BENEMÉRITA UNIVERSIDAD AUTONOMA DE PUEBLA

Facultad en Ciencias de la Electrónica

Sistema para monitorear la PCM de un vehículo mediante una aplicación móvil y su potencial aplicación en IoV

Tesis para obtener el grado de:
Licenciado en Ingeniería en Sistemas Automotrices

Presenta:
González Huerta Ramón Iván

Director de Tesis:
Dr. Israel Vivaldo de la Cruz

Asesor de Tesis:
Dr. Roberto Carlos Ambrosio Lázaro

Septiembre 2024 Puebla, Pue

Dedicatoria

Dedico esta tesis a mis dos grandes pilares, mis padres quienes siempre estuvieron junto a mi apoyándome, motivándome a seguir, me tuvieron paciencia y me brindaron todo el amor que un padre puede dar, permitiéndome este gran logro.
A mí hermano que siempre está conmigo, alegrándome con sus ocurrencias y creyendo en mí. Serás un gran Actuario.
Y a todos mis amigos que estuvieron conmigo en toda mi formación y apoyando en este proceso, estando conmigo en las buenas y en las malas. Brito, Faa, Erick, Jackson, Pepilla y Leydi. Por siempre DreamTeam.
Dedico esta tesis a todos mis compañeros y ahora colegas.

Agradecimientos

Agradezco a mis padres, Abigail Huerta y José Ramón González. A y mi hermano Carlos Emilio, quienes siempre estuvieron conmigo, me ayudaron, apoyaron a seguir y mejorar y me otorgaron todo lo necesario y más obtener este gran logro en mi vida

A mis mejores amigos quienes me dieron momentos inolvidables durante esta etapa. Y a todos aquellos que formaron párate de esta etapa en mi vida.

Agradezco principalmente a la Benemérita Universidad Autónoma de Puebla, la Facultad de Ciencias de la Electrónica, docentes y sinodales, que me permitieron terminar la carrera, compartieron su conocimiento y me brindaron una excelente preparación académica y profesional, en especial a mi asesor el Dr. Roberto Carlos Ambrosio Lázaro, por apoyar mi proyecto, aconsejarme y apoyándome en este proceso.

Índice general

Índice de figuras

Índice de tablas

Resumen

Actualmente los avances tecnológicos nos permiten estar conectados a internet desde diferentes equipos o dispositivos (IoT), entre ellos un automóvil o específicamente los diferentes módulos o computadoras que se encuentran dentro del mismo, esto se conoce como Internet de los vehículos (IoV) por sus siglas en ingles. Esto nos permiten tener comunicaciones mucho más rápidas, accesibles y en tiempo real. Sin embargo esos avances se pueden observar únicamente en vehículos de gama alta, los cuales representan un porcentaje muy bajo de los automóviles en México. Así que, se diseñó un sistema usando un microcontrolador *Arduino Uno* y *ESP32* en conjunto con un *transceptor CAN*, con el que se pueda recibir información del *OBD-II* a través de la red *CAN-bus* y posteriormente enviar y almacenar los datos obtenidos en la nube. El sistema se probó y finalizó con la tarjeta *ESP32 devKit V1*, esta cuenta con un controlador CAN y un módulo *wifi* lo que facilita la conexión a internet, lo que sienta bases para posibles trabajos en IoT y el envió de datos a la nube. Los datos se registrarán en una base de datos en tiempo real y con posibilidad de mostrarla en una aplicación para *teléfonos inteligentes* esto con el fin de tener un monitoreo constante que permita realizar un mantenimiento preventivo ampliando así la vida útil de automóvil de manera accesible.

Para asegurar que exista una buena comunicación CAN, tanto recibir como transmitir mensajes, se realizaron varias pruebas con las diferentes librerías y dispositivos (*microcontrolador* y *transceptor)* y una prueba final en uno de los vehículos que se encuentran en el *laboratorio de sistemas automotrices*, ubicado en el edificio *EMA7* de Ciudad Universitaria.

Capítulo 1

Introducción

El internet de las cosas (IoT), por su abreviatura en inglés, es un concepto que se refiere a una interconexión digital de objetos cotidianos con internet, el cual ha tomado mucha relevancia en los últimos años con el desarrollo de las Tecnologías de la Información y desarrollo [20] [41]. Esta creciente tecnología se ha expandido cada vez más a diferentes sectores, entre ellos la industria automotriz, lo que se conoce como internet de los vehículos (*IoV*) por sus siglas en ingles. IoV permite una interconexión entre vehículos con otros vehículos, cosas, entorno y usuarios a través de internet [25].

Los vehículos actuales, en especial los de gama alta cuenta con una gran cantidad de sensores y actuadores, los cuales por medio de las (ECU) Unidades de Control Electrónico, se encargan de asegurar el correcto funcionamiento del vehículo. Existen diferentes ECU y cada una se encarga de una función específica del automóvil [41] [40].

Las ECU reciben datos de los nodos repartidos en el vehículo, Los datos se envía a través de señales eléctricas entre actuadores, sensores y otras ECU a través del protocolo de comunicación CAN bus [20] [40].

El sistema OBDII tiene la función de detectar fallos químicos mecánicos y eléctricos que afecten las emisiones contaminantes del vehículo, se puede extraer información de diferentes parámetros del vehículo, como lo es la velocidad, rpm, temperatura, presión del combustible, entre otros [28].

Con el CAN bus en conjunto con el OBDII podemos recolectar la información antes mencionada de los diferentes sistemas del vehículo, dicha información se puede clasificar en diferentes datos como se muestra en la figura [20][28]. Estos datos se pueden almacenar en servidores en la nube, lo que permite realizar las siguientes acciones:

- Un potencial uso para desarrollar proyectos IoV aplicados a mejora de la seguridad, infotaiment y confort [41].
- Hacer un diagnóstico en línea [28].
- Mantenimiento preventivo [40].
- Ofrecer la información a organizaciones que requieran datos del comportamiento del conductor [40].

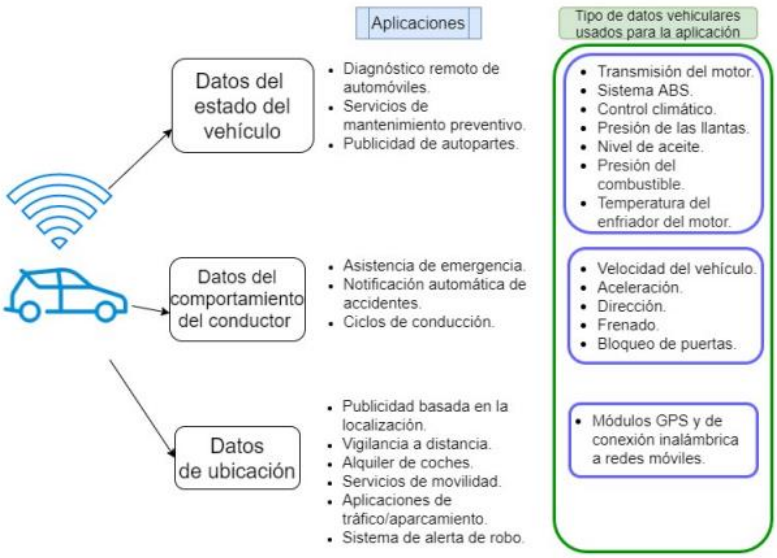

Figura 1. Datos posibles para obtener del vehículo.[25]

1.1 Objetivos

1.1.1 Objetivo general

Implementar un sistema de recolección de información de los sensores involucrados en el correcto funcionamiento del motor, como lo son el ECT (engine coolant temperatura sensor), VSS (vehicle speed sensor), CKP (Crankshft position Sensor), mediante el protocolo CAN, para su envió y almacenamiento en la nube y su potencial aplicación en la obtención de estados de alerta, diagnósticos y fallas a los conductores.

1.1.2 Objetivos específicos

- Realizar un algoritmo y programar un microcontrolador con controlador CAN para que sea compatible con la arquitectura del CAN bus.
- Implementar una comunicación con el CAN bus a través del OBDII para obtener los códigos PID con la información de los distintos módulos del automóvil.
- Crear un espacio en la nube para almacenar la información obtenida del automóvil haciendo uso de un módulo wi-fi.
- Realizar pruebas de funcionamiento del hardware y software con un vehículo VW beeatle

1.2 Estado del arte

Existen diferentes proyectos, dispositivos y aplicaciones para la recolección de datos a través del CAN bus y en los años recientes es está implementado el IoV, usando envió de datos a través de internet. Los artículos más conocidos y fáciles de encontrar en el mercado son los escáneres automotrices, que son dispositivos electrónicos que tienen la finalidad de detectar y mostrar todos los diferentes tipos de códigos OBDII (DTC) que se encuentran en la computadora del vehículo, con el objetivo de determinar un cierto tipo de falla [5].

Actualmente existe gran variedad de escáneres con diferentes características, tamaños y precios, como se muestran en la siguiente tabla. Sin embargo, no son de fácil acceso al encontrase únicamente en talleres y siendo en el mayor de los casos costosos, en todo caso, se espera que el sistema a desarrollar en este proyecto sea mucho más económico, no obstante, no contará con todas las funciones que llegan a tener los escáneres.

	El Preferido	Mejor Relación Calidad/Precio	Para Iniciar	Más completo	Más Vendido	Más Barato
Imagen						
	FOXWELL NT301 OBD2 Scanner Professional Mechanic OBDII Diagnostic...	ANCEL FX2000 Enhanced Four-System Diagnostic Scanner, Premier...	LAUNCH Creader 3001 OBD2 Scanner, Engine Fault Code Reader Mode 6...	LAUNCH CRP129X OBD2 Scanner 7.0 Android Scan Tool with...	BlueDriver Bluetooth Pro OBDII Scan Tool for iPhone & Android	OBD2 Scanner TOPDON AL200 Car Code Reader, Auto Check Engine...
Valoraciones	★★★★½	★★★★½	★★★★½	★★★★½	★★★★½	★★★★½
Opiniones	20,232 Opiniones	2,539 Opiniones	6,086 Opiniones	1,338 Opiniones	41,683 Opiniones	1,513 Opiniones
Prime	✓Prime	✓Prime	✓Prime	✓Prime	✓Prime	✓Prime
Precio	$69.99	$141.08	$25.00	$259.00	$119.95	$18.99

Tabla 1. Diferentes tipos de escáner. [5].

También tenemos en la red varios trabajos en donde se realiza una comunicación CAN usando un microcontrolador, actualmente existen microcontroladores que ya cuentan con un controlador CAN y con librerías especializadas para trabajar con este protocolo de comunicación, realizando la programación adecuada junto con un dispositivo conocido como transceptor CAN se puede desarrollar un sistema capaz de enviar y recibir datos a través del CAN bus, algunos trabajos de este tipo se pueden encontrar en las siguientes referencias:[27], [17], [2], [3]. En donde se ocupan tarjetas como Arduino, Raspberry pi e incluso con ESP32, pero este programado mediante phyton y con la función de conversor obdII-WiFi.

Algunos ejemplos de aplicaciones IoV más desarrolladas son las siguientes:

CarPlay

Es una aplicación sacada por iOS, que integra en el automóvil todos los mapas de Apple, además de la navegación por voz, el celular, mensajes y el reproductor musical [41].

Google Android Auto

Similar que su contra parte, Android proporciona una interface libre de distracciones, que integra los servicios de android al vehículo, como llamadas, reproductor musical, navegación en Internet y mapas [41].

GStress

Gstress es un modelo un modelo que estima el estrés del conductor a través de la información adquirida por los sensores GPS y sensores AutoSense, así los conductores serán conscientes de su estrés a la hora de manejar y poder hacer una estrategia para evitarlo, incluso se pueden bloquear o posponer llamadas en caso de que el conductor este estresado, si se usa en conjunto este modelo se puede hacer un mapeo de zonas de estrés similar a las actualizaciones de tráfico [31].

Usage-based-insurance (UBI)

El seguro basado en el riesgo, por sus siglas en inglés (UBI), se basa en usar la información obtenida de IoV para generar un seguro vehicular basado en el riesgo de conducción, donde entre más riesgo mayor será le precio. Se divide en dos categorías; Pay-as-you-drive (PAYD) el cual se basa en el kilometraje y el combustible consumido. Por otro lado, tenemos Pay-how-you-drive (PHYD), que comparado con PAYD, este toma en consideración factores del conductor como lo son la aceleración, el frenado, las curvas y la velocidad, lo que permite calcular el riego con mayor precisión y así proporcionar la prima de seguro adecuada [31].

1.3 Justificación

En México se registraron casi 36.4 millones de unidades automóviles para finales del año 2022 [18], sin embargo, la gran mayoría son modelos antiguos y de gama media, por lo que no cuentan las nuevas tecnologías de IoV, por lo que para tener una idea de la condición del vehículo se lleva con personal especializado en el área automotriz, lo cual ocurre frecuentemente hasta que ya hubo alguna avería y tiene las desventajas de consumir dinero, tiempo y vida útil del vehículo.

Una de las ventajas de este trabajo es tener fácil acceso a la información de un vehículo. Al extraer los datos de los diferentes sistemas del automóvil y almacenarlos en la nube en tiempo real, se puede hacer un análisis de datos y monitorear los diferentes parámetros en tiempo real, y así, a través de internet hacer llegar al conductor posibles averías y datos anormales del automóvil. Permitiendo un diagnóstico preventivo, envió de notificaciones y registros en caso de obtener datos que requieran de atención, como cuando hay poco combustible, alguna irregularidad en la batería, fallo en algún sensor, sobre calentamiento, entre otras cosas.

Con este trabajo se pretende recuperar y tener acceso a los datos proporcionados por los sensores de manera rápida, económica y sencilla, para posteriormente trabajar y analizar la información adquirida para desarrollar diferentes programas, modelos y aplicaciones, como se mencionó anteriormente.

1.4 Metodología

Al realizar el proyecto se siguió la metodología mostrada en la figura 2. El primer paso consta en hacer una investigación sobre las características y normativas presentes en los protocolos CAN y OBDII, así como del hardware y software requeridos para el trabajo.

Una vez revisado lo necesario para realizar el trabajo, se seleccionaron los elementos que mejor se acoplaban al trabajo, para luego realizar el diseño del sistema se tuvieron que hacer las conexiones entre dispositivos, descargar librerías, estudiar y hacer las configuraciones necesarias para realizar código, así como para tener un espacio en la nube y una interface en una app móvil.

Después, se realizaron diferentes pruebas en las acciones esenciales ejecutadas por el sistema, con el objetivo de observar que funciona correctamente y en caso contrario hacer las correcciones necesarias, para finalmente, hacer una prueba final de todo el sistema en un automóvil real.

