistencia medida en una temperatura exacta. En esta práctica, configurarás el termistor en un divisor de voltaje, medirás su resistencia en diferentes temperaturas, y usarás la ecuación de Steinhart-Hart para calcular la temperatura, simulando cómo un vehículo monitorea, por ejemplo, el refrigerante del motor.

## 5.4. Ecuación de Steinhart-Hart

La ecuación de Steinhart-Hart modela la relación no lineal entre la resistencia ( $R$ ) de un termistor NTC y su temperatura absoluta ( $T$ , en Kelvin):

$$\frac{1}{T} = A + B \cdot \ln(R) + C \cdot (\ln(R))^3$$

Donde:

- $T$ : Temperatura en Kelvin ( $T_{\text{Celsius}} + 273,15$ ).
- $R$ : Resistencia del termistor en ohmios.
- $\ln(R)$ : Logaritmo natural de la resistencia.
- $A, B, C$ : Coeficientes específicos del termistor, obtenidos de la hoja de datos o mediante calibración experimental.

### ¿Por qué usamos esta ecuación?

- **Alta precisión:** Proporciona una precisión de  $\pm 0.1^{\circ}\text{C}$  a  $\pm 0.01^{\circ}\text{C}$  en rangos de temperatura típicos ( $0^{\circ}\text{C}$  a  $100^{\circ}\text{C}$ ), ideal para aplicaciones automotrices donde la exactitud es crucial, como evitar el sobrecalentamiento del motor.
- **Captura la no linealidad:** La resistencia de un termistor NTC cambia exponencialmente con la temperatura. El término  $\ln(R)$  modela la curva básica, y el término  $(\ln(R))^3$  corrige pequeñas desviaciones, asegurando mediciones precisas en un rango amplio.
- **Aplicación práctica:** En un vehículo, la ECU usa esta ecuación (o una tabla de conversión derivada) para traducir la resistencia del termistor en una temperatura que informa decisiones, como activar el ventilador o ajustar la mezcla de combustible.

Para obtener  $A, B, C$ , medirás la resistencia del termistor a tres temperaturas conocidas (por ejemplo,  $25^{\circ}\text{C}$ ,  $50^{\circ}\text{C}$ ,  $75^{\circ}\text{C}$ ) y resolverás un sistema de ecuaciones lineales. Alternativamente, puedes usar coeficientes proporcionados por el fabricante (por ejemplo, para un termistor Vishay NTCLE100E3103JB0:  $A = 3,354016 \times 10^{-3}$ ,  $B = 2,569850 \times 10^{-4}$ ,  $C = 2,620131 \times 10^{-7}$ ).

## 5.5. Caracterización

### Experimento de Caracterización

1. **Materiales:** Termistor NTC  $10\text{k}\Omega$  (por ejemplo, Vishay NTCLE100E3103JB0), resistencia fija de  $10\text{k}\Omega$ , multímetro, termómetro de referencia, fuente de calor (por ejemplo, calentador de agua), protoboard, cables.
2. **Conexión:** Configura el termistor en un divisor de voltaje con una resistencia de  $10\text{k}\Omega$ . Conecta la salida del divisor al pin analógico A0 del Arduino o ESP32.
3. **Medir resistencias:**
  - Sumerge el termistor en agua a temperaturas controladas:  $25^{\circ}\text{C}$  (ambiente),  $50^{\circ}\text{C}$  (calentador bajo),  $75^{\circ}\text{C}$  (calentador medio). Usa un termómetro para verificar la temperatura.
  - Mide la resistencia del termistor en cada punto con el multímetro o calcula la resistencia a partir del voltaje en A0.
4. **Registrar datos:** Completa una tabla con las resistencias medidas.

Cuadro 2: Datos de calibración del termistor NTC 10kΩ.

Temperatura (°C)	Temperatura (K)	Resistencia (Ω)	ln(R)
25	298.15	10000	9.210
50	323.15	4000	8.294
75	348.15	2000	7.601



Figura 5: Configuración experimental del termistor NTC en un divisor de voltaje.

## Calcular Coeficientes Steinhart-Hart

1. Usa los datos de la tabla 2 (tres temperaturas y resistencias).
2. Convierte las temperaturas a Kelvin (por ejemplo, 25°C = 298.15 K).
3. Ingresa los datos en una calculadora Steinhart-Hart online (<https://www.thinksrs.com/downloads/programs/Therm%20Calc/NTCCalculator.htm>) or resuelve el sistema de ecuaciones lineales:

$$\frac{1}{T_i} = A + B \cdot \ln(R_i) + C \cdot (\ln(R_i))^3 \quad \text{para } i = 1, 2, 3$$

4. Obtén los coeficientes. Ejemplo para un termistor de 10kΩ:  $A = 3,354016 \times 10^{-3}$ ,  $B = 2,569850 \times 10^{-4}$ ,  $C = 2,620131 \times 10^{-7}$ .

## 5.6. Implementación

Código (ESP32 con SN65HVD230)

```

1 #include <CAN.h>
2 const int thermPin = A0;
3 const float A = 3.354016e-03, B = 2.569850e-04, C = 2.620131e
4 -07;
5 void setup() {
6     Serial.begin(9600);
7     CAN.setPins(4, 5); // RX = GPIO4, TX = GPIO5
8     if (CAN.begin(500000)) {
9         Serial.println("Nodo\u201dTemperatura\u201diniciado");
10    } else {
11        Serial.println("Error\u201dal\u201diniciar\u201dCAN");
12    }
13 }
14 void loop() {
15     int raw = analogRead(thermPin);
16     float R = 10000.0 * (1023.0 / raw - 1.0); // Divisor de
17     voltaje
18     float lnR = log(R);
19     float T = 1.0 / (A + B * lnR + C * lnR * lnR * lnR);
20     T = T - 273.15; // Kelvin a Celsius
21     Serial.print("T:");
22     Serial.println(T);
23     uint8_t data[2] = {(uint8_t)((int)T >> 8), (uint8_t)((int)
24     )T & 0xFF)};
25     CAN.beginPacket(0x101);
26     CAN.write(data, 2);
27     CAN.endPacket();
28     delay(100);
29 }
```

Código (Arduino con MCP2515)

```

1 #include <SPI.h>
2 #include <mcp_can.h>
3 MCP_CAN CAN(10); // Pin CS
4 const int thermPin = A0;
5 const float A = 3.354016e-03, B = 2.569850e-04, C = 2.620131e
6 -07;
7 void setup() {
8     Serial.begin(9600);
9     while (CAN_OK != CAN.begin(CAN_500KBPS)) {
10         Serial.println("Error\u201dal\u201diniciar\u201dCAN,\u201dreintentando...
11             ");
12         delay(100);
13     }
14     Serial.println("Nodo\u201dTemperatura\u201diniciado");
15 }
```

```

4 void loop() {
5     int raw = analogRead(thermPin);
6     float R = 10000.0 * (1023.0 / raw - 1.0); // Divisor de
7         voltaje
8     float lnR = log(R);
9     float T = 1.0 / (A + B * lnR + C * lnR * lnR * lnR);
10    T = T - 273.15; // Kelvin a Celsius
11    Serial.print("T:");
12    Serial.println(T);
13    unsigned char data[2] = {(unsigned char)((int)T >> 8), (
14        unsigned char)((int)T & 0xFF)};
15    CAN.sendMsgBuf(0x101, 0, 2, data);
16    delay(100);
17 }

```

## 5.7. Actividad

### Actividades

#### 1. Configura el termistor:

- Conecta el termistor NTC de  $10\text{k}\Omega$  en un divisor de voltaje con una resistencia de  $10\text{k}\Omega$ , según el esquemático de la figura 5.
- Conecta la salida del divisor al pin analógico A0 del Arduino o ESP32, según la solución CAN elegida.