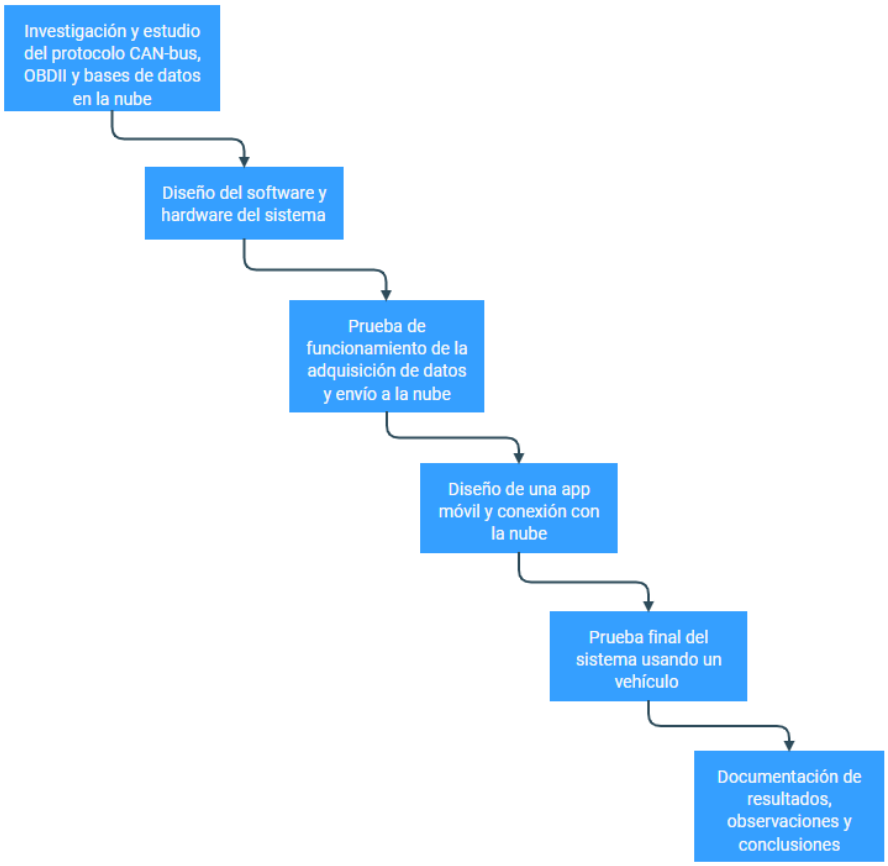

Figura 2. Diagrama de la metodología seguida

1.5 Organización de la tesis

El presente documento se divide en 5 capítulos. En el primer capítulo se muestra una introducción relacionada con el trabajo a desarrollar. En el capítulo 2 se realizó una investigación de los fundamentos teóricos implícitos en el protocolo de comunicación CAN, así como la adquisición y envió de datos a la nube. Partiendo desde el OBDII, dispositivos necesarios para comunicarse al CAN-bus y la nube, además del espacio en un servidos de la nube.

En el capítulo 3 se empieza con el diseño del sistema, tanto hardware como software, mostrando cada parte que compone el sistema y haciendo las modificaciones, adiciones y acciones necesarias para el funcionamiento del sistema, para después, en el capítulo 4 realizar algunas pruebas de funcionamiento en ciertos procesos del sistema, para analizar y modificar de ser necesario, así como una prueba final en un vehículo real una vez que el sistema sea optimo.

Finalmente, en el capítulo 5 se muestran las conclusiones y observaciones obtenidas de los resultados del trabajo de tesis, además de los posibles trabajos a futuro y mejoras que se pueden implementar a partir de este trabajo.

Capítulo 2

Fundamentos teóricos

En este capítulo se describirán las tecnologías utilizadas para la transmisión y recuperación de datos en el vehículo mediante el sistema OBDII y el CAN-bus, el almacenamiento en una base de datos en la nube y la interface en una aplicación móvil.

2.1 Sistema OBDII

En año 1996 la industria automotriz integra en todos sus automóviles un sistema de diagnóstico electrónico llamado OBD II (On Board Diagnostic Second Generation). Este sistema permite el monitoreo del motor y otros dispositivos del vehículo, detecta, almacena y evalúa fallos detectados por las unidades de control que tienen un automóvil, mediante los llamados códigos de error DTC (Dianostic Trouble Code) adicionalmente permite el acceso a dicha información a través de la interfaz DLC (Data Link Connector) y de un único conector. En un principio se utilizaba para monitorear las emisiones contaminantes, pero en la actualidad se ocupa monitorear el motor, el sistema ABS, entre ostros sistemas; para así diagnosticar un problema sin la necesidad de desmontar partes del automóvil [9] [2].

2.1.1 Conector

El conector del OBDII debe ser de fácil acceso y la mayoría de las veces se encuentra en la zona del conductor, por debajo del cuadro de instrumentos. En la Figura 3 se muestra el conector OBDII y la numeración de sus terminales. Este cuenta con 16 terminales las cuales tiene una función específica, como se muestra en la Tabla 2.

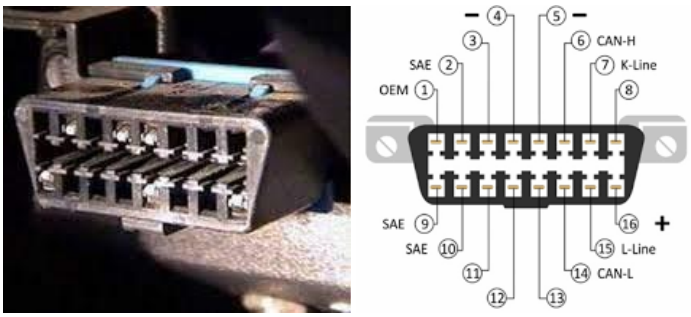

Figura 3.A la izquierda se muestra el conector OBDII y a la derecha las características correspondientes a cada pin [2].

Terminal	Función
1	Uso del fabricante
2	Bus (+) SAE-J1850 VPM y PWM

3	Uso del fabricante
4	Negativo de la batería
5	Tierra de la señal
6	CAN (high) ISO-15765-4
7	Línea K ISO 9141-2
8	Uso del fabricante
9	Uso del fabricante
10	Bus (-) J1850
11	Uso del fabricante
12	Uso del fabricante
13	Uso del fabricante
14	CAN (low)
15	Línea L ISO 9141-2
16	Positivo de la batería

Tabla 2. Correspondencia de los pines del OBDII

La mayoría de los vehículos implementan únicamente uno de los protocolos mencionados en la tabla anterior, pero ocasionalmente pueden soportar más de uno. En Estados Unidos a partir del 2008 todos los vehículos deben poder intercambiar mensajes del OBDII a través de ISO 15765-4 (CAN bus) [3].

2.1.2 Modos de medición

Uno de los puntos a destacar del OBDII es que se indican algunas recomendaciones sobre ciertos datos obtenidos, externos a los fallos detectados que provoquen una contaminación suprior a la permitida, pero deja la libertad al fabricante de compartir esos datos o no. El sistema OBD II utiliza nueve modos de medición, aunque los fabricantes no están obligados a incluir todos y dependiendo del modo seleccionado se accede a cierto tipo de información de la ECU del automóvil. Cada modo utiliza los denominados PIDs (Parameter Identification), que son códigos que piden información al automóvil [3] [2]. A continuación, se describen los diferentes modos:

Modo 1. Muestra la Identificación de Parámetro (PID), es el acceso a datos en tiempo real de valores analógicos o digitales de salidas y entradas a la ECU. Este modo es también llamado flujo de datos. Aquí es posible ver, por ejemplo, la temperatura de motor, velocidad, RPM, etc.

Modo 2. (Freeze Frame) Muestra los PID, que son los mismos que en el modo 1, con el mismo significado, pero el acceso es al cuadro de datos congelados, con los valores de los sensores al momento de ocurrir una falla. Esta es una función muy útil del OBD-II, porque al recuperar estos datos, se pueden conocer las condiciones exactas en las que ocurrió dicho fallo. El cuadro de datos corresponde al primer fallo detectado.

Modo 3. Este modo permite extraer de la memoria de la ECU todos los códigos de fallo (DTC) almacenados. Se usa en talleres de mantenimiento y reparación de vehículos, donde utilizan los escáneres que trabajan principalmente con este modo.

Modo 4. Con este modo se pueden borrar todos los códigos almacenados en la PCM y el cuadro de datos guardados en la memoria del modo 2.

Modo 5. El propósito de este modo es permitir el acceso a los resultados de las mediciones de voltaje hechas a los sensores de oxígeno del módulo de control del motor, y así determinar el funcionamiento de los mismos y la eficiencia del convertidor catalítico. La información de este modo no está disponible en vehículos que utilizan el sistema CAN. Para esos vehículos se requiere utilizar el modo 6.

Modo 6. El propósito de este modo es permitir el acceso a los resultados de las pruebas hechas a componentes que no están monitoreados continuamente, pero sólo en periodos de tiempo y condiciones de operación determinadas, estos resultados reportan rangos de valores máximos y mínimos. También muestra las pruebas hechas a los sensores de oxígeno de la misma manera que en el modo 5 pero sólo para CAN.

Modo 7. Este modo permite leer de la memoria de la ECU todos los DTCs pendientes. Este modo es usado por los técnicos automotrices posterior a la reparación del vehículo ya que les permite ver los resultados después de borrar la información de diagnóstico y luego de un ciclo de conducción, para así validar si el problema fue corregido

Modo 8. Este modo nos permite realizar un control de operación a bordo de componentes y/o sistemas del vehículo, así, este modo permite realizar la prueba de actuadores, con lo que el mecánico puede activar y desactivar actuadores como bombas de combustible, válvula de ralentí, etc.

Modo 9. Solicita información del vehículo como el VIN (Vehicle Identification Number). Este número es una secuencia de dígitos que identifica los vehículos de motor de cualquier tipo y es un código específico y único para cada unidad fabricada.

Cuando se hace una comunicación con el sistema OBDII se debe hacer mediante uno de los ya mencionados modos. La información solicitada se envía dentro de una trama de cualquiera de los protocolos que se usan, esta trama incluye, el modo de funcionamiento con el cual se trabajará, así como cuál es la información que se requiere. Con la finalidad de especificar el dato que se pide se hace uso de los PID, algún dispositivo del sistema OBD-II conectado en el bus reconoce ese PID como uno al que tiene que responder, y reporta el valor para ese PID por medio del mismo bus. La mayoría de los datos están codificados con una fórmula, otros están codificados de forma especial [2].

En el caso de este proyecto se usará el modo de funcionamiento 1, para un monitoreo a tiempo real junto con el protocolo CAN (Controller Area Network).

2.2 Protocolo CAN

El CAN es un protocolo de comunicación desarrollado por Bosch en 1980 y surgió por la necesidad de reducir el cableado en el automóvil, es una red que funciona para comunicar los diferentes nodos (Unidades de Control) que existen en el vehículo, las cuales requieren recibir y enviar datos de otros

nodos para el correcto funcionamiento de los sistemas en el automóvil. Este protocolo es obligatorio en todos los vehículos en Estados Unidos a partir del 2008 [17].

En la figura 4 se muestra un ejemplo de una red con conexión CAN bus.

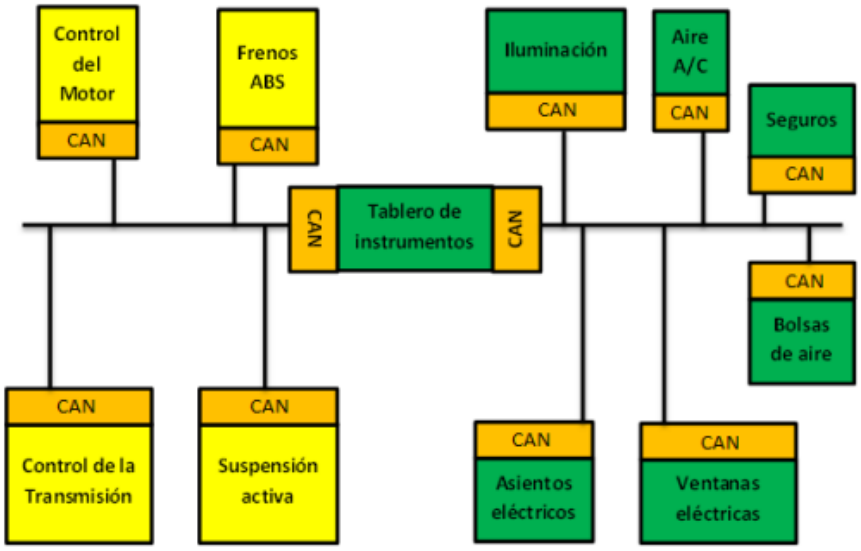

Figura 4. Ejemplo de red CAN bus [17]

Algunas de las principales características que posee el protocolo que lo han hecho destacar en la comunicación automotriz [17], son las siguientes:

- Implementación de un protocolo para la combinación del manejo de error y aislamiento de fallas con una alta velocidad de transmisión.
- Presenta una red distribuida donde todos los nodos se conectan al bus, evitando así exceso de cables.
- El estándar es un cable de par trenzado y una señal diferencial, lo que otorga mayor protección contra interferencias electromagnéticas.
- El protocolo se enfoca en los mensajes, todos los mensajes cuentan con un identificador y dependiendo de este cada nodo decidirá si lee el mensaje o lo deja pasar.
- Detección de errores con CRC (cyclic redundancy check)
- Es un sistema robusto a fallos de un nodo de la red, tiene la característica de que si un nodo falla no se bloquea el sistema y sigue funcionando con normalidad.
- Protocolo más usado en el mundo industrial y automotriz.
- Cualquier nodo puede empezar a transmitir si el bus esta libre y cuenta con un sistema de arbitraje donde se envía primero el mensaje que cuenta con mayor prioridad en el identificador.

Algunas desventajas que se deben mencionar sobre el bus CAN, son que presenta un costo elevado y que se requieren conocimientos más complejos cuando se le da mantenimiento.

Cabe mencionar que el protocolo bus CAN trabaja bajo modelo OSI (Open System Intecnection); específicamente la capa de enlace de datos y la capa física. Estas capaz están estandarizadas conforme a ISO-11898, para velocidades de hasta 1Mbps, denominado CAN de alta velocidad e ISO-11519 para velocidades de hasta 125kbps, CAN de baja velocidad [9].

A continuación, se describe el funcionamiento y características de trabajo del bus CAN:

2.2.1 Capa de enlace

Esta capa se encarga de los filtros con los cuales cada nodo decide qué mensajes enviados en el bus se aceptan. Proporciona servicios durante la comunicación entre los nodos, ya sea para solicitar y transmitir datos y proporciona los medios para notificar la sobrecarga del bus. En esta capa presentan los mensajes recibidos al control de enlace lógico, también es responsable de la trama de mensajes, arbitraje, reconocimiento, señalización y detección de error [17].

2.2.2 Capa física

La capa física en CAN es responsable de la transferencia de las señales (bits) entre los nodos que componen la red. Define aspectos como niveles de señal, codificación, sincronización y tiempos en que los bits se transfieren al bus. Las características de las señales eléctricas en el bus fueron establecidas por el estándar ISO 11898. La especificación CiA (CAN in AUTOMATION), complementó las definiciones respecto al medio físico y conectores [28].

El bus de comunicación de CAN consta de dos cables de par trenzado (aunque pueden ser otros medios), en los que viaja una señal diferencial, estas dos características hacen que la comunicación sea en gran medida inmune al ruido, tomando en cuenta que está sometido al ruido proveniente de la temperatura del motor, de la chispa de ignición y de todos los demás dispositivos con los que cuenta el automóvil. Todos los dispositivos (nodos) que se comunican deben hacerlo a la misma velocidad y estar conectados a los cables trenzados, llamados respectivamente “CAN-High” (línea “H”) y “CAN-Low” (línea “L”). Estos cables tienen una impedancia característica de 120 Ohms, por lo que se debe tener una resistencia de terminación del mismo valor en cada extremo para evitar que se refleje la señal [17] [2].

El protocolo CAN es multi-maestro, esto significa que todos los nodos conectados tienen la misma oportunidad de acceder al bus para enviar mensajes. La comunicación se basa en un protocolo de transmisión orientado a mensajes, esto quiere decir que define el contenido del mensaje en lugar de definir al nodo o a la dirección del nodo. Este mecanismo de comunicación permite que el acceso al bus se realice por prioridad del mensaje utilizando CSMA/CR (Carrier Sense Multiple Access/Collision Resolution). CSMA significa que los nodos tienen la misma oportunidad de enviar mensajes (MA) y también que todos están escuchando los mensajes del bus (CS). En un determinado momento más de un nodo tratará de acceder al bus, y los mensajes colisionan, esta colisión se soluciona por arbitraje basado en prioridades (CR). Este método se basa en una topología eléctrica que aplica una función lógica determinista a cada bit; la prioridad se le da a un nivel de bit de tipo dominante. Se define como bit dominante al nivel lógico ‘0’ y el valor lógico ‘1’ es un bit recesivo.

Para la solución a una colisión de mensajes se aplica una función AND a todos los bits transmitidos simultáneamente. Cada transmisor escucha continuamente el valor presente en el bus; el transmisor se retira cuando el valor en el bus es dominante y éste ha enviado un bit recesivo. Mientras el valor del bus y el enviado coincidan, el transmisor permanecerá; finalmente el mensaje con identificador de máxima

prioridad sobrevive, y los demás nodos reintentarán la transmisión lo antes posible. En concreto, la resolución de colisión (CR) hace que la comunicación sea no-destructiva y se garantiza que el mensaje que tenga la mayor prioridad no se detenga y se transmite sin perder tiempo. La prioridad del mensaje se establece dentro de la misma trama que contiene al mensaje, en una sección dentro del campo "arbitraje" llamada identificador. Cada nodo, con base en el identificador, debe decidir si el mensaje debe atenderse o ignorarse [17].

2.2.3 Nivel eléctrico

Para que los nodos conectados en el bus puedan enviar y recibir mensajes del bus, cada ECU debe contar con un controlador can en su microprocesador, el controlador es el que determina la velocidad de transmisión de mensajes y también se encarga de procesar la información que entra y sale del nodo. Por otro lado, el transceptor CAN tiene la función de recibir y transmitir los datos que vienen del bus, debe acondicionar las señales a los niveles de tensión adecuados, ya sea amplificando la señal cuando se transmite al bus o reduciéndola cuando es recogida y suministrada al controlador. Por lo tanto, el transceptor se encuentra entre la línea del bus y el controlador [17] [35].

Como se mencionó anteriormente, se denomina bit dominante al ('0') y recesivo al ('1'), estos bits tienen niveles TTL de 5V para positivo ó '1' y 0V para negativo ó '0'. Los cuales están presentes a la salida del controlador CAN, en las líneas de transmisión (TX) y recepción (Rx) correspondientes. Cuando se presenta un estado dominante el voltaje nominal en "H" es de 3.5 V y para "L" es de 1.5 V, esto arroja un voltaje diferencial nominal de 2 V; y para un estado recesivo el voltaje nominal en las líneas "H" y "L" es de 2.5 V para cada una de ellas, lo que arroja un voltaje diferencial nominal de 0 V [17].

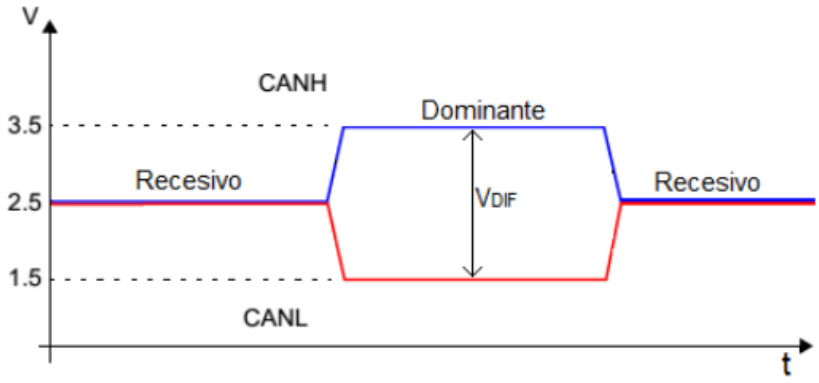

Figura 5. Niveles de voltaje en el CAN bus [17].

Cuando se utiliza el par trenzado, según ISO 11898, el nivel dominante es un voltaje diferencial positivo en el bus; el nivel recesivo es ausencia de voltaje, o cierto valor negativo. Lo que quiere decir que los niveles son definidos por el protocolo y se presentan eléctricamente, como se dijo anteriormente; entonces, el valor dominante '0' provoca un voltaje diferencial positivo en el bus a través del transceptor, y un valor recesivo '1' provoca un voltaje diferencial cero o negativo en el bus a través del transceptor. La figura 5 muestra los niveles de voltaje antes mencionados.

2.2.4 Codificación

El protocolo CAN utiliza una codificación llamada NRZ (Non Return to Zero), en la cual cada bit está representado por un solo nivel, esto quiere decir que el nivel de la señal permanece constante a lo largo del tiempo que dura un bit; un nivel lógico '0' se representa con un nivel de voltaje bajo, y un nivel lógico '1' se representa con un nivel de voltaje alto. Usar la codificación NRZ asegura mensajes compactos con un número mínimo de transiciones. Una acción que se debe mencionar en esta parte, es lo que se denomina relleno de bits (Bit stuffing), el cual se aplica en caso de que el nivel de voltaje permanezca constante durante mucho tiempo lo cual puede afectar el tiempo de sincronización del bus, y consiste en insertar un bit complementario y de valor lógico diferente después de cinco bits con el mismo valor, los nodos en el bus ignoran estos bits de relleno para leer el mensaje original. En la figura 6 podemos ver cómo funciona el relleno de bits [2].

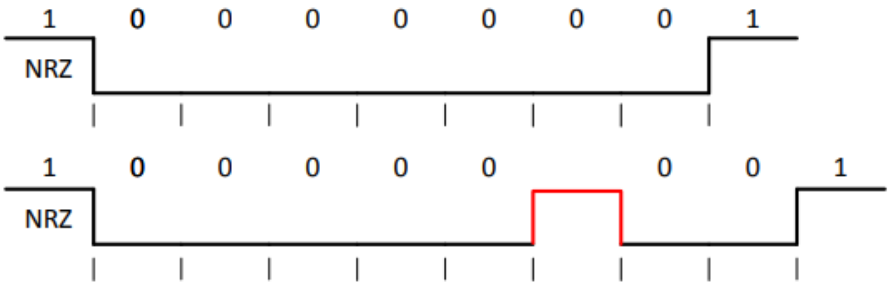

Figura 6. Ejemplo de relleno de bits [2].

2.2.5 Formato de tramas

Los mensajes enviados en el bus CAN poseen una estructura predefinida, llamada "trama". Existen dos variantes de CAN, el "CAN estándar" o CAN A y el "CAN extendido" o CAN 2.B, los cuales se diferencian uno del otro por la longitud del identificador de mensajes, donde el CAN extendido maneja 29 bits que da 536.870.912 identificadores, mientras que en CAN estándar ocupa 11bits, pero en este caso existe una especificación que dice que los 7 bits más significativos no pueden ser simultáneamente recesivos, debido a los 4 bits restantes existen 16 combinaciones donde estos siete bits son recesivos, por lo tanto, el número de identificadores que se tiene es de 2033 [17].

Existen 4 tipos de tramas usados en ambas variantes, que son: trama de datos (Data Frame), trama remota (Remote Frame), trama de error (Error Frame) y trama de sobrecarga (Overload Frame).

2.2.5.1 Trama de datos

En la siguiente figura se muestra la trama de datos y los diferentes campos que la componen.

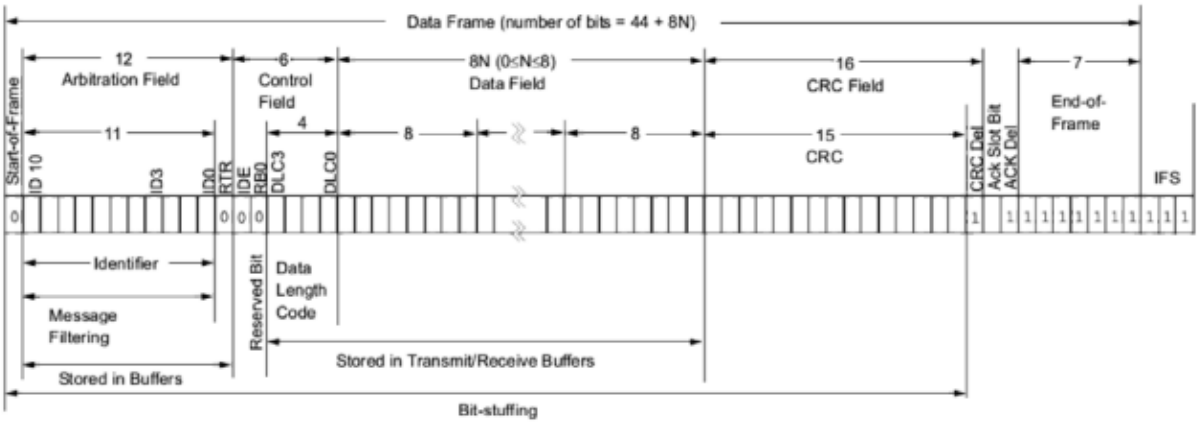

Figura 7. Trama de datos del Can bus y sus campos [35]

Como se muestra en la figura 7 la trama de datos se compone de 7 campos de bits, los cuales son: Inicio de trama (SOF – Start Of Frame), arbitraje, control, datos, CRC (Cyclic Redundancy Check), reconocimiento y fin de trama [35] [17]. A continuación se describe cada uno de ellos.

- **Inicio de trama (SOF):** El inicio de trama es un campo de un solo bit siempre dominante '0', indica el inicio de la transmisión. Los nodos receptores se sincronizan con el flanco de bajada de este bit.
- **Arbitraje:** Este campo se compone por el identificador de mensaje (11 bits) y el bit RTR (Remote Transmission Request). En una trama de datos el bit RTR es dominante, en una trama remota es recesivo. Los bits de identificador se transmiten en orden de más significativo a menos significativo.
- **Control:** El campo de control está formado por dos bits reservados, el bit IDE (Identifier Extension), que se utiliza para indicar si la trama es de CAN Estándar (IDE dominante) o Extendido (IDE recesivo) y el bit RB0 (Reserved Bit Zero), que siempre es recesivo; y cuatro bits adicionales que indican el número de bytes de datos. Los cuatro bits de código de longitud (DLC-Data Lenght Code) indican en binario el número de bytes de datos en el mensaje (0 a 8).
- **Datos:** Es un campo formado por 0 a 8 bytes de datos, es decir 0 a 64 bits. Cada byte se transmite con el bit más significativo primero. Estos datos contienen la información que la ECU desea transmitir al bus.
- **CRC:** el código de redundancia cíclica que genera lo transmisor. La división de todos los bits precedentes del mensaje, incluyendo los de relleno si existen, por el polinomio generador: $X^{15} + X^{14} + X^8 + X^7 + X^4 + X^3 + X^1 + 1$, el residuo de esta división es el código CRC transmitido y tiene un máximo de 15 bits. Los receptores comprueban este código. Después del código CRC se añada un bit recesivo (delimitador de CRC).
- **Campo de reconocimiento (ACK):** Este es un campo conformado por dos bits que el transmisor envía como recesivos. El primero de estos bits es cambiado a un bit dominante por los nodos que han recibido el mensaje correctamente y el bit siguiente queda siempre como recesivo.
- **Fin de trama (EOF):** El fin de la trama, consiste en 7 bits recesivos sucesivos.

Finalmente, al término de cada trama se agrega un espaciado entre tramas (IFS), que consta de un mínimo de 3 bits recesivos.

Para la trama de datos extendida, el bit SOF se mantiene igual que la forma estándar, pero el campo de arbitraje ahora constará de 32 bits. Los primeros 11 bits son los bits más significativos del identificador (MSB, Most Significant Bits) (identificador base) de los 29 bits; Esos 11 bits son seguidos por el bit sustituto de petición remota (SRR – Substitue Remote Request), el cual es recesivo y aparece en la posición del bit RTR de la trama estándar. Después del bit SRR está el bit IDE, éste es recesivo para indicar que se trata de una trama extendida. Posterior a los bits SRR e IDE les siguen los restantes 18 bits del identificador (identificador extendido) y el bit de petición de transmisión remota [35] [17]. Como se observa en la figura 8.

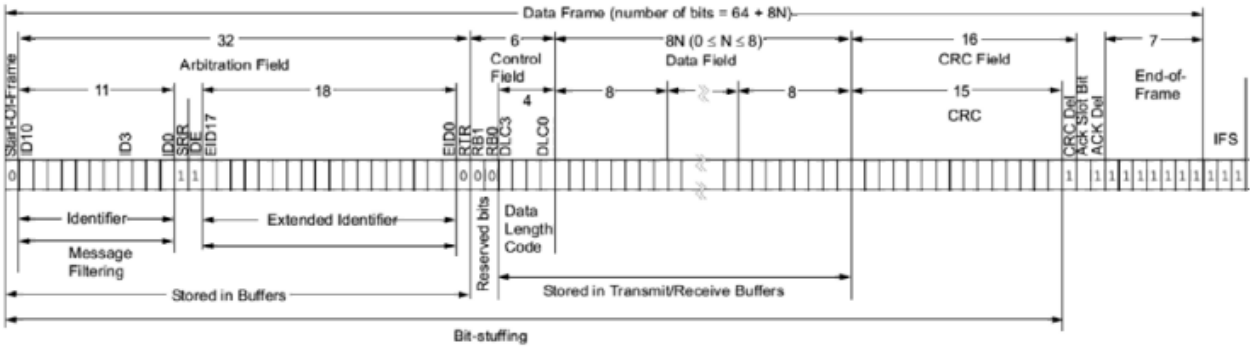

Figura 8. Trama de datos extendida [35].

La parte restante de la trama (campo de datos, campo del CRC, campo de reconocimiento, fin de trama y el espacio entre tramas) está construida de igual manera que una trama de datos estándar.

2.2.5.2 Trama remota

La trama remota es transmitida por algún nodo del bus que requiere la transmisión de una trama de datos con el mismo identificador, ósea solicita información de algún otro nodo. El formato de la trama remota es análogo a la trama de datos, pero con el bit RTR recesivo. Además, como se muestra en la figura 9. La trama remota no incluye datos (0 bytes), el identificador es el del mensaje que se solicita y el campo longitud (DLC) corresponde a la longitud del mensaje solicitad [17].

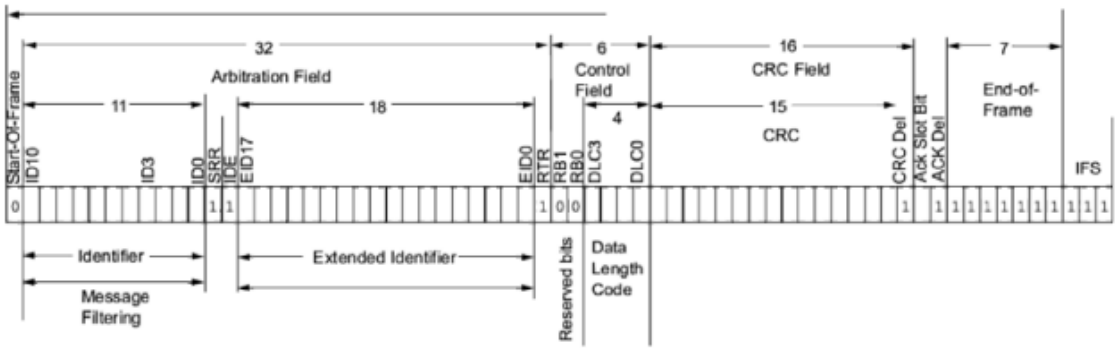

Figura 9. Trama remota [35].