#### 2. Caracteriza el termistor:

- Realiza el experimento de caracterización, midiendo la resistencia del termistor a  $25^\circ\text{C}$  (ambiente),  $50^\circ\text{C}$  (calentador bajo), y  $75^\circ\text{C}$  (calentador medio) usando agua y un termómetro de referencia.
- Registra los datos en una tabla como la 2. Usa un multímetro o calcula la resistencia a partir del voltaje en A0.

#### 3. Calcula los coeficientes:

- Usa los datos de la tabla 2 para calcular los coeficientes  $A, B, C$  de la ecuación de Steinhart-Hart, ya sea con una calculadora online (<https://www.thinksrs.com/downloads/programs/Therm%20Calc/NTCCalculator.htm>) o resolviendo el sistema de ecuaciones lineales.
- Alternativamente, usa los coeficientes proporcionados (por ejemplo,  $A = 3,354016 \times 10^{-3}$ ,  $B = 2,569850 \times 10^{-4}$ ,  $C = 2,620131 \times 10^{-7}$ ).

#### 4. Carga el código:

- Carga el código correspondiente (ESP32 o Arduino) en tu microcontrolador.
- Verifica que el código inicialice correctamente el bus CAN y calcule la temperatura usando la ecuación de Steinhart-Hart.

**5. Mide la temperatura:**

- Expon el termistor a diferentes temperaturas (por ejemplo, ambiente, agua tibia, agua caliente).
- Observa los valores de temperatura en el monitor serie (9600 baudios, formato T:valor).
- Asegúrate de que los datos se envíen por CAN con ID 0x101 (esto se verificará en prácticas posteriores).

**6. Reflexiona:**

- ¿Cómo usaría la ECU la temperatura medida (por ejemplo, 90°C) para controlar el ventilador de enfriamiento en un vehículo?
- ¿Por qué es importante la precisión de la ecuación de Steinhart-Hart en un vehículo eléctrico, especialmente para la gestión de baterías?

**7. Opcional: Experimenta con el termistor:**

- Si tienes acceso a un rango más amplio de temperaturas (por ejemplo, hielo a 0°C), mide la resistencia del termistor y verifica si la ecuación de Steinhart-Hart sigue siendo precisa.
- Compara los resultados con una tabla de resistencias proporcionada por el fabricante del termistor.

## 5.8. Referencias

- *NTC Thermistor 10k Datasheet*, Vishay, <https://www.vishay.com/docs/29049/ntcle100.pdf>.
- Hart, J. A., *A Practical Guide to Using NTC Thermistors*, Stanford Research Systems, [https://www.thinksrs.com/downloads/PDFs/ApplicationNotes/NTC\\_AppNote.pdf](https://www.thinksrs.com/downloads/PDFs/ApplicationNotes/NTC_AppNote.pdf).
- *Evaluation of Thermistors Used for Temperature Measurement in Automotive Industry*, SAE International, <https://www.sae.org/publications/technical-papers>.
- *Temperature Sensing in Automotive Applications*, Texas Instruments, <https://www.ti.com/lit/an/snua267/snua267.pdf>.

## 6. Práctica 5: Sensor de Nivel de Combustible

**Objetivo:** Implementar un sensor de nivel de combustible (o un potenciómetro como alternativa) para medir el nivel de combustible y enviar datos por CAN, usando la solución CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230) elegida en la Práctica 2.

## 6.1. Importancia en Vehículos

El sensor de nivel de combustible es un componente crítico en los vehículos modernos, proporcionando datos esenciales sobre la cantidad de combustible disponible, lo que impacta directamente en la seguridad, la eficiencia, la experiencia del conductor y el cumplimiento de normativas. En un vehículo como el Kia Sorento 2019, este sensor envía datos en tiempo real a través del bus CAN (ID 0x103) al cuadro de instrumentos, mostrando el nivel de combustible en una pantalla digital o una aguja analógica, permitiendo al conductor planificar paradas para repostar y evitar quedarse varado.

### **Impacto en la seguridad:**

- **Prevención de emergencias:** Un sensor preciso evita que el conductor subestime el combustible disponible, reduciendo el riesgo de quedarse sin combustible en carreteras aisladas o en condiciones climáticas adversas. Por ejemplo, un nivel inferior al 10 % activa una alerta en el tablero, instando al conductor a repostar.
- **Integración con ADAS:** En sistemas avanzados de asistencia al conductor (ADAS), el nivel de combustible se combina con datos de navegación para sugerir estaciones de servicio cercanas cuando la autonomía es baja, mejorando la seguridad. Por ejemplo, en un Ford Explorer 2020, el sistema SYNC 3 usa estos datos para mostrar rutas optimizadas.

### **Eficiencia de combustible:**

- **Optimización de rutas:** Los datos del sensor permiten calcular la autonomía (por ejemplo, 400 km con un tanque de 70 L), ayudando al conductor a elegir rutas que minimicen el consumo de combustible, especialmente en vehículos diésel o híbridos.
- **Gestión del motor:** La unidad de control electrónico (ECU) usa el nivel de combustible para optimizar parámetros como la inyección de combustible o la regeneración del filtro de partículas diésel (DPF), reduciendo emisiones y cumpliendo con normativas como Euro 6 o EPA Tier 2 Bin 5.

### **Experiencia del conductor:**

- **Visualización clara:** Una lectura precisa del nivel de combustible mejora la confianza del conductor, mostrando datos en tiempo real en el tablero. En vehículos premium, como el Kia Sorento 2019, los datos se integran en pantallas head-up display (HUD) o interfaces digitales, ofreciendo una experiencia moderna.
- **Personalización:** Algunos vehículos permiten configurar alertas personalizadas (por ejemplo, notificación al 25 % de combustible) o integrar los datos con aplicaciones móviles, como Kia Connect.

### **Cumplimiento de normativas:**

- **Seguridad funcional:** El sensor debe cumplir con ISO 26262, garantizando que los sistemas electrónicos no fallen de manera que comprometan la seguridad. Un sensor defecuoso podría mostrar un nivel incorrecto, llevando a decisiones erróneas.
- **Diagnósticos OBD-II:** Los datos del nivel de combustible se registran en la red CAN y son accesibles vía OBD-II, requeridos por regulaciones como la EPA o la UE para monitorear consumo y emisiones.

### Aplicaciones avanzadas:

- **Vehículos híbridos:** En híbridos enchufables (PHEV), como el Toyota Prius Plug-in, el sensor de nivel se combina con datos de la batería para optimizar el uso del motor de combustión versus el eléctrico, maximizando la eficiencia.
- **Telemetría en flotillas:** En camiones o flotillas comerciales, los sensores de nivel se integran con sistemas de telemetría para monitorear el consumo, prevenir robos de combustible y optimizar la logística, como en sistemas de DHL.

En resumen, el sensor de nivel de combustible es clave para garantizar seguridad, eficiencia y una experiencia de conducción óptima, mientras cumple con normativas estrictas.