2.2.5.3 Trama de error

Una trama de error es transmitida cuando algún nodo detecta alguno de los siguientes errores:

- Una trama mal formada.
- Error de acuse de recibo. Se presenta cuando el nodo no introduce el bit dominante en el campo ACK de la trama, independientemente si el nodo decide descartar los datos.
- Detección de CRC incorrecto. Todos los nodos deben calcular el CRC del mensaje, por lo que emitirán un error si este no coincide con el recibido.
- Error de bit de relleno. Ocurre cuando se detectan más de 5 bits iguales.

La trama de error es una trama diferente a las anteriores. En la figura 10 se pueden observar dos tramas; la primera es una trama de datos interrumpida por la trama de error mostrada en la parte inferior de la figura. Las tramas de error son generadas por cualquier nodo del bus al detecta un error. Consiste de dos campos: Bandera de error (Error Flag) y Delimitador de error. La bandera de error es diferente y depende del estado de error del nodo que detecta el error; los estados de error de nodo pueden ser “activo” y “pasivo”. El delimitador de error consta de 8 bits recesivos consecutivos lo que permite a los nodos reiniciar la comunicación limpiamente tras el error [35] [17].

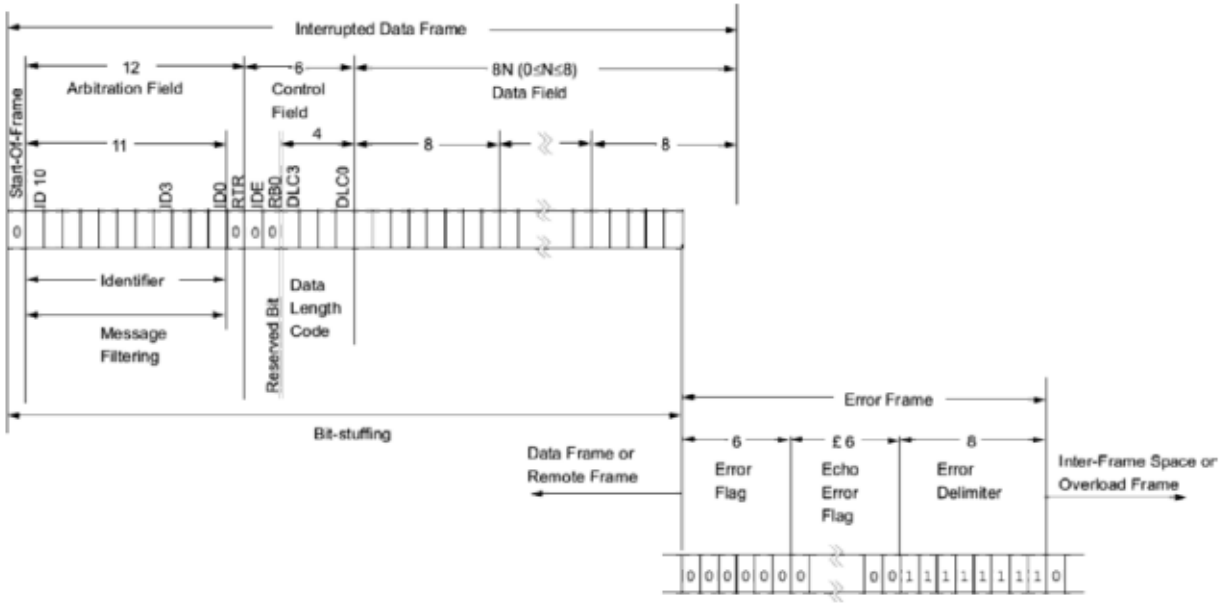

Figura 10. Trama de datos interrumpida y trama de error [35].

Cuando se tiene un nodo en estado de error "Activo"; el bus interrumpe la comunicación del mensaje en proceso, generando una "Bandera de error activo", que consiste en una secuencia de 6 bits dominantes sucesivos. Esta secuencia rompe la regla de relleno de bits y provocará la generación de tramas de error en otros nodos, por lo que la bandera de error puede extenderse entre 6 y 12 bits dominantes sucesivos. Finalmente se espera el campo de delimitación de error formado por los 8 bits recesivos. Entonces la

comunicación se reinicia y el nodo que había sido interrumpido reintenta la transmisión del mensaje [35] [17].

En el caso de temer un nodo en estado de error "Pasivo"; el nodo transmite una "Bandera de error pasivo". La Bandera de error de tipo pasivo consiste en 6 bits recesivos consecutivos y el delimitador de tipo pasivo es de 8 bit, por tanto, la trama de error para un nodo pasivo es una secuencia de 14 bits recesivos. La trama de error de tipo pasivo no afectará a ningún nodo en la red, excepto cuando el error es detectado por el propio nodo que está transmitiendo. En ese caso los demás nodos detectarán una violación de las reglas de bits de relleno y transmitirán a su vez tramas de error.

2.2.5.4 Trama de sobrecarga

Una trama de sobrecarga es generada por un nodo por las siguientes condiciones:

1. Debido a sus condiciones internas el nodo no puede iniciar la recepción de un nuevo mensaje. De esta forma retrasa el inicio de transmisión de un nuevo mensaje. Un nodo puede generar como máximo 2 tramas de sobrecarga consecutivas para retrasar un mensaje.

2. Por la detección por cualquier nodo de un bit dominante en los 3 bits de "intermisión".

La trama de sobrecarga es similar a la de error, como se muestra en la figura 11, cuenta con dos campos: la bandera de sobrecarga formada por 6 bits dominantes y el delimitador por 8 bits recesivos. Se diferencia de la trama de error, debido a que esta trama solo puede generarse durante el espacio entre tramas, a diferencia de la de error que se transmite durante la transmisión de un mensaje [17].

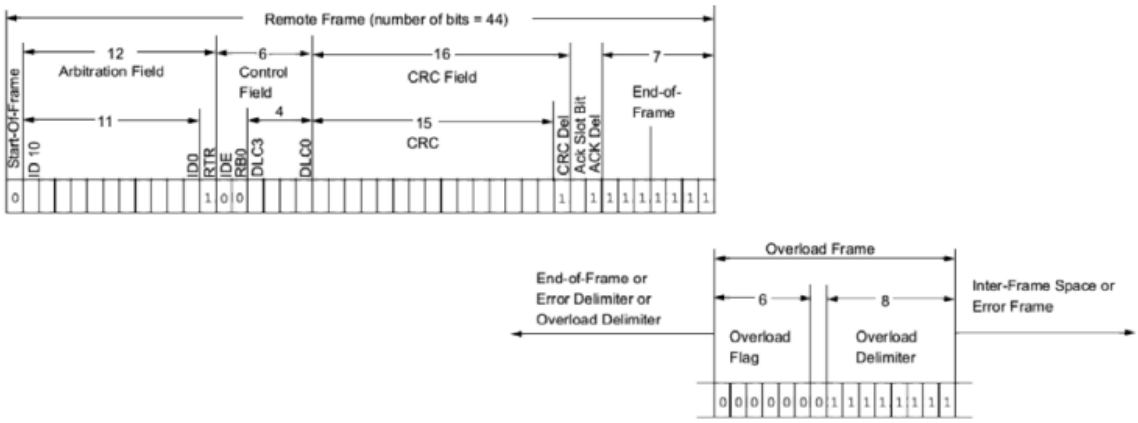

Figura 11. Trama remota y trama de sobrecargo [35].

Generalmente cuando se genera una trama de sobrecarga, los demás nodos comenzaran a generar estas tramas, dando lugar a que se permita un máximo de 12 bits dominantes en la bandera de sobrecarga.

2.2.6 Filtros y máscaras

Para rechazar paquetes, el CAN tiene los filtros y las máscaras que, dependiendo de su configuración admitiremos más paquetes o menos.

Si un dispositivo tiene el ID 165h y se configura el filtro en 165h y la máscara 1FFFFFFFh, con esto sólo pasarán los paquetes con ID165h. Si se cambia la máscara a 1FFFFFF0h, pasarán los paquetes con ID del 160h al 16Fh y si se pone una máscara del 1FFFFFF5h, pasarán todos con ID comprendidas entre 160h a 164h. Para que pasen todos los ID, se debe poner un 0 tanto en el filtro y en la máscara [2].

2.2.7 Tiempo de bits

En una red CAN, todos los módulos deben tener la misma velocidad de bits, sin embargo, no es necesario que tengan la misma frecuencia de oscilador del reloj. Debido a esto el tiempo de bits debe ser ajustado, estableciendo un pre-escaldor del "Baud rate" y el número de cuantos de tiempo en cada segmento [17].

El tiempo de bits en CAN se compone de segmentos no superpuestos, cada uno de estos segmentos está conformado por unidades de tiempo llamados "Cuantos de tiempo" (TQ – Time Quanta). Este está relacionado inversamente con la velocidad nominal de bits (NBR – Nominal Bit Rate). Que se define como el número de bits transmitidos por segundo por un transmisor ideal sin resincronización [17]. Y se describe con la ecuación siguiente:

$$NBR = f_{bti} = \frac{1}{t_{bit}}$$

Para que exista una correcta comunicación entre los nodos del CAN es impórtate que cada nodo se sincronice en los flancos de los bits, para que así que en cada uno concuerde el valor del bit transmitido actualmente en el bus. Para sincronizarse, se implementa un protocolo de sincronización que mantiene la velocidad de bits del receptor alineado con la velocidad actual de los bits transmitidos. El protocolo de sincronización usa los flancos de transición para re- sincronizar a los nodos. Una de las razones por las que pueden ocurrir desfasamientos en los bits, es cuando existe una larga sucesión de bits consecutivos, sin transición. Es por esto que se emplea el protocolo de la técnica llamada "bit de relleno", lo que fuerza a complementar el flujo después de transmitir 5 bits del mismo tipo.

La transmisión síncrona de los bits requiere un sofisticado protocolo de sincronización. La sincronización de los bits se realiza al recibir el bit de inicio disponible con cada transmisión asíncrona. Después, para permitir que los receptores lean correctamente el contenido del mensaje, se requiere de una continua re-sincronización en cada nodo. Para propósito de arbitraje del bus, reconocimiento del mensaje y señalización de error, el protocolo requiere que los nodos puedan cambiar el estado de un bit transmitido de recesivo a dominante, informando del cambio de estado del bit a todos los demás nodos de la red antes de finalizar la transmisión.

El tiempo del bit incluye un segmento que toma en cuenta la propagación de la señal en el bus, así como los retardos causados por la transmisión y recepción en los nodos. En la práctica esto significa que la señal de propagación se determina por los dos nodos dentro del sistema que están más apartados entre sí.

El flanco del bit de un nodo transmisor alcanza a un nodo receptor, después la señal se propaga por todo el camino desde los dos nodos. En este punto el receptor puede cambiar su valor de recesivo a dominante, pero el nuevo valor no alcanzará el transmisor hasta que la señal se propague por todo el camino de

regreso; sólo entonces el primer nodo puede decidir si el nivel de su propia señal (en este caso recesivo) es el valor actual en el bus o si éste ha sido remplazado por un nivel dominante por otro nodo [35] [17].

El tiempo nominal de bit (NBR) ($\mathbf{t_{bit}}$) está compuesto por segmentos no superpuestos, que son los siguientes: “Segmento de sincronización” (SyncSeg), “Segmento de propagación” (PropSeg), “Segmento de fase1” (PS1) y “Segmento de fase2” (PS2), como se muestra en la figura 12. El NBR es la suma de los segmentos ya mencionados.

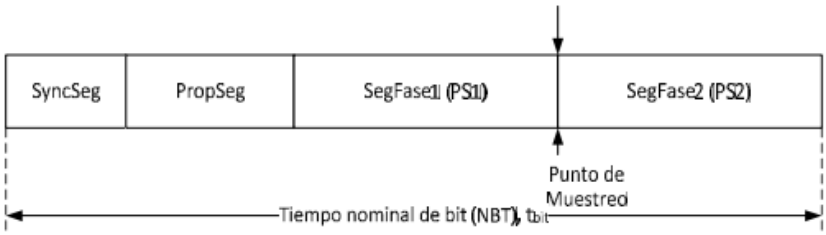

Figura 12. Partes del tiempo nominal de bit.

A continuación, se describen estos segmentos, en los que se contempla lo mencionado anteriormente sobre la sincronización de los nodos [17].

Segmento de sincronización (SyncSeg): Es el primer segmento en el NBT y se usa para sincronizar a los nodos en el bus. Se espera que los flancos de los bits ocurran en este segmento.

Segmento de propagación (PropSeg): Este segmento sirve para compensar los retardos físicos dentro de la red. El retardo de propagación se define como dos veces la suma del tiempo de propagación de la señal en el cable del bus, el retardo del comparador de entrada y el retardo del controlador de salida.

Segmento de fase 1 y 2 (PS1 y PS2 respectivamente): Los dos segmentos de fase se usan para compensar el error de fase de los bordes de los bits dentro del bus. PS1 se puede alargar o PS2 acortar en la re-sincronización.

Punto de muestreo (SP): El punto de muestreo es el punto en el cual se lee e interpreta el nivel del bus como el valor respectivo del bit. El punto de muestro se localiza al final de PS1.

Tiempo de Cuanta (TQ): El tiempo de cuanta es una unidad de tiempo fija derivada del periodo del oscilador. Hay un pre-escalador programable, con valores enteros, que van desde 1 a 32. Empezando con el TIEMPO MINIMO DE CUANTA. El TIEMPO DE CUANTA puede tener una longitud de:

$$\text{TIEMPO DE CUANTA} = m * \text{TIEMPO MÍNIMO DE CUANTA}$$

En la que m es el valor del pre-escalador. El TIEMPO DE CUANTA es la mínima resolución en la definición del tiempo del bit y el error máximo asumido por el protocolo de sincronización orientado a bits. La longitud de cada segmento queda:

- SyncSeg es de 1 TQ de longitud.

- PropSeg se programa para ser de longitud 1,2,…,8 TQ.
- PS1 se programa para ser de longitud 1,2,…,8 TQ.
- PS2 con un máximo de entre PS1 y del TIEMPO DE PROCESAMIENTO DE LA INFORMACIÓN, éste último debe ser menor o igual a 2 TQ.

El número total de TQ en el tiempo de un bit puede programarse desde al menos 8 hasta 25.

2.2.7.1 Sincronización

Para compensar los desplazamientos de fase entre las frecuencias de los osciladores de cada nodo del bus, cada controlador CAN debe ser capaz de sincronizarse con el flanco relevante de la señal entrante. Cuando se detecta un flanco en el dato transmitido, la lógica comparará la ubicación del flanco en el momento previsto (SyncSeg). Entonces el circuito ajustará los valores de PS1 y PS2 como sea necesario [17]. Los dos métodos de sincronización son:

1. Hard synchronization
2. Re-sincronización

Hard Synchronización: El proceso inicia al comienzo de la trama, cuando el bit de inicio de trama SOF cambia el estado del bus de recesivo a dominante. Después de la detección del flanco correspondiente, el tiempo de bit se reinicia al final del segmento de sincronización. Por lo tanto “hard synchronization” fuerza que el flanco del bit de inicio se encuentre dentro del segmento de sincronización del tiempo de bit reiniciado.

Re-sincronización: Como resultado de la resincronización el PS1 puede alargarse o el PS2 acortarse, ver figura 13.

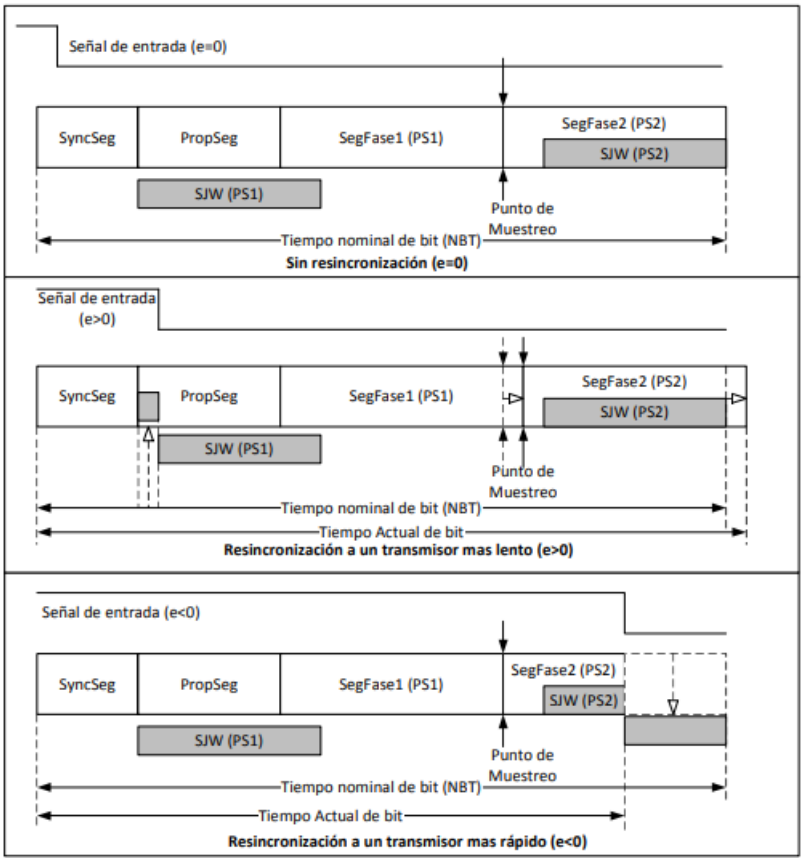

Figura 13. Estructuras tiempo nominal de bits [2]

La cantidad de alargamiento o acortamiento de PS1 y PS2 tiene un límite superiordado por SJW, el cual debe ser programado entre 1 TQ y 4TQ.

La información de sincronización sólo se puede derivar de la transición de un valorde bit a otro. Por lo tanto, la posibilidad de volver a sincronizar un nodo del bus con el flujo de bits durante una trama depende de la propiedad de que exista un intervalo máximo de tiempo entre dos transiciones de bit (aquí la importancia del protocolo de bits de relleno).

2.3 Protocolo SPI

El protocolo SPI (Serial Peripheral Interface) es un protocolo de datos en serie síncrono, usado en microcontroladores para comunicarse rápidamente con uno o más dispositivos periféricos a distancias cortas. También se puede usar para comunicación entre microcontroladores [2].

El SPI es un bus sobre el cual se transmiten paquetes de información. Cada dispositivo conectado al bus puede actuar como transmisor y receptor al mismo tiempo, por lo que este tipo de comunicación serial es *full dúplex*. Dos de estas líneas trasfieren los datos (una en cada dirección) y la tercera línea es la del reloj. Algunos dispositivos sólo pueden ser transmisores y otros sólo receptores, generalmente un dispositivo que transmite datos también puede recibir. En la figura 14 se muestra un ejemplo de una conexión SPI con un maestro y tres esclavos.

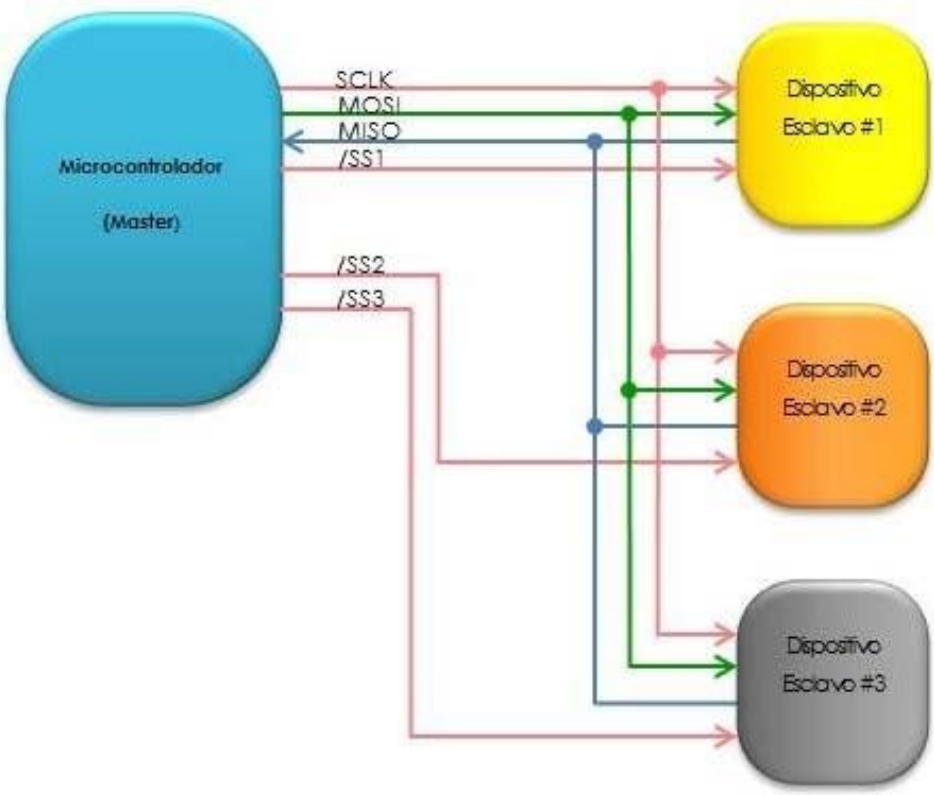

Figura 14. Diagrama de bloques del Protocolo SPI [2].

Las tres líneas más comunes para los dispositivos que ocupan este protocolo de comunicación son:

- **MISO** (Master In Slave Out) – Línea del esclavo para enviar datos al maestro.
- **MOSI** (Master Out Slave In) – Línea del maestro para enviar datos a los periféricos (esclavos).
- **SCK** (Serial Clock) – Pulsos de reloj generados por el maestro que sincronizan la transmisión de datos.

Y una línea específica para cada dispositivo:

- **SS** (Slave Select) – es la terminal en cada dispositivo que el maestro puede usar para habilitar y deshabilitar dispositivos específicos.

Cuando la terminal SS de un dispositivo se selecciona mediante un estado “bajo”, éste se comunica con el maestro. Cuando está en estado “alto" el dispositivo ignora al maestro. Esto permite tener múltiples dispositivos SPI compartiendo las mismas líneas, MOSI, MISO y CLK.

2.4 Microcontrolador

Un microprocesador puede definirse como una Unidad Central de Procesamiento (CPU), formada por una Unidad Aritmética-Lógica (ALU: Arithmetic Logic Unit), una unidad de control y algunos registros de transferencia, todo encapsulado en un solo circuito integrado [30].

Un microcontrolador es un sistema que incluye en un solo circuito integrado a un microprocesador y a un conjunto de subsistemas como memoria, puertos de entrada y salida, comunicación serie, convertidor analógico–digital, unidad de tiempo, oscilador, temporizador, entre otros.

Algunas de las características internas más importantes con los que cuenta un microcontrolador se mencionan a continuación:

- **Reloj del sistema**: El microcontrolador ejecuta las instrucciones del programa que se le cargó a cierta velocidad, la cual está determinada por la frecuencia del oscilador o también llamado "reloj del sistema". Dicha frecuencia puede ser generada a través de osciladores basados en cristal de cuarzo.

- **Memoria de programa:** Es el espacio designado para almacenar las instrucciones que constituyen al código del programa. Es de tipo no-volátil, por lo general de tipo EEPROM o FLASH.

- **Memoria de lectura y escritura:** Es la memoria utilizada por el microcontrolador para el almacenamiento de datos y variables. Generalmente es la memoria RAM.

- **Módulo USART (Universal Synchronous Asynchronous Receiver Transmitter):** El módulo USART proporciona al microcontrolador la capacidad para comunicarse a través de las interfaces seriales RS–485 y RS–232.

- **Puerto E/S digital:** Los puertos de entrada y salida digital proveen terminales que permiten controlar y recibir información de dispositivos periféricos, como teclados, display's, relevadores, motores y algunos otros.

- **Puerto de conversión analógico–digital:** Estos puertos permiten la conversión de señales analógicas a digitales; con ello se obtiene un número binario proporcional al voltaje de entrada detectado. Es utilizado para manipular la señal proveniente de algunos sensores.

- **Temporizador (Timmer):** El temporizador es implementado como un contador que establece el tiempo preciso para la realización de alguna acción ante evento o rutina. También es útil como contador de eventos.

2.5 Cuadro de instrumentos

El cuadro de instrumentos o tablero se conforma por un conjunto de indicadores que nos permiten conocer información esencial sobre el funcionamiento y estado de nuestro vehículo. Esto mediante diferentes maneras como luces, sonidos, agujas y en los autos más recientes pantallas digitales. Los indicadores se encuentran en un tablero enfrente del conductor para facilitar su visibilidad. El cuadro de instrumentos

recibe información de los diferentes sensores ubicados en el automóvil a través del CAN bus. La información que podemos encontrar en el tablero es diferente dependiendo el fabricante del vehículo [1], se pueden distinguir 4 categorías y se enlistan a continuación:

Funcionamiento técnico del coche

- Tacómetro o contador de revoluciones
- Estado del líquido refrigerante
- Nivel de combustible en el depósito
- Nivel de carga del acumulador
- Presión del aceite de motor
- RPM
- Presión de los neumáticos
- Estado de la batería

Señales de alerta

Estas señales avisan al conductor acerca de los sistemas que están bajo su responsabilidad una vez que el coche está en marcha:

- Luces encendidas
- Freno de mano
- Puertas abiertas
- Cinturón de seguridad
- Estado del airbag

Señales de alarma

Estos indicadores muestran los sistemas del coche que requieren de mantenimiento y que pueden poner en peligro la integridad del coche y de los pasajeros.

- Fallo en el sistema de frenos
- Falta de aceite en el motor
- Bajo nivel de combustible
- Temperatura alta del motor
- Estado del líquido refrigerante
- Problemas en la dirección
- Fallo en el motor

Reglamentación vial

- Indicador de velocidad o velocímetro
- Indicador de la distancia recorrida o cuentakilómetros

2.6 Sistemas PXI

Los sistemas PXI (PCI eXtensions for Instrumentation) fueron desarrollados y especificados en 1997 por National Instruments. Son una plataforma basada en PC (hardware y software) que proporcionan instrumentos modulares de alto rendimiento y bajo costo. Cuentan con una sincronización especializada para trabajar con sistemas de medida y pruebas de automatización. Sirven para tener un mejor enfoque en pruebas de producción y validación, cumpliendo con los requisitos de temporización, sincronización y rendimiento de diferentes instrumentos. PXI también agrega características mecánicas, eléctricas y de software que definen sistemas completos para aplicaciones de prueba y medición, adquisición de datos y fabricación [24].

Los sistemas PXI se componen de tres componentes: Chasis, controlador del sistema y módulos periféricos. El software del sistema es reconfigurable y personalizable. En la siguiente figura se muestran las partes del que conforman un sistema PXI.

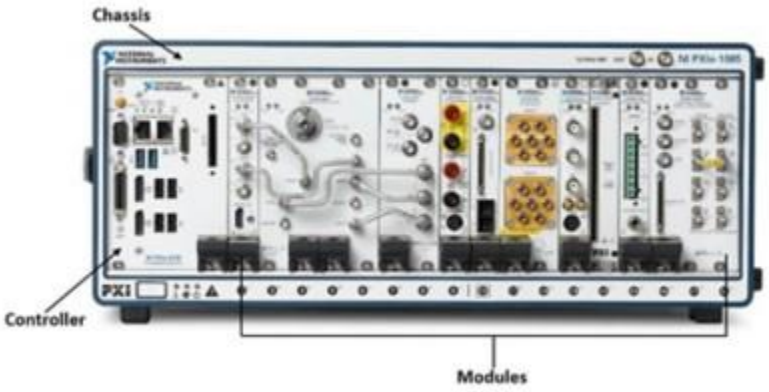

Figura 15. Partes que conforman los sistemas PXI [24]

Chasis: Es una cubierta metálica similar a las de las PC, proporciona alimentación, refrigeración y un bus de comunicación, también tiene espacio para contener varios módulos de instrumentación [24].

Controlador: Los controladores pueden ser embebidos o remotos. Los controladores integrados permiten ejecutar el PXI sin una PC externa, mientras que los controladores remotos permiten controlar el PXI desde PCs [24].

Módulos: Los módulos PXI adquieren datos, activan y sincronizan dispositivos, generan y enrutan señales y realizan una variedad de medidas que van desde DC a onda milimétrica. Además, los portafolios de PXI incluye instrumentos modulares, como osciloscopios y multímetros digitales, que pueden reemplazar los instrumentos tradicionales y con los que se pueden integrar conmutadores PXI en una variedad de topologías [24].

2.7 Base de datos en la nube

Una base de datos en la nube, es un servicio de base de datos que está disponible a través de una plataforma proveedor de servicios en la nube, estas pueden ser en plataformas públicas, privadas o híbridas. Al igual que las bases de datos tradicionales, las bases de datos en la nube permiten almacenar, organizar, recuperar y entregar datos estructurados, semiestructurados y no estructurados a sus usuarios finales [4] [26].

Las bases de datos en la nube se pueden ofrecer como un servicio administrado (DBaaS) o implementada en una máquina virtual (VM) basada en la nube, y autoadministrada por un equipo de TI propio [4].

2.7.1 Tipos de bases de datos en la nube

Las bases de datos en la nube se pueden clasificar en bases de datos relacionales y no relacionales.

- Las **bases de datos relacionales en la nube** consisten en una o más tablas de columnas y filas, que te permiten organizar los datos en relaciones predefinidas para comprender la relación lógica de los datos. Por lo general, estas bases de datos usan un esquema de datos fijo, y puedes usar el lenguaje de consulta estructurado (SQL) para consultar y manipular datos. Son muy coherentes, confiables y son más adecuados para manejar grandes cantidades de datos estructurados [26].

- Las **bases de datos no relacionales en la nube** almacenan y administran datos no estructurados, como mensajes de texto y correos electrónicos para mensajes móviles, documentos, encuestas, archivos de rich media y datos de sensores. No siguen un esquema definido claramente, como las bases de datos relacionales, y te permiten guardar y organizar la información sin importar su formato [26].

Las bases de datos en la nube ofrecen flexibilidad, confiabilidad, seguridad, accesibilidad y mucho más [26]. Algunas ventajas de usar bases de datos en la nube son:

- Eliminan la administración y el mantenimiento de cualquier infraestructura física.
- Puedes iniciar una nueva base de datos en la nube o retirarla en minutos. Esto te permite probar, poner en funcionamiento y validar ideas nuevas más rápido.
- Puedes elegir bases de datos en la nube diseñadas para propósitos específicos con las capacidades y el rendimiento que coinciden con tu caso práctico específico y las necesidades de la aplicación.
- Es posible acceder a la base de datos desde cualquier dispositivo con conexión a internet y visualizar la información en tiempo real.

Algunos de los softwares- servidores más populares para trabajar con bases de datos en la nube son los siguientes:

MySQL. Es un servidor de administración de bases de datos de tipo relacional, que es desarrollado y soportado por Oracle. MySQL utiliza la Licencia Pública General de GNU, por lo que se puede descargar, utilizar y modificar a voluntad. Esto facilita su uso tanto académico como profesional. Se caracteriza por la ejecución de tareas en simultáneo tanto lectura como escritura. Además, es software libre licenciado

bajo GNU/GPL, ofrece gran velocidad de acceso a los datos y soporta múltiples motores de almacenamiento como MyISAM e INNODB [4].