### 6.2. ¿Qué aprenderás?

- Implementar un sensor resistivo de nivel de combustible o un potenciómetro como alternativa.
- Configurar el sensor para medir el nivel de combustible y enviar datos por CAN.

### 6.3. Explicación

El sensor de nivel de combustible utiliza un flotador conectado a un brazo que mueve un elemento resistivo, variando su resistencia linealmente según el nivel del tanque. En esta práctica, tienes dos opciones:

- **Sensor de nivel de combustible:** Usa un sensor resistivo automotriz (100–900Ω), como los disponibles en DigiKey o Mouser (200–400 MXN). Varía de 100Ω (tanque lleno) a 900Ω (tanque vacío).
- **Potenciómetro:** Usa un potenciómetro de 1kΩ ajustado a 100–900Ω, girándolo manualmente para simular el movimiento del flotador.

Ambas opciones se conectan en un divisor de voltaje, y la resistencia medida se convierte en un porcentaje de nivel de combustible usando una ecuación lineal.

### 6.4. Ecuación

$$\text{Nivel (\%)} = \frac{R_{\max} - R}{R_{\max} - R_{\min}} \times 100$$

Donde:

- $R$ : Resistencia medida del sensor o potenciómetro (en ohmios).
- $R_{\min}$ : Resistencia cuando el tanque está lleno (100Ω).
- $R_{\max}$ : Resistencia cuando el tanque está vacío (900Ω).

Ejemplo: Si  $R = 500\Omega$ , entonces:

$$\text{Nivel (\%)} = \frac{900 - 500}{900 - 100} \times 100 = 50\%$$

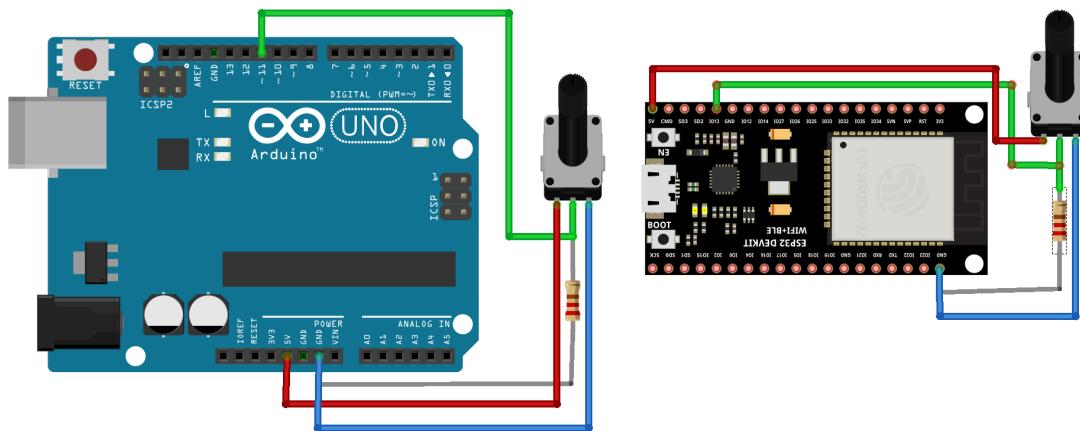


Figura 6: Configuración experimental del sensor de nivel o potenciómetro en un divisor de voltaje.

## 6.5. Implementación

Código (ESP32 con SN65HVD230)

```

1 #include <CAN.h>
2 const int fuelPin = A1;
3 const float R_min = 100, R_max = 900;
4 void setup() {
5     Serial.begin(9600);
6     CAN.setPins(4, 5); // RX = GPIO4, TX = GPIO5
7     if (CAN.begin(500000)) {
8         Serial.println("Nodo Nivel de Combustible iniciado");
9     } else {
10         Serial.println("Error al iniciar CAN");
11     }
12 }
13 void loop() {
14     int raw = analogRead(fuelPin);
15     float R = 10000.0 * (1023.0 / raw - 1.0); // Divisor de
16         voltaje
17     float level = (R_max - R) / (R_max - R_min) * 100;
18     if (level < 0) level = 0;
19     if (level > 100) level = 100;
20     Serial.print("G:");
21     Serial.println(level);
22     uint8_t data[1] = {(uint8_t)level};
23     CAN.beginPacket(0x103);
24     CAN.write(data, 1);
25     CAN.endPacket();
26     delay(100);
27 }
```

## Código (Arduino con MCP2515)

```

1 #include <SPI.h>
2 #include <mcp_can.h>
3 MCP_CAN CAN(10); // Pin CS
4 const int fuelPin = A1;
5 const float R_min = 100, R_max = 900;
6 void setup() {
7     Serial.begin(9600);
8     while (CAN_OK != CAN.begin(CAN_500KBPS)) {
9         Serial.println("Error al iniciar CAN, reintentando...");
10        delay(100);
11    }
12    Serial.println("Nodo Nivel de Combustible iniciado");
13}
14 void loop() {
15     int raw = analogRead(fuelPin);
16     float R = 10000.0 * (1023.0 / raw - 1.0); // Divisor de
17     // voltaje
18     float level = (R_max - R) / (R_max - R_min) * 100;
19     if (level < 0) level = 0;
20     if (level > 100) level = 100;
21     Serial.print("G:");
22     unsigned char data[1] = {(unsigned char)level};
23     CAN.sendMsgBuf(0x103, 0, 1, data);
24     delay(100);
}

```

## 6.6. Actividad

### 1. Configura el sensor o potenciómetro:

- Conecta el sensor de nivel de combustible (100–900Ω) o un potenciómetro ajustado a ese rango en un divisor de voltaje con una resistencia de 10kΩ, según la figura 6.
- Conecta la salida del divisor al pin analógico A1 del Arduino o ESP32.

### 2. Carga el código:

- Carga el código correspondiente (ESP32 o Arduino) en tu microcontrolador.
- Verifica que el código inicialice el bus CAN y calcule el nivel de combustible.

### 3. Mide el nivel de combustible:

- **Sensor:** Mueve el flotador para simular diferentes niveles de combustible (lleno, medio, vacío).
- **Potenciómetro:** Gira el potenciómetro lentamente para simular el cambio de nivel.

- Observa los valores de nivel (en porcentaje) en el monitor serie (9600 baudios, formato G:valor).
- Asegúrate de que los datos se envíen por CAN con ID 0x103 (se verificará en la Práctica 6).

#### 4. Valida los cálculos:

- Usa la ecuación Nivel (%) =  $\frac{R_{\max} - R}{R_{\max} - R_{\min}} \times 100$  para verificar manualmente los niveles. Por ejemplo, si  $R = 500\Omega$ , calcula:  $\frac{900 - 500}{900 - 100} \times 100 = 50\%$ .
- Compara los valores calculados con los mostrados en el monitor serie.

#### 5. Reflexiona:

- ¿Cómo usaría el sistema de navegación de un vehículo los datos del nivel de combustible para optimizar una ruta?
- ¿Por qué es importante que el sensor cumpla con ISO 26262 en términos de seguridad funcional?
- ¿Qué problemas podría causar un sensor de nivel defectuoso en un vehículo híbrido o una flotilla comercial?

#### 6. Opcional: Experimenta:

- Simula un tanque que se vacía gradualmente (por ejemplo, de 100% a 0% en 10 pasos) y registra los valores en el monitor serie.