Microsoft SQL Server. Es el servidor de base de datos SQL relacional de Microsoft. Es muy popular entre usuarios de la plataforma Windows Server, debido a que ofrece una compatibilidad nativa con el lenguaje de programación ASP/ASP.NET, así como con toda la suite de desarrollo de aplicaciones de sistemas operativos Windows [4].

Microsoft SQL Server cuenta con una edición gratis, ideal para aprender, desarrollar y fortalecer las aplicaciones de escritorio, de servidores web y para la redistribución por parte de fabricantes de software independiente.

Firebase. *Firebase* de Google es una plataforma en la nube para el desarrollo de aplicaciones web y móvil. Está disponible para distintas plataformas (iOS, Android y web), con lo que es más rápido trabajar en el desarrollo. Cuenta con una gran variedad de herramientas, productos, extensiones e integraciones que facilitan y permiten desarrollar y administrar apps. La desventaja más comúnmente mencionada es el precio. El inicio con el plan Spark es gratuito, sin embargo, tiene limitaciones (principalmente de número de usuarios simultáneos y de espacio de almacenamiento). Si se requiere trabajar de manera profesional se tendrá que buscar cuál es el precio [4] [15].

2.7.2 Firebase

Se escogió *Firebase* debido a sus múltiples herramientas y facilidades para el desarrollo de apps, teniendo la posibilidad de sincronizarse con *Android Studio* y de crear bases de datos, así como se verá a continuación, además de que se puede trabajar de manera gratuitita para realizar pruebas y apps simples, pero, que si se desea profundizar y desarrollar de manera profesional tendrá un costo, pero también facilidades para conseguir lanzar el proyecto. En la siguiente tabla se muestran algunas herramientas y los cargos y ventajas sobre la versión gratuita [12].

	Plan Spark Sin costo	**Plan Blaze Prepago**
A/B Testing	Sin costo	
Analitycs	Sin costo	
App Distribution	Sin costo	
App Indexing	Sin costo	
Autentificaión Autentificación (telefónica)	10 SMS enviados por día	Facturación por SMS enviado
Otros servicios de autentificación	Si	Si
Usuarios Activos por mes		Sin costo hasta 50,000 MAU Luego se aplican los precios

		mencionados en [15]
Usuarios Activos por mes: SAML/OIDC	50 por mes	Sin costo hasta 50,000 MAU Luego se aplican los precios mencionados en [15]
Cloud Firestone Datos almacenados	1 GiB en total	Sin costo hasta 1 GiB en total Luego se aplican los precios mencionados en [15]
Salida de red	10 GiB por mes	Sin costo hasta 10 GiB por mes Luego se aplican los precios mencionados en [15]
Operaciones de escritura de documentos	20,000 operaciones de escritura por día	Sin costo hasta 20,000 operaciones de escritura por día Luego se aplican los precios mencionados en [15]
Operaciones de lectura de documentos	50,000 operaciones de lectura por día	Sin costo hasta 50,000 operaciones de lectura por día Luego se aplican los precios mencionados en [15]
Eliminaciones de documentos	20,000 eliminaciones por día	Sin costo hasta 20,000 eliminaciones por día Luego se aplican los precios mencionados en [15]
Cloud Messaging (FCM)	Sin costo	
Hosting Almacenamiento	10GB	$0.026 por GB
Transferencia de datos	360MB por día	$0.15 GB por día
Dominio personalizado y SSL	Si	Si
Varios sitios por proyecto	Si	Si
Crashlytics	Sin costo	
Performance Monintoring	Sin costo	
Realtime Database		

Conexiones simultaneas	100	200,000 por base de datos
GB almacenados	1GB	$5 Por GB
GB descargados	10 GB por mes	$1 por GB
Varias bases de datos por proyecto	No	Si
Remote config	Sin Costo	
Google Cloud BigQuery	Si	Si
Otras IaaS	No	Si

Tabla 3. Comparación de versión gratuita y de pago en Firebase

En *Firebase* se encuentran 2 diferentes bases de datos que nos permiten almacenar los datos, las cuales son *Cloud Firestone* y *Realtime Database*. A continuación, se verán algunas características de cada una de ellas.

- Cloud Firestone

Es la base de datos más reciente de *Firebase* para el desarrollo de apps para dispositivos móviles. Aprovechó lo mejor de *Realtime Database* con un modelo de datos nuevo y más intuitivo. También se pueden realizar consultas más ricas y rápidas [13].

Se trata de una base de datos NoSQL alojada en la nube, flexible y escalable. Mantiene los datos sincronizados entre apps cliente a través de objetos de escucha en tiempo real. El modelo de datos de *Cloud Firestore* admite estructuras de datos flexibles y jerárquicas. Los documentos donde se almacenan los datos pueden contener objetos anidados complejos, además de subcolecciones [16].

Se pueden usar consultas para recuperar documentos individuales o una colección de documentos. Las consultas pueden incluir varios filtros en cadena y combinar los filtros con criterios de orden. Todos los datos de caché que puede usar una app de forma activa, por lo que se puede escribir, leer, escuchar y consultar datos, aunque el dispositivo se encuentre sin conexión. Cuando el dispositivo vuelve a estar en línea, se sincronizan todos los cambios locales de vuelta a *Cloud Firestore* [16] [13].

- Realtime Database

Es la base de datos original de *Firebase.* Es una solución eficiente de baja latencia destinada a las apps para dispositivos móviles que necesitan estar sincronizados entre los clientes en tiempo real.

Es una base de datos alojada en la nube. Los datos se almacenan en formato JSON y se sincronizan en tiempo real con cada cliente conectado. Una característica de *Realtime Database*, es que en lugar de solicitudes HTTP típicas, usa la sincronización de datos, de forma que cada vez que cambian los datos, los dispositivos conectados reciben esa actualización en milisegundos [16] [13].

Los datos persisten de forma local, incluso cuando no hay conexión, se siguen activando los eventos en tiempo real, lo que proporciona una experiencia adaptable al usuario final. Cuando el dispositivo vuelve a conectarse, se sincronizan los cambios de los datos locales con las actualizaciones remotas que ocurrieron mientras el cliente estuvo sin conexión [13].

2.8 Android Studio

Para el desarrollo de apps Android la mejor opción que se encuentra disponible y la oficial es *Android Studio* que es el entorno de desarrollo integrado (IDE) oficial para el desarrollo de apps Android. Basado en el potente editor de código y las herramientas para desarrolladores de *IntelliJ IDEA* [7], ofrece funciones especializadas para compilar apps para Android, como las siguientes:

- Un sistema de compilación flexible basado en *Gradle*
- Un emulador rápido y cargado de funciones
- Un entorno unificado donde puedes desarrollar para todos los dispositivos Android
- Ediciones en vivo para actualizar elementos componibles en emuladores y dispositivos físicos, en tiempo real
- Integración con *GitHub* y plantillas de código para ayudarte a compilar funciones de apps comunes y también importar código de muestra
- Variedad de marcos de trabajo y herramientas de prueba
- Herramientas de Lint para identificar problemas de rendimiento, usabilidad y compatibilidad de versiones, entre otros
- Compatibilidad con C++ y NDK
- Compatibilidad integrada con *Google Cloud Platform*, que facilita la integración con *Google Cloud Messaging* y *App Engine*

El software nos permite crear proyectos y sus interfaces visuales, programarlos, probarlos y subirlos a *Play Store*. Utiliza el lenguaje de programación de *Java* y uno más nuevo que está sustituyendo a este, llamado *Kotlin* [7].

Capítulo 3

Diseño e implementación

En este capítulo se describen los dispositivos y componentes que conforman el hardware del sistema, así como los programas y archivos necesario en la parte de software. Además, de cómo se configuró cada una de ellas, su función y como se acoplaron dentro del sistema. En la figura 16 se muestra el diagrama general del envió de datos del sistema, desde el automóvil hasta que llegan a la aplicación móvil.

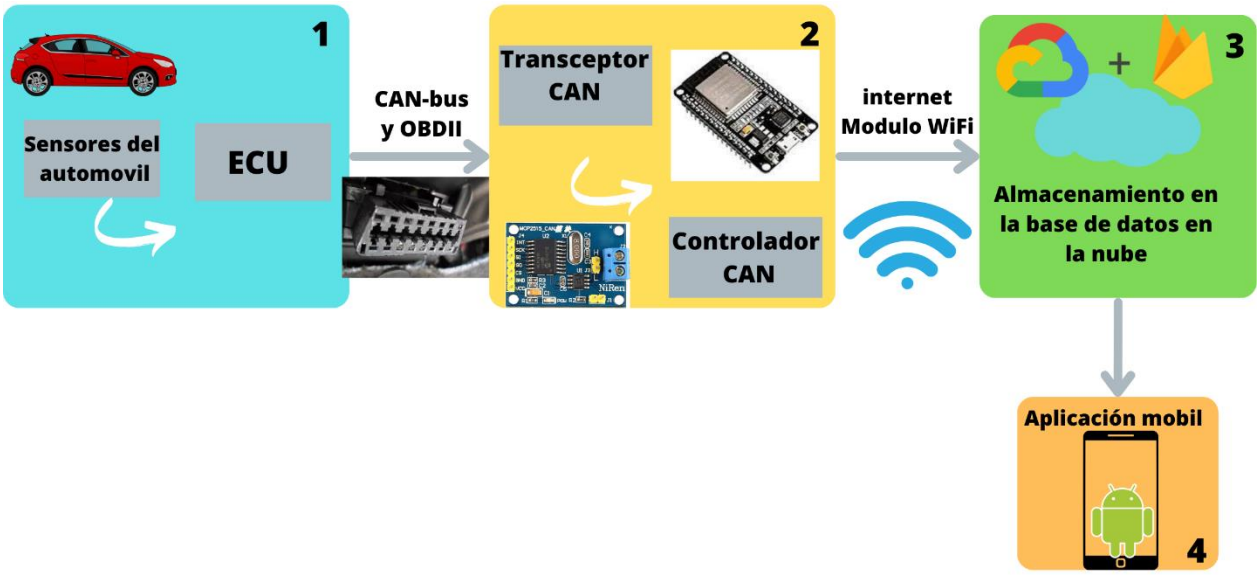

Figura 16. Diagrama de bloques del flujo de datos en el sistema.

En la figura anterior se describe el flujo de datos del sistema. En el primer bloque se encuentra la información que las computadoras (ECUs) reciben de los diferentes sensores del automóvil, luego en el bloque 2 se utiliza un microcontrolador en conjunto con un controlador y transceptor CAN para adquirir la información de las ECUs, comunicándose a través del CACN-bus y el conector OBDII. En el bloque 3 se recibe la información adquirida en una base de datos con servidor en la nube, mediante un módulo wi-fi usado por el microcontrolador. Para finalmente en el último bloque hacer llegar la información y mostrarla en una aplicación móvil del usuario.

3.1 Hardware

El hardware del sistema está conformado por todos los dispositivos y partes físicas usadas para que funcione el sistemas, estos elementos son los siguientes: el microcontrolador, que es la parte central del

sistema, es el que se encarga de realizar el control y procesamiento de la información recibida del CAN-bus, utilizando el controlador y transceptor CAN; el transceptor CAN el cual acondiciona los niveles eléctricos con los que trabaja el CAN, siendo la interfaz entre el microcontrolador y el bus; un conector USB para poder programar el microcontrolador y finalmente un teléfono celular donde se mostrará la información.

3.1.1 ESP32 DevKit V1

Para este proyecto se utilizó un microcontrolador ESP32 DevKit v1, desarrollada por Espressif Systems. Se utilizó este dispositivo debido a su bajo costo y la cantidad de características que posee, aptas para el desarrollo de proyectos de tipo IoT, entre estas se encuentra un controlador CAN [36]. Algunas de las características y especificaciones del este dispositivo son los siguientes:

- Voltaje de Alimentación (USB): 5V DC
- Voltaje de Entradas/Salidas: 3.3V DC
- Consumo de energía de 5μA en modo de suspensión
- Pines Digitales GPIO: 24 (Algunos pines solo como entrada)
- Conversor Analógico Digital: Dos ADC de 12bits tipo SAR, soporta mediciones en hasta 18 canales, algunos pines soportan un amplificador con ganancia programable
- Antena en PCB
- Tipo: Módulo Wifi + Bluetooth
    - Wifi: 802.11 b/g/n/e/i (802.11n @ 2.4 GHz hasta 150 Mbit/s)
    - Bluetooth: 4.2 BR/EDR BLE Modo de control dual
- CPU principal: Tensilica Xtensa 32-bit LX6
- Memoria: 448 KByte ROM, 520 KByte SRAM, 6 KByte SRAM en RTC y QSPI admite múltiples chips flash /SRAM
- Procesador secundario: Permite hacer operaciones básicas en modo de ultra bajo consumo
- Desempeño: Hasta 600 DMIPS.
- Frecuencia de Reloj: hasta 240Mhz
- Seguridad: IEEE 802.11, incluyendo WFA, WPA/WPA2 y WAPI
- Criptografía acelerada por hardware: AES, SHA-2, RSA, criptografía de curva elíptica (ECC), generador de números aleatorios (RNG)

Cuenta con un sistema operativo liviano FreeRTOS, lo que permite que sea multitarea, necesario cuando se deben ejecutar varios subprocesos al mismo tiempo. El programador de RTOS cambia rápidamente entre tareas, lo que hace que parezca que cada tarea se ha estado ejecutando simultáneamente.

En la siguiente figura se muestran los pines y las características específicas de cada uno.

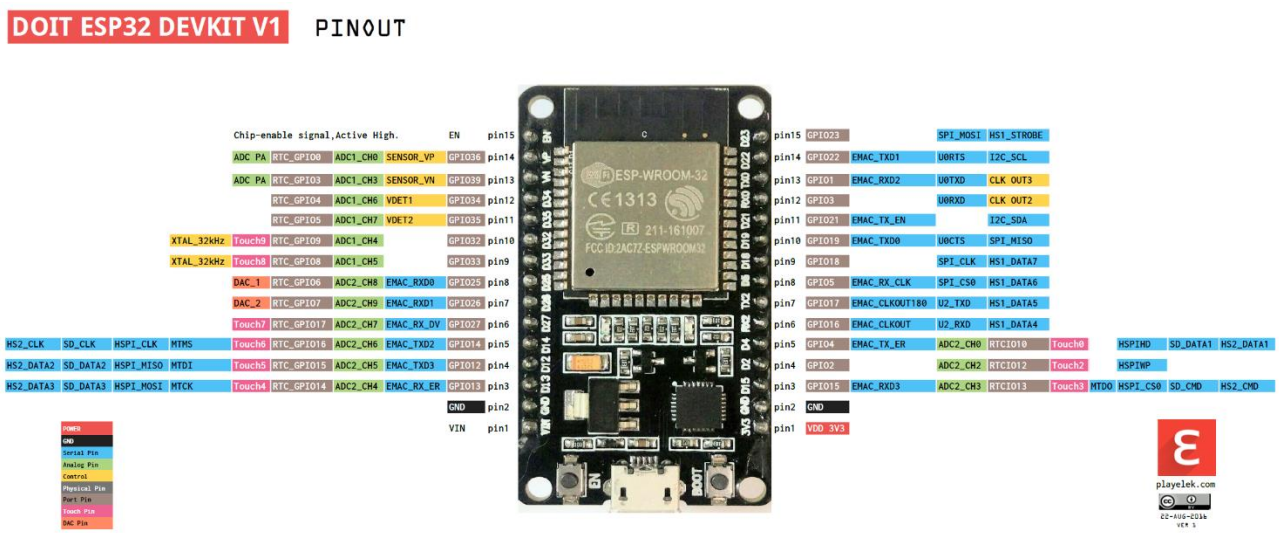

Figura 17. Acciones características de los pines del ESP32 [14].

3.1.1.1 Controlador universal CP210X

Para comenzar a programar la placa se debe instalar un controlador en la PC o laptop. Para este caso, el dispositivo incorpora un chip CP2102, fabricado por Silicon Labs. Los drivers o controladores del puerto virtual COM (VCP) del puente USB a UART en los CP210x, son necesarios para el funcionamiento del dispositivo como un VCP para facilitar la comunicación del host con los productos CP210x. Estos dispositivos también pueden interactuar con un host mediante el controlador de acceso directo [29]. En la página de SiliconLabs se pueden descargar los drivers, específicamente en el siguiente link: https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers [29].

En la figura 18 se muestra señalado por un círculo rojo el chip CP210x.

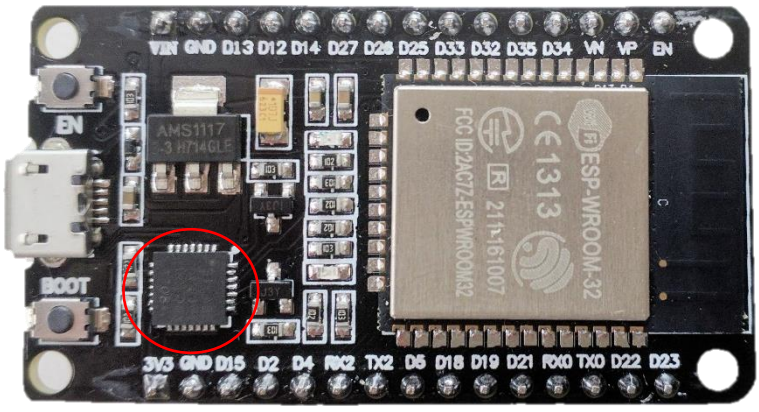

Figura 18. Señalización del controlador CP210 en el ESP32.

Una vez instalado el controlador, se comprueba que se halla hecho la instalación correctamente y que se ha asignado un puerto a la placa. Para esto lo primero es abrir el "administrador de dispositivos" del

equipo, luego se conecta la placa y se abre el apartado de puertos (COM y LPT). Aquí se debe encontrar el dispositivo CP210X con un numero asignado al COM, como se muestra en la figura 19.

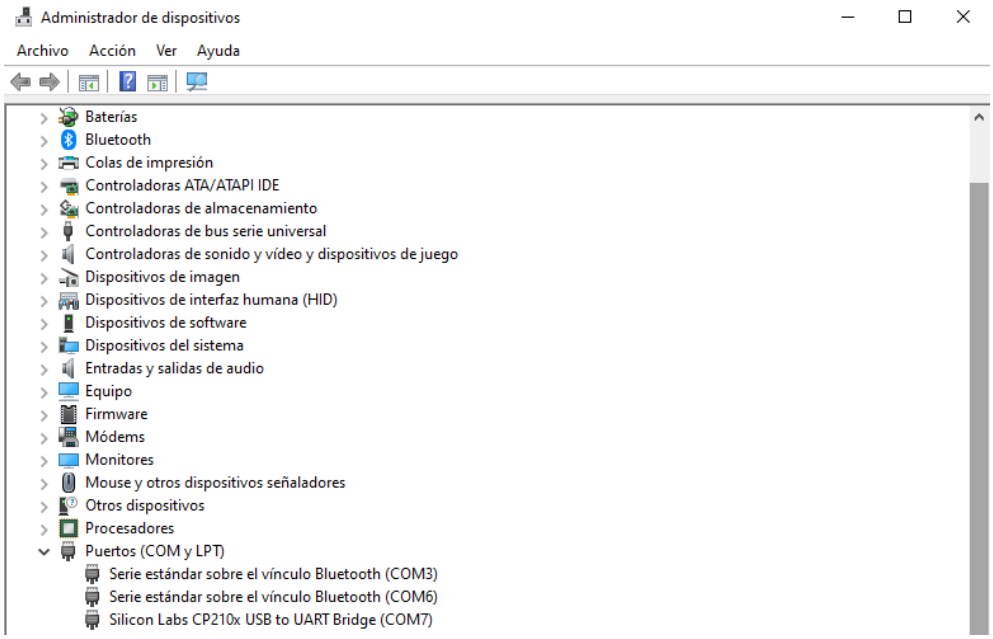

Figura 19. Muestra de que se reconoce el controlador CP210 en el equipo.

Es importante mencionar que si ya se han instalado los controladores con anterioridad ya no es necesario instalarlos nuevamente.

3.1.2 Tarjeta Arduino uno

Las tarjetas de Arduino son de las más ocupadas en proyectos de electrónica, debido a su facilidad de programación usando plataformas de código abierto y a su robustes. La tarjeta se muestra en la figura 20.

Algunas de las características de esta tarjeta son:

- Microcontrolador: ATMega328P.
- Velocidad de reloj: 16 MHz.
- Voltaje de trabajo: 5V.
- Voltaje de entrada: 7,5 a 12 voltios.
- Pinout: 14 pines digitales (6 PWM) y 6 pines analógicos.
- 1 puerto serie por hardware.
- Memoria: 32 KB Flash (0,5 para bootloader), 2KB RAM y 1KB Eeprom

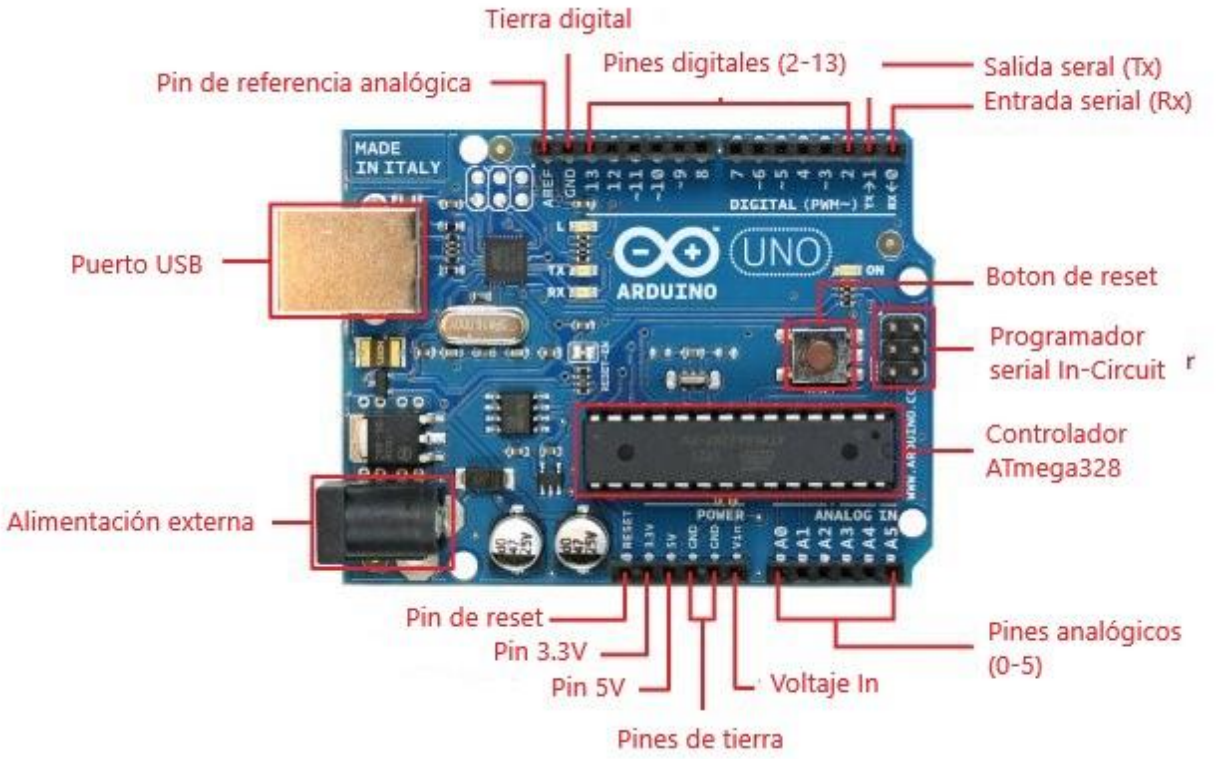

Figura 20. Pines y dispositivos integrados en la placa Arduino UNO.

3.1.3 Transceptor SN65HVD230

El transceptor CAN permite que el controlador CAN del ESP32 acceda a las líneas CAN_H y CAN_L de la red. Por lo tanto, su función básica es adaptar los niveles del bus CAN a niveles compatibles con el controlador CAN para los mensajes recibidos y el caso contrario, para enviados. También es muy importante para proteger el controlador por una posible subida de tensión en el bus (protección ESD) [33].

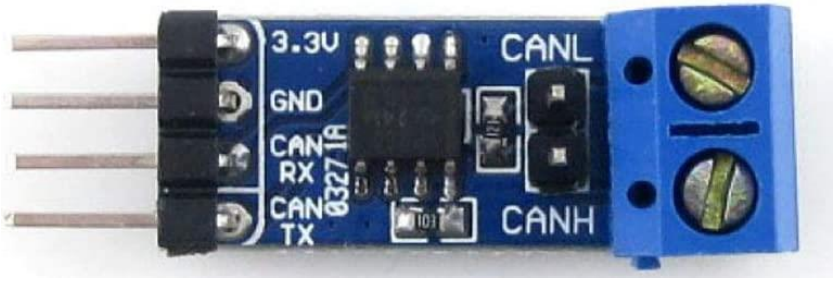

Figura 21. Transceptor CAN SN65HVD230.

El transceptor CAN SN65HVD230 fue desarrollado por Texas Instruments y con el módulo fabricado por Waveshare. En la figura 22 y la tabla 4 se muestra la configuración de los pines con sus funciones. El transceptor funciona con un nivel de tensión de 3.3V, es compatible con el ESP32. Está destinado a utilizarse en aplicaciones que emplean la comunicación serie CAN, que va de acuerdo con la norma ISO

11898 la cual define el protocolo CAN. Puede trabajar en modo de alta velocidad, conectando el pin Rs a tierra, como se muestra en la figura 3, puede permitir hasta 1 Mbps de velocidad de señal [33].

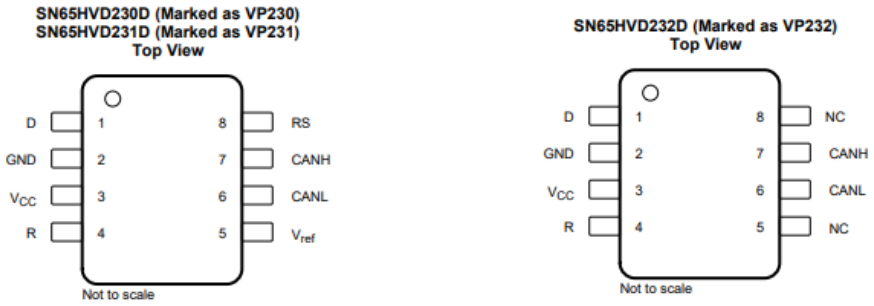

Figura 22. Pines caracteristicos del transceptor SN65HVD230 [33].

PIN		TYPE	DESCRIPTION
NAME	NO.		
D	1	I	CAN transmit data input (LOW for dominant and HIGH for recessive bus states), also called TXD, driver input
GND	2	GND	Ground connection
V_{CC}	3	Supply	Transceiver 3.3V supply voltage
R	4	O	CAN receive data output (LOW for dominant and HIGH for recessive bus states), also called RXD, receiver output
V_{ref}	5	O	SN65HVD230 and SN65HVD231: V_{CC} / 2 reference output pin
NC		NC	SN65HVD232: No Connect
CANL	6	I/O	Low level CAN bus line
CANH	7	I/O	High level CAN bus line
R_S	8	I	SN65HVD230 and SN65HVD231: Mode select pin: strong pull down to GND = high speed mode, strong pull up to V_{CC} = low power mode, 10kΩ to 100kΩ pull down to GND = slope control mode
NC		I	SN65HVD232: No Connect

Tabla 4. Descripcion de los pines del transceptor SN65HVD230 [33].

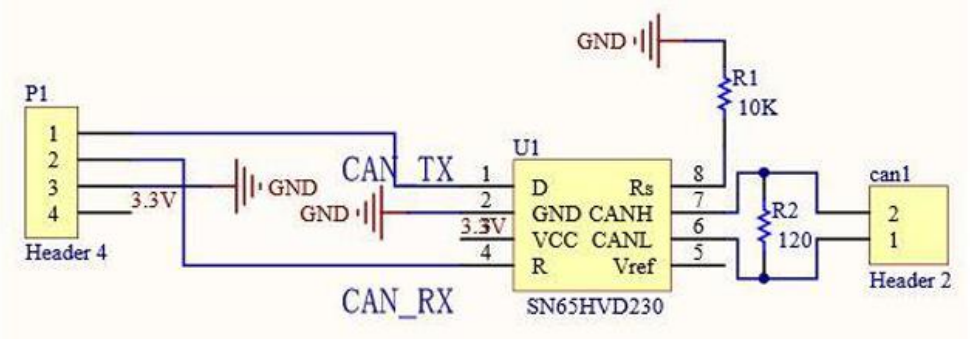

Figura 23. Diagrama eléctrico del transceptor SN65HVD230 [33].

En la figura 23 se muestra el diagrama del SN65HVD230 mencionado anteriormente. El cabezal 4 del esquema está conectado al controlador CAN, siendo los pines 1 y 2 para transmisión y recepción, respectivamente (CAN_TX y CAN_RX). Los pines 4 y 3 alimentan la placa transceptora.

3.1.4 Transceptor y controlador MCP2515

El Módulo MCP2515 CAN Bus TJA1050 es un bus de uso común en la industria, debido a su amplio alcance en distancias, velocidad de comunicación y alta fiabilidad. Incluye el controlador MCP2515 con interfaz SPI y el transceptor TJA1050 con el fin de facilitar la comunicación con microcontroladores y tarjetas de desarrollo como Arduino [21]. Las características se muestran a continuación:

- Completa SPI de alta velocidad (10 MHz)
- Campo de datos de 0-8 bytes
- Corriente de trabajo: Tipico 5mA
- Voltaje de funcionamiento: 2,7 V ~ 5,5 V
- Corriente en espera típica: 1uA (modo de espera)
- Soporta CAN2.0B
- Alta velocidad del transceptor CAN TJA1050: 1Mb/s
- Cristal de 8MHZ
- Temperatura de funcionamiento: – 40 ° C ~ 85 ° C
- Trae indicador LED, indicador de energía

En la figura 24 se muestra el diagrama eléctrico del dispositivo, se puede observar los pines y las conexiones entre el transceptor y del controlador, así como las salidas por donde se harán las conexiones a los microcontroladores. El módulo físico se muestra en la figura 25.