## 6.7. Referencias

- *Fuel Level Sensor FL-100 Datasheet*, Standard Motor Products, <https://www.standardbrand.com/en/products/sensors/fuel-level-sensors>.
- *How Fuel Level Sensors Work in Automotive Applications*, SAE International, <https://www.sae.org/publications/technical-papers>.
- *Potentiometer as a Sensor*, Bourns, <https://www.bourns.com/docs/technical-documents/technical-library/sensors/automotive-sensors.pdf>.
- *Automotive Sensor Applications*, Texas Instruments, <https://www.ti.com/lit/an/slyt766/slyt766.pdf>.

## 7. Práctica 6: Integración de Sensores en una Red CAN y Visualización en GUI

**Objetivo:** Integrar sensores de RPM (A3144), temperatura (termistor NTC 10kΩ), nivel de combustible (sensor resistivo o potenciómetro) y velocidad (simulada o sensor) en una red CAN, enviando datos al programa GUI Cuadro de Instrumentos, que emula el cuadro de instrumentos de un Kia Sorento 2019, usando la solución CAN Autónomo (Arduino con MCP2515) o CAN Integrado (ESP32 con SN65HVD230).

### 7.1. Importancia en Vehículos

La integración de sensores en una red CAN y su visualización en una interfaz gráfica de usuario (GUI) son esenciales en vehículos modernos, mejorando la seguridad, eficiencia, experiencia del conductor y cumplimiento normativo. La red CAN conecta sensores, ECUs y sistemas de visualización, mientras que la GUI muestra datos en tiempo real, como en el Kia Sorento 2019.

#### Impacto en la seguridad:

- **Monitoreo en tiempo real:** Los datos de temperatura (ID 0x101), RPM (ID 0x102), nivel de combustible (ID 0x103) y velocidad (ID 0x104) se transmiten instantáneamente. Por ejemplo, una temperatura superior a 110°C activa alertas en el tablero, previniendo daños al motor.
- **ADAS:** Los datos CAN se integran con sistemas de asistencia (ADAS), como control de crucero adaptativo, ajustando la velocidad según RPM y nivel de combustible.
- **Seguridad funcional:** Cumple con ISO 26262, asegurando que fallos en la red no comprometan la seguridad.

#### Eficiencia de combustible:

- **Optimización del motor:** Los datos de RPM, temperatura y velocidad permiten a la ECU ajustar la inyección y sincronización, reduciendo emisiones (Euro 6).
- **Gestión en híbridos:** En vehículos híbridos, los datos optimizan el uso del motor eléctrico o de combustión.

#### Experiencia del conductor:

- **Cuadro de instrumentos:** La GUI muestra velocidad (centro), RPM (izquierda), temperatura (derecha, arriba) y combustible (derecha, abajo), con alertas si el combustible cae por debajo del 10% o la temperatura excede 110°C.
- **Diagnósticos:** Los datos CAN son accesibles vía OBD-II, facilitando reparaciones.

#### Cumplimiento de normativas:

- **ISO 11898-2:** Garantiza interoperabilidad en la red CAN.
- **OBD-II:** Monitorea emisiones y consumo, requerido por EPA y UE.

## 7.2. ¿Qué aprenderás?

- Integrar sensores de RPM, temperatura, nivel de combustible y velocidad en una red CAN.
- Configurar un nodo transmisor para enviar datos con IDs 0x101–0x104.
- Configurar un nodo receptor para enviar datos a la GUI vía puerto serie (115200 baudios, formato 101:valor, 102:valor, 103:valor, 104:valor).
- Verificar la visualización en la GUI Cuadro de Instrumentos.

## 7.3. Explicación

Conectarás los sensores de las prácticas anteriores (A3144, termistor NTC, sensor de nivel o potenciómetro) y un sensor de velocidad simulado (potenciómetro) a un nodo transmisor que envía datos por CAN (IDs 0x101–0x104). Un nodo receptor envía los datos al programa GUI Cuadro de Instrumentos vía puerto serie, emulando un cuadro de instrumentos.

## 7.4. Conexión

### Conexión de la Red CAN y Sensores

- **Materiales:** Sensor A3144, termistor NTC 10kΩ, sensor de nivel (100–900Ω) o potenciómetro, potenciómetro (velocidad), resistencias de 10kΩ, Arduino Uno o ESP32, MCP2515 o SN65HVD230, cables, protoboard, cable USB.
- **Nodo Transmisor:**
  - Conecta el A3144 a pin 2 (Arduino) o GPIO 5 (ESP32).
  - Conecta el termistor a A0 en un divisor de voltaje.
  - Conecta el sensor de nivel o potenciómetro a A1 en un divisor de voltaje.
  - Conecta un potenciómetro (velocidad) a A2 en un divisor de voltaje.
  - Conecta el MCP2515 (Arduino) o SN65HVD230 (ESP32) al bus CAN.
- **Nodo Receptor:**
  - Conecta otro Arduino o ESP32 al bus CAN.
  - Conecta el receptor al computador vía USB (puerto COM, 115200 baudios).
- **Bus CAN:** Une CAN H y CAN L, con resistencias de 120Ω en los extremos.
- **GUI:** Ejecuta GUI Cuadro de Instrumentos y selecciona el puerto COM.

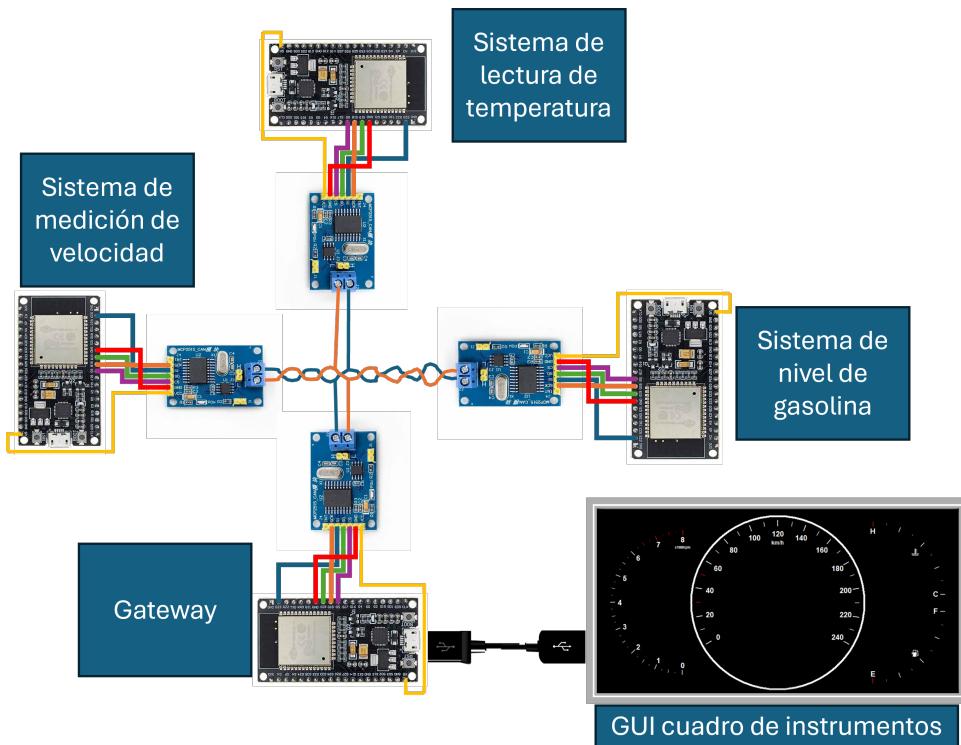


Figura 7: Esquemático de la red CAN con sensores y nodo receptor conectado a la GUI.

## 7.5. Implementación

Código del Nodo Transmisor (ESP32 con SN65HVD230)

```

1 #include <CAN.h>
2 const int hallPin = 5;
3 const int thermPin = A0;
4 const int fuelPin = A1;
5 const int speedPin = A2;
6 const float A = 3.354016e-03, B = 2.569850e-04, C = 2.620131e
-07;
7 const float R_min = 100, R_max = 900;
8 volatile int pulses = 0;
9 void IRAM_ATTR countPulse() {
10     pulses++;
11 }
12 void setup() {
13     pinMode(hallPin, INPUT);
14     attachInterrupt(digitalPinToInterrupt(hallPin),
15                     countPulse, RISING);
16     Serial.begin(115200);
17     CAN.setPins(4, 5); // RX = GPIO4, TX = GPIO5
18     if (CAN.begin(500000)) {
19         Serial.println("Nodo Transmisor iniciado");
20     } else {
21         Serial.println("Error al iniciar CAN");
22     }
23 }
```

```

22 }
23 void loop() {
24     // RPM (ID 0x102)
25     pulses = 0;
26     delay(1000);
27     int rpm = pulses * 60;
28     uint8_t rpmData[2] = {(uint8_t)(rpm >> 8), (uint8_t)(rpm
29         & 0xFF)};
30     CAN.beginPacket(0x102);
31     CAN.write(rpmData, 2);
32     CAN.endPacket();
33     // Temperatura (ID 0x101)
34     int thermRaw = analogRead(thermPin);
35     float R = 10000.0 * (1023.0 / thermRaw - 1.0);
36     float lnR = log(R);
37     float T = 1.0 / (A + B * lnR + C * lnR * lnR * lnR);
38     T = T - 273.15;
39     uint8_t tempData[2] = {(uint8_t)((int)T >> 8), (uint8_t)
40         ((int)T & 0xFF)};
41     CAN.beginPacket(0x101);
42     CAN.write(tempData, 2);
43     CAN.endPacket();
44     // Nivel de combustible (ID 0x103)
45     int fuelRaw = analogRead(fuelPin);
46     float R_fuel = 10000.0 * (1023.0 / fuelRaw - 1.0);
47     float level = (R_max - R_fuel) / (R_max - R_min) * 100;
48     if (level < 0) level = 0;
49     if (level > 100) level = 100;
50     uint8_t fuelData[1] = {(uint8_t)level};
51     CAN.beginPacket(0x103);
52     CAN.write(fuelData, 1);
53     CAN.endPacket();
54     // Velocidad (ID 0x104)
55     int speedRaw = analogRead(speedPin);
56     float speed = (speedRaw / 1023.0) * 120; // Escala de
57         0-120 km/h
58     uint8_t speedData[2] = {(uint8_t)((int)speed >> 8), (
59         uint8_t)((int)speed & 0xFF)};
60     CAN.beginPacket(0x104);
61     CAN.write(speedData, 2);
62     CAN.endPacket();
63     delay(100);
64 }
```

### Código del Nodo Transmisor (Arduino con MCP2515)

```

1 #include <SPI.h>
2 #include <mcp_can.h>
3 MCP_CAN CAN(10); // Pin CS
```

```

4 const int hallPin = 2;
5 const int thermPin = A0;
6 const int fuelPin = A1;
7 const int speedPin = A2;
8 const float A = 3.354016e-03, B = 2.569850e-04, C = 2.620131e
-07;
9 const float R_min = 100, R_max = 900;
10 volatile int pulses = 0;
11 void countPulse() {
12     pulses++;
13 }
14 void setup() {
15     pinMode(hallPin, INPUT);
16     attachInterrupt(digitalPinToInterrupt(hallPin),
17                     countPulse, RISING);
18     Serial.begin(115200);
19     while (CAN_OK != CAN.begin(CAN_500KBPS)) {
20         Serial.println("Error al iniciar CAN, reintentando...");
21         delay(100);
22     }
23     Serial.println("Nodo Transmisor iniciado");
24 }
25 void loop() {
26     // RPM (ID 0x102)
27     pulses = 0;
28     delay(1000);
29     int rpm = pulses * 60;
30     unsigned char rpmData[2] = {(unsigned char)(rpm >> 8),
31                                 (unsigned char)(rpm & 0xFF)};
32     CAN.sendMsgBuf(0x102, 0, 2, rpmData);
33     // Temperatura (ID 0x101)
34     int thermRaw = analogRead(thermPin);
35     float R = 10000.0 * (1023.0 / thermRaw - 1.0);
36     float lnR = log(R);
37     float T = 1.0 / (A + B * lnR + C * lnR * lnR * lnR);
38     T = T - 273.15;
39     unsigned char tempData[2] = {(unsigned char)((int)T >> 8),
40                                 (unsigned char)((int)T & 0xFF)};
41     CAN.sendMsgBuf(0x101, 0, 2, tempData);
42     // Nivel de combustible (ID 0x103)
43     int fuelRaw = analogRead(fuelPin);
44     float R_fuel = 10000.0 * (1023.0 / fuelRaw - 1.0);
45     float level = (R_max - R_fuel) / (R_max - R_min) * 100;
46     if (level < 0) level = 0;
47     if (level > 100) level = 100;
48     unsigned char fuelData[1] = {(unsigned char)level};
49     CAN.sendMsgBuf(0x103, 0, 1, fuelData);
50     // Velocidad (ID 0x104)
51     int speedRaw = analogRead(speedPin);

```

```

49     float speed = (speedRaw / 1023.0) * 120; // Escala de
50     0-120 km/h
51     unsigned char speedData[2] = {(unsigned char)((int)speed
52         >> 8), (unsigned char)((int)speed & 0xFF)};
53     CAN.sendMsgBuf(0x104, 0, 2, speedData);
54     delay(100);
55 }
```

## Código del Nodo Receptor (ESP32 con SN65HVD230)

```

1 #include <CAN.h>
2 void setup() {
3     Serial.begin(115200);
4     CAN.setPins(4, 5); // RX = GPIO4, TX = GPIO5
5     if (CAN.begin(500000)) {
6         Serial.println("Nodo_Receptor_iniciado");
7     } else {
8         Serial.println("Error al iniciar CAN");
9     }
10 }
11 void loop() {
12     if (CAN.parsePacket()) {
13         long unsigned int rxId = CAN.packetId();
14         int len = CAN.available();
15         uint8_t rxBuf[8];
16         for (int i = 0; i < len; i++) {
17             rxBuf[i] = CAN.read();
18         }
19         if (rxId == 0x102) { // RPM
20             int rpm = (rxBuf[0] << 8) | rxBuf[1];
21             Serial.print("102:"); Serial.println(rpm);
22         } else if (rxId == 0x101) { // Temperatura
23             int temp = (rxBuf[0] << 8) | rxBuf[1];
24             Serial.print("101:"); Serial.println(temp);
25         } else if (rxId == 0x103) { // Nivel de combustible
26             int level = rxBuf[0];
27             Serial.print("103:"); Serial.println(level);
28         } else if (rxId == 0x104) { // Velocidad
29             int speed = (rxBuf[0] << 8) | rxBuf[1];
30             Serial.print("104:"); Serial.println(speed);
31         }
32         Serial.println();
33     }
34 }
```

## Código del Nodo Receptor (Arduino con MCP2515)

```

1 #include <SPI.h>
2 #include <mcp_can.h>
3 MCP_CAN CAN(10); // Pin CS
4 void setup() {
5     Serial.begin(115200);
6     while (CAN_OK != CAN.begin(CAN_500KBPS)) {
7         Serial.println("Error al iniciar CAN, reintentando...");
8         delay(100);
9     }
10    Serial.println("Nodo Receptor iniciado");
11}
12void loop() {
13    unsigned long rxId;
14    unsigned char len = 0;
15    unsigned char rxBuf[8];
16    if (CAN_MSGAVAIL == CAN.checkReceive()) {
17        CAN.readMsgBuf(&len, rxBuf);
18        rxId = CAN.getCanId();
19        if (rxId == 0x102) { // RPM
20            int rpm = (rxBuf[0] << 8) | rxBuf[1];
21            Serial.print("102:"); Serial.println(rpm);
22        } else if (rxId == 0x101) { // Temperatura
23            int temp = (rxBuf[0] << 8) | rxBuf[1];
24            Serial.print("101:"); Serial.println(temp);
25        } else if (rxId == 0x103) { // Nivel de combustible
26            int level = rxBuf[0];
27            Serial.print("103:"); Serial.println(level);
28        } else if (rxId == 0x104) { // Velocidad
29            int speed = (rxBuf[0] << 8) | rxBuf[1];
30            Serial.print("104:"); Serial.println(speed);
31        }
32        Serial.println();
33    }
34}

```

## 7.6. Actividad

## Actividades

## 1. Configura los sensores:

- Conecta el sensor A3144, termistor NTC, sensor de nivel (o potenciómetro) y potenciómetro (velocidad) al nodo transmisor, según las figuras 4, 5, 6, y un esquema similar para el potenciómetro de velocidad.
- Verifica los pines: A3144 a pin 2/GPIO 5, termistor a A0, sensor de nivel a A1, velocidad a A2.

**2. Configura la red CAN:**

- Conecta el nodo transmisor y receptor al bus CAN, según la figura 7.
- Usa resistencias de  $120\Omega$  en los extremos del bus.

**3. Carga los códigos:**

- Carga el código del transmisor en un Arduino o ESP32.
- Carga el código del receptor en otro Arduino o ESP32.
- Verifica la inicialización del bus CAN en el monitor serie (115200 baudios).

**4. Ejecuta la GUI:**

- Instala GUI Cuadro de Instrumentos según el Manual de Usuario (carpeta por defecto: C:\Program Files\GUI Cuadro de Instrumentos).
- Conecta el nodo receptor al computador vía USB.
- Ejecuta la GUI, selecciona el puerto COM (por ejemplo, COM3) y haz clic en “Conectar”.
- Verifica que la GUI muestre velocidad (centro), RPM (izquierda), temperatura (derecha, arriba) y nivel de combustible (derecha, abajo).

**5. Verifica los datos:**

- Simula condiciones:
  - Gira el eje con el imán (A3144) para RPM (600, 1200, 1800).
  - Expon el termistor a temperaturas (25°C, 50°C, 75°C).
  - Ajusta el sensor de nivel o potenciómetro (100 %, 50 %, 25 %).
  - Ajusta el potenciómetro de velocidad (0, 60, 120 km/h).
- Confirma que la GUI muestra alertas si el combustible es <math>\leq 10\%</math> o la temperatura >110°C.
- Usa el monitor serie para depurar (formato: 101:valor, 102:valor, 103:valor, 104:valor).

**6. Solución de problemas:**

- Si la GUI no detecta el puerto COM, verifica la conexión USB y controladores (Arduino: <https://www.arduino.cc/en/software>; ESP32: documentación del fabricante).
- Si los indicadores no se actualizan, confirma el baudrate (115200) y el formato de datos.
- Si las imágenes no cargan, verifica la carpeta Imágenes en la ruta de instalación.

**7. Reflexiona:**

- ¿Cómo mejora la red CAN la seguridad y eficiencia en un vehículo?

- ¿Por qué es crítica la visualización en tiempo real para conductores y técnicos?
- ¿Cómo se integran los datos de velocidad en ADAS?

#### 8. Opcional: Experimenta:

- Añade un segundo nodo transmisor con otro sensor (ID 0x105) y verifica en la GUI.
- Simula un fallo desconectando una resistencia de 120Ω y observa el impacto.

### 7.7. Referencias

- ISO 11898-2:2003, *Controller Area Network (CAN) - Part 2: High-speed medium access unit*, International Organization for Standardization, 2003.
- ISO 26262:2018, *Road vehicles – Functional safety*, International Organization for Standardization, 2018.
- *CAN Bus in Automotive Applications*, Texas Instruments, <https://www.ti.com/lit/an/sloa101b/sloa101b.pdf>.
- *Automotive Instrument Cluster Design*, NXP Semiconductors, <https://www.nxp.com/docs/en/application-note/AN12345.pdf>.
- *Manual de Usuario: GUI Cuadro de Instrumentos*, Ing. Sergio Josué Ortiz Hernández, 2025.

## 8. Práctica 7: Análisis de Tramas CAN con Analizador Lógico Saleae Logic 2

**Objetivo:** Utilizar un analizador lógico Saleae Logic 2 para capturar y analizar tramas CAN (IDs 0x101–0x104) generadas por la red CAN de la Práctica 6, identificando campos como ID, DLC y datos, para comprender la comunicación en un bus CAN automotriz.

### 8.1. Importancia en Vehículos

El análisis de tramas CAN con un analizador lógico es una técnica clave en el diagnóstico y desarrollo de sistemas automotrices. Los analizadores lógicos, como el Saleae Logic 2, permiten visualizar las señales CAN H y CAN L, decodificar tramas y detectar errores, lo que es esencial para garantizar la fiabilidad de la red CAN en vehículos como el Kia Sorento 2019.

#### Impacto en la seguridad:

- **Detección de errores:** Identifica fallos como tramas corruptas o colisiones, asegurando que datos críticos (temperatura, RPM, combustible, velocidad) lleguen correctamente a la ECU, cumpliendo con ISO 26262.

- **Validación de sistemas:** Verifica que los sensores envíen datos correctos, previniendo fallos que podrían desactivar alertas (por ejemplo, temperatura  $\geq 110^{\circ}\text{C}$ ).

#### Eficiencia en desarrollo:

- **Diagnósticos rápidos:** Los técnicos usan analizadores para depurar redes CAN, reduciendo el tiempo de reparación en talleres.
- **Optimización:** Analizar el tráfico CAN ayuda a optimizar la prioridad de mensajes, mejorando la eficiencia del bus.

#### Cumplimiento de normativas:

- **ISO 11898-2:** El análisis asegura que las tramas cumplen con el estándar CAN de alta velocidad (500 kbps).
- **OBD-II:** Facilita la verificación de datos accesibles vía OBD-II para pruebas de emisiones.

#### Aplicaciones avanzadas:

- **Vehículos autónomos:** El análisis de tramas valida la comunicación entre sensores, ECUs y actuadores en sistemas ADAS.
- **Telemetría:** En flotillas, el análisis de datos CAN permite monitorear el rendimiento en tiempo real.

### 8.2. ¿Qué aprenderás?

- Configurar un analizador lógico Saleae Logic 2 para capturar señales CAN.
- Conectar el analizador al bus CAN (CAN H y CAN L).
- Decodificar tramas CAN y analizar campos (ID, DLC, datos).
- Identificar errores en la comunicación CAN.

### 8.3. Explicación

Usarás el analizador lógico Saleae Logic 2 para capturar las señales diferenciales CAN H y CAN L del bus CAN configurado en la Práctica 6. El software decodificará las tramas, mostrando los campos: Identificador (ID), Longitud de Datos (DLC), Datos, CRC, etc. Esto simula un diagnóstico automotriz real, donde los técnicos verifican la integridad de la red CAN.

## 8.4. Conexión

### Conexión del Analizador Lógico

- **Materiales:** Analizador lógico Saleae (Logic 8 o Pro 16), red CAN de la Práctica 6 (Arduino/ESP32, MCP2515/SN65HVD230, sensores), cables, computador con Saleae Logic 2.
- **Conexión al bus CAN:**
  - Conecta el canal 0 del analizador a CAN H.
  - Conecta el canal 1 a CAN L.
  - Conecta el cable de tierra del analizador a GND del bus CAN.
- **Software:** Instala Saleae Logic 2 desde <https://www.saleae.com/downloads/>.
- **Nodo Transmisor:** Usa el mismo nodo transmisor de la Práctica 6 para generar datos CAN (IDs 0x101–0x104).

## 8.5. Implementación

### Código del Nodo Transmisor (ESP32 con SN65HVD230)

```

1 #include <CAN.h>
2 const int hallPin = 5;
3 const int thermPin = A0;
4 const int fuelPin = A1;
5 const int speedPin = A2;
6 const float A = 3.354016e-03, B = 2.569850e-04, C = 2.620131e
-07;
7 const float R_min = 100, R_max = 900;
8 volatile int pulses = 0;
9 void IRAM_ATTR countPulse() {
10     pulses++;
11 }
12 void setup() {
13     pinMode(hallPin, INPUT);
14     attachInterrupt(digitalPinToInterrupt(hallPin),
15                     countPulse, RISING);
16     Serial.begin(115200);
17     CAN.setPins(4, 5);
18     if (CAN.begin(500000)) {
19         Serial.println("Nodo\u20d7Transmisor\u20d7iniciado");
20     } else {
21         Serial.println("Error\u20d7al\u20d7iniciar\u20d7CAN");
22     }
23 }
24 void loop() {
// RPM (ID 0x102)

```

```

25     pulses = 0;
26     delay(1000);
27     int rpm = pulses * 60;
28     uint8_t rpmData[2] = {(uint8_t)(rpm >> 8), (uint8_t)(rpm
29         & 0xFF)};
30     CAN.beginPacket(0x102);
31     CAN.write(rpmData, 2);
32     CAN.endPacket();
33     // Temperatura (ID 0x101)
34     int thermRaw = analogRead(thermPin);
35     float R = 10000.0 * (1023.0 / thermRaw - 1.0);
36     float lnR = log(R);
37     float T = 1.0 / (A + B * lnR + C * lnR * lnR * lnR);
38     T = T - 273.15;
39     uint8_t tempData[2] = {(uint8_t)((int)T >> 8), (uint8_t)
40         ((int)T & 0xFF)};
41     CAN.beginPacket(0x101);
42     CAN.write(tempData, 2);
43     CAN.endPacket();
44     // Nivel de combustible (ID 0x103)
45     int fuelRaw = analogRead(fuelPin);
46     float R_fuel = 10000.0 * (1023.0 / fuelRaw - 1.0);
47     float level = (R_max - R_fuel) / (R_max - R_min) * 100;
48     if (level < 0) level = 0;
49     if (level > 100) level = 100;
50     uint8_t fuelData[1] = {(uint8_t)level};
51     CAN.beginPacket(0x103);
52     CAN.write(fuelData, 1);
53     CAN.endPacket();
54     // Velocidad (ID 0x104)
55     int speedRaw = analogRead(speedPin);
56     float speed = (speedRaw / 1023.0) * 120;
57     uint8_t speedData[2] = {(uint8_t)((int)speed >> 8), (
58         uint8_t)((int)speed & 0xFF)};
59     CAN.beginPacket(0x104);
60     CAN.write(speedData, 2);
61     CAN.endPacket();
62     delay(100);
63 }
```

## 8.6. Actividad

### Actividades

#### 1. Configura el hardware:

- Mantén la red CAN de la Práctica 6 activa (nodo transmisor enviando datos).
- Conecta el analizador lógico Saleae a CAN H (canal 0), CAN L (canal 1) y GND, según la figura ??.

**2. Configura Saleae Logic 2:**

- Abre Saleae Logic 2 y conecta el analizador vía USB.
- Configura el protocolo CAN: selecciona “CAN” en “Analyzers”, asigna CAN H a canal 0, CAN L a canal 1, y establece 500 kbps.
- Ajusta la frecuencia de muestreo a 10 MHz (mínimo para CAN a 500 kbps).

**3. Captura datos:**

- Inicia la captura en Logic 2.
- Simula datos en el nodo transmisor:
  - Gira el imán (A3144) para RPM (600, 1200, 1800).
  - Cambia la temperatura del termistor (25°C, 50°C, 75°C).
  - Ajusta el potenciómetro de combustible (100%, 50%, 25%).
  - Ajusta el potenciómetro de velocidad (0, 60, 120 km/h).
- Detén la captura tras 10 segundos.

**4. Analiza las tramas:**

- En Logic 2, visualiza las tramas decodificadas (ID, DLC, datos).
- Identifica las tramas con IDs 0x101 (temperatura), 0x102 (RPM), 0x103 (combustible), 0x104 (velocidad).
- Verifica los datos, por ejemplo:
  - ID 0x103, DLC=1, datos=[0x32] (50 % combustible).
  - ID 0x104, DLC=2, datos=[0x00, 0x3C] (60 km/h).

**5. Detecta errores:**

- Busca errores en Logic 2 (por ejemplo, CRC error o “Bit stuffing error”).
- Simula un fallo desconectando una resistencia de 120Ω y observa el impacto en las tramas.

**6. Reflexiona:**

- ¿Cómo ayuda un analizador lógico a diagnosticar problemas en una red CAN?
- ¿Por qué es importante verificar los campos ID y DLC en aplicaciones automotrices?
- ¿Cómo se usa el análisis de tramas en vehículos autónomos o telemetría?

**7. Opcional: Experimenta:**

- Aumenta la frecuencia de envío (reduce el delay a 50 ms) y observa el tráfico CAN.
- Añade un segundo nodo transmisor con ID 0x105 y analiza las nuevas tramas.

## 8.7. Referencias

- *Logic 2 User Guide*, Saleae, <https://support.saleae.com/logic-2>.
- ISO 11898-2:2003, *Controller Area Network (CAN) - Part 2: High-speed medium access unit*, International Organization for Standardization, 2003.
- *CAN Bus Debugging with Logic Analyzers*, Texas Instruments, <https://www.ti.com/lit/an/sloa102/sloa102.pdf>.
- *Automotive CAN Bus Analysis*, SAE International, <https://www.sae.org/publications/technical-papers>.

## 9. Conclusión

### Reflexión Final

A lo largo de las siete prácticas de este manual, los estudiantes han recorrido un camino integral para comprender y aplicar la red CAN (Controller Area Network) en un contexto automotriz, específicamente emulando el cuadro de instrumentos de un Kia Sorento 2019. Desde los fundamentos de la comunicación CAN en las prácticas iniciales hasta la integración avanzada de sensores y la visualización de datos en tiempo real en la Práctica 6, y culminando con el análisis de tramas CAN mediante el analizador lógico Saleae Logic 2 en la Práctica 7, este manual ha proporcionado una base sólida para el desarrollo de competencias técnicas esenciales en ingeniería automotriz.

En las primeras prácticas, se exploraron los principios de la red CAN, incluyendo su arquitectura, protocolos y estándares (ISO 11898-2), así como la configuración de hardware como el MCP2515 (Arduino) y SN65HVD230 (ESP32). Los estudiantes aprendieron a conectar sensores como el A3144 (RPM), termistor NTC 10kΩ (temperatura) y sensores resistivos (nivel de combustible), enviando datos con identificadores específicos (0x101–0x103). La Práctica 6 avanzó al integrar un sensor de velocidad (ID 0x104) y conectar la red CAN a la interfaz gráfica *GUI Cuadro de Instrumentos*, mostrando datos en tiempo real (velocidad, RPM, temperatura, combustible) con alertas para condiciones críticas (combustible  $\leq 10\%$ , temperatura  $\geq 110^{\circ}\text{C}$ ). Este proceso resaltó la importancia de la comunicación serie (115200 baudios, formato 101:valor, 102:valor, etc.) y la interoperabilidad con sistemas de visualización, fundamentales para la experiencia del conductor y el cumplimiento de normativas como OBD-II.

La Práctica 7 introdujo una herramienta clave en el diagnóstico automotriz: el analizador lógico Saleae Logic 2. Los estudiantes aprendieron a capturar y decodificar tramas CAN, identificando campos como ID, DLC y datos, y detectando errores como CRC o bit stuffing. Esta práctica simuló escenarios reales de depuración en talleres, esenciales para garantizar la seguridad funcional (ISO 26262) y optimizar sistemas avanzados como ADAS (Advanced Driver Assistance Systems).

### Impacto y aprendizajes:

- **Seguridad:** La integración de sensores y el análisis de tramas aseguran que datos críticos se transmitan sin errores, previniendo fallos que podrían comprometer la seguridad del vehículo.
- **Eficiencia:** La red CAN optimiza el rendimiento del motor y el consumo de com-

bustible, apoyando normativas ambientales (Euro 6).

- **Diagnósticos:** El uso de herramientas como la GUI y el analizador lógico permite a técnicos diagnosticar y resolver problemas rápidamente.
- **Aplicaciones modernas:** Las habilidades adquiridas son aplicables en vehículos autónomos, telemetría de flotillas y desarrollo de sistemas embebidos.

**Reflexión para los estudiantes:** Este manual no solo ha desarrollado habilidades técnicas, como programación de microcontroladores, manejo de buses CAN y análisis de señales, sino que también ha fomentado un entendimiento profundo de cómo las tecnologías de comunicación impactan la seguridad, eficiencia y experiencia en vehículos modernos. Los conocimientos adquiridos son un punto de partida para explorar áreas avanzadas, como CAN FD, integración con IoT o desarrollo de sistemas ADAS. Se recomienda experimentar con sensores adicionales, simular fallos complejos o conectar la red CAN a un vehículo real (bajo supervisión) para profundizar en estas aplicaciones. En conclusión, este manual prepara a los estudiantes para enfrentar desafíos en la industria automotriz, desde el diseño de sistemas embebidos hasta el diagnóstico en tiempo real. La combinación de teoría, práctica y herramientas profesionales, como el Saleae Logic 2, asegura una formación robusta, alineada con las demandas de la ingeniería automotriz moderna.

## 10. Referencias

- ISO 11898-2:2003, *Controller Area Network (CAN) - Part 2: High-speed medium access unit*, International Organization for Standardization, 2003.
- Robert Bosch GmbH, *CAN Specification Version 2.0*, 1991, <https://www.nxp.com/docs/en/reference-manual/CAN2spec.pdf>.
- Lawrenz, W., *CAN System Engineering: From Theory to Practical Applications*, Springer, 2013, ISBN: 978-1447154181.
- Texas Instruments, *SN65HVD230 CAN Transceiver Datasheet*, 2020, <https://www.ti.com/lit/ds/symlink/sn65hvd230.pdf>.
- Microchip Technology, *MCP2515 Stand-Alone CAN Controller Datasheet*, 2010, <https://ww1.microchip.com/downloads/en/DeviceDoc/20001801J.pdf>.
- *ESP32-CAN Library*, GitHub, <https://github.com/miwagner/ESP32-Arduino-CAN>.
- *MCP\_CAN Library*, GitHub, [https://github.com/coryjfowler/MCP\\_CAN\\_lib](https://github.com/coryjfowler/MCP_CAN_lib).
- *Hall Effect Sensor A3144*, SparkFun, <https://www.sparkfun.com/datasheets/Components/A3144.pdf>.
- Storey, N., *Electronics: A Systems Approach*, Pearson, 2017, ISBN: 978-1292114064.
- *Using Hall Effect Sensors for RPM Measurement*, SparkFun, <https://learn.sparkfun.com/tutorials/hall-effect-sensor-hookup-guide>.

- *NTC Thermistor 10k Datasheet*, Vishay, <https://www.vishay.com/docs/29049/ntcle100.pdf>.
- Hart, J. A., *A Practical Guide to Using NTC Thermistors*, Stanford Research Systems, [https://www.thinksrs.com/downloads/PDFs/ApplicationNotes/NTC\\_AppNote.pdf](https://www.thinksrs.com/downloads/PDFs/ApplicationNotes/NTC_AppNote.pdf).
- Sedra, A. S., & Smith, K. C., *Microelectronic Circuits*, Oxford University Press, 2014, ISBN: 978-0199339136.
- *Automotive Fuel Level Sensors*, Texas Instruments, <https://www.ti.com/lit/an/snua220/snua220.pdf>.
- Ribbens, W. B., *Understanding Automotive Electronics*, Butterworth-Heinemann, 2017, ISBN: 978-0128108239.
- *How Fuel Level Sensors Work*, Analog Devices, <https://www.analog.com/en/technical-articles/fuel-level-sensing.html>.
- *Controller Area Network (CAN) Tutorial*, National Instruments, [https://www.ni.com/docs/en-US/bundle/labview/page/lvhowto/can\\_tutorial.html](https://www.ni.com/docs/en-US/bundle/labview/page/lvhowto/can_tutorial.html).
- *Saleae Logic User Guide*, Saleae, <https://support.saleae.com/user-guide>.
- *Analyzing CAN Bus with Logic Analyzers*, Saleae, <https://support.saleae.com/protocols/can>.
- Erjavec, J., *Automotive Technology: A Systems Approach*, Cengage Learning, 2014, ISBN: 978-1133612315.
- Denton, T., *Advanced Automotive Fault Diagnosis*, Routledge, 2016, ISBN: 978-1138429109.