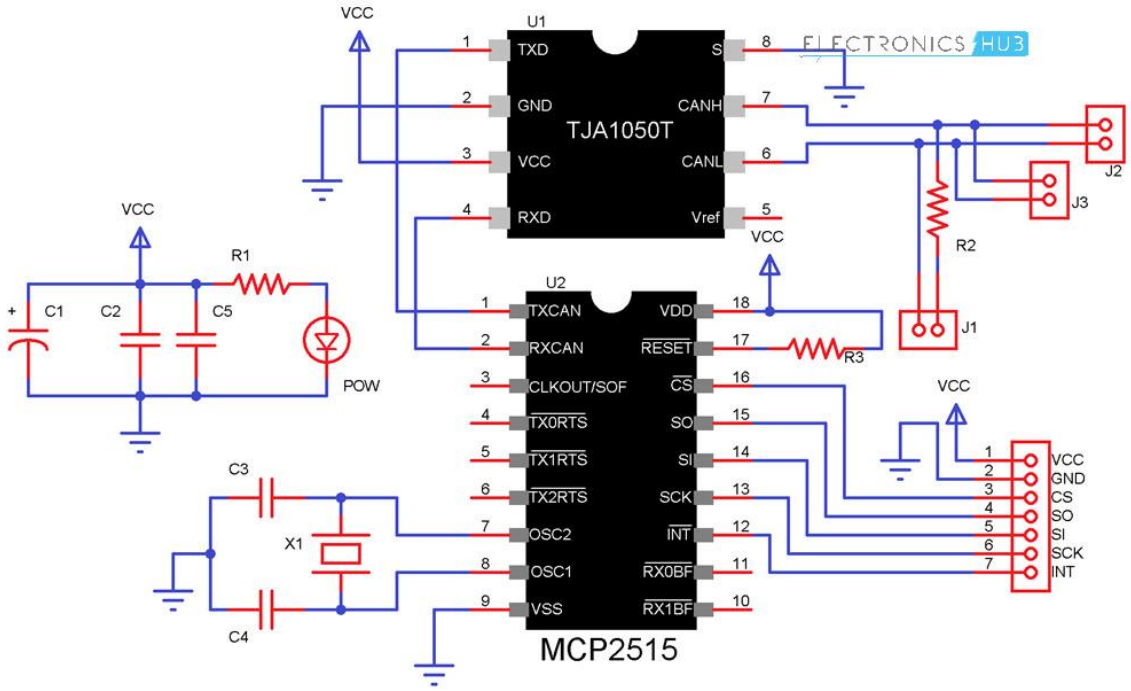

Figura 24. Diagrama eléctrico del módulo MCP2515 [10].

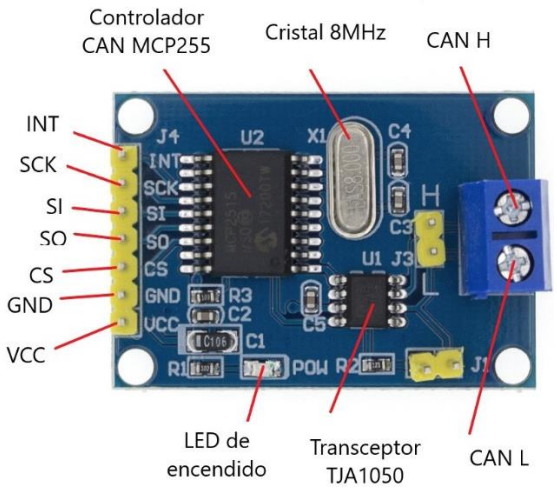

Figura 25. Pines y partes del módulo MCP2515.

3.2 Software

En esta sección se explica la manera de obtener las librerías necesarias para programar el controlador ESP32 y Arduino, así como los controladores y el entorno necesario para programar en la placa, también se describirán algunos aspectos del código implementado. Finalmente, se verá el entrono para crear una base de datos en *Firebase* y el software de *Android Studio*.

3.2.1 Algoritmo general del sistema

A continuación, se muestra el diagrama de flujo del proceso con el que se diseñará la parte del software del sistema.

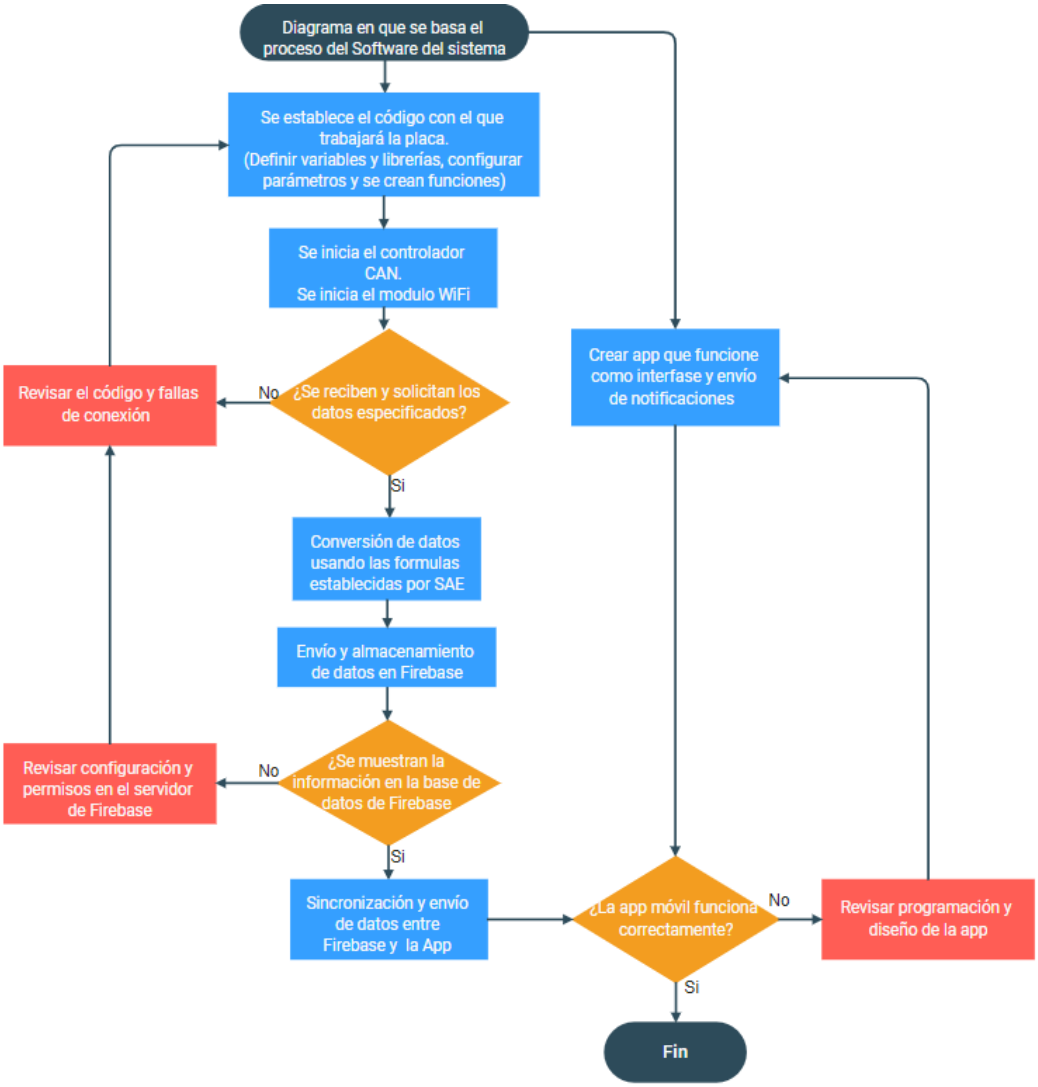

Figura 26. Diagrama del flujo del proceso a realizar por el software del sistema

El proceso inicia estableciendo el entorno de trabajo (librerías y espacios en la nube), para así poder configurar el controlador CAN y empezar a recibir mensajes, de lo contrario se revisa nuevamente el código. Una vez que se recibieron los mensajes, se ocupan las formulas proporcionadas por la SAE para obtener el valor solicitado en decimal y luego se envía a una base de datos en *Firebase.* A la par, se realiza una aplicación móvil que se sincronizará con *Firebase*. Para finalmente, comprobar que la aplicación funciona correctamente y en caso contrario revisar la programación y configuración.

3.2.2 ESP IDF

La compañía de *Espressif* cuenta con un sitio web en el que se encuentra una guía para poder iniciarse en el entorno de programación de las placas ESP a través de ESP-IDF (Espressif IoT Development

Framework). En esta guía se encuentra un menú con diferentes acciones disponibles, y la primera acción que aparece es la de “Star” (Empezar). Aquí se encuentra una pequeña introducción, que se necesita para empezar, guías, herramientas de configuración, como iniciar un proyecto, conectar, configurar, etc [11].

Aquí se nos dice que para desarrollar una aplicación en ESP32 se necesitan los siguientes elementos:

- Una PC cargada con sistema operativo de Windows, Linux o Mac.
- Cadena de herramientas para construir la aplicación para ESP32
- ESP-IDF que esencialmente contiene API para ESP32 y scripts para operar la cadena de herramientas
- Un editor de texto para escribir programas (Proyectos) en C, por ejemplo, Eclipse. En este trabajo se hará uso de IDE de Arduino.
- La placa ESP32 y un cable USB para conectarlo al PC

En la figura 27 se muestra un diagrama del proceso para desarrollar aplicaciones para ESP32.

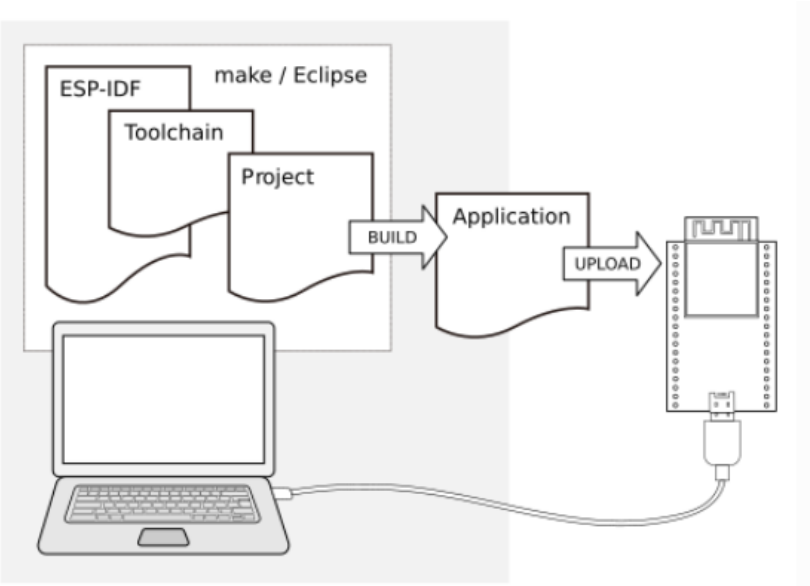

Figura 27. Diagrama del proceso para desarrollar aplicaciones con el ESP32 [11]

La preparación del entorno de desarrollo se divide en 3 pasos:

1. Configuración de la cadena de herramientas (toolchain)
2. Obtener ESP-IDF desde GitHub
3. Instalación y configuración del editor de textos

En esta misma guía se encuentra un apartado para descargar la cadena de herramientas dependiendo del sistema operativo en el que se valla a trabajar y obtener el ESP-IDF. Sin embargo, como en este trabajo se ocupará el IDE de Arduino estos pasos no se seguirán. Una vez que se configuró el entorno ya se puede iniciar el desarrollo de la aplicación. Espresiff resume este proceso en cuatro pasos [11] :

1. Configuración del proyecto y escritura del código
2. Compilación del proyecto y vinculación para construir la aplicación
3. Flasheo (carga) de la aplicación a la placa ESP32
4. Monitoreo/depuración de la aplicación

3.2.3 Arduino IDE

El entorno de programación de Arduino es de los más populares debido a su accesibilidad, la comunidad que a menudo comparte sus proyectos y aporta ideas en el foro propio de Arduino o alguno otro y debido a la facilidad de programar en lenguaje C++. Por eso y por la posibilidad de poder trabajar con las tarjetas de Arduino y ESP32 fue que se trabajó usando este entrono de programación.

3.2.4 Agregar tarjetas de Espressif a Arduino

Antes de usa el Arduino IDE en conjunto con la placa ESP32, primero se debe instalar un conjunto de herramientas (toolchain) específico del ESP32 que se integra con facilidad al entorno, sobre todo a partir de su versión 1.6.4 [34].

La instalación se inicia abriendo el Arduino IDE y desde el menú "Archivo" – "Preferencias". Dentro de esta ventana, debemos buscar "Gestor de URLs adicionales de tarjetas" y allí copiar la siguiente dirección: https://dl.espressif.com/dl/package_esp32_index.json.

En la figura 28 se muestra la ventana con la dirección añadida. Si ya tenemos otra dirección cargada, la agregamos separándola de la anterior con una coma.

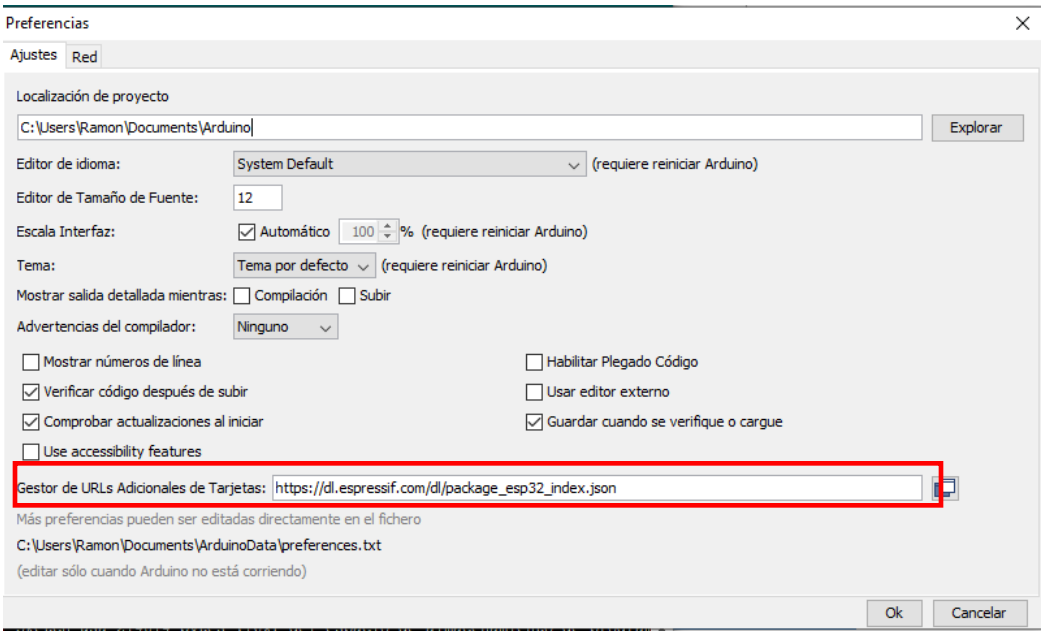

Figura 28. Ventana de "preferencias" Arduino donde se muestra el aparatado para agregar la URL de otra placa (ESP32)

Posteriormente nos dirigimos a el gestor de tarjetas dentro del menú "Herramientas" – "Placa" – "Gestor de tarjetas". Esperamos un momento mientas cargan las tarjetas disponibles, luego buscamos ESP32 y

pulsamos sobre “Instalar”. Comenzará la descarga y la instalación. En la figura 29 se muestra la tarjeta instalada.

Figura 29. Gestor de librerías de Arduino con el paquete de ESP32 instalado

Una vez finalizada la instalación, ya es posible trabajar con la ESP32 en el IDE de Arduino. Se puede observar que se añadieron diferentes placas fabricadas por *Espressif*, estás se encuentran en el menú “Herramientas” – “Placa”. Para empezar a programar la placa, seleccionamos “ESP32 Dev Module” o “DOIT ESP32 Devkit V1”, y el puerto (COM7 en mi caso). Como se observa en la figura 30.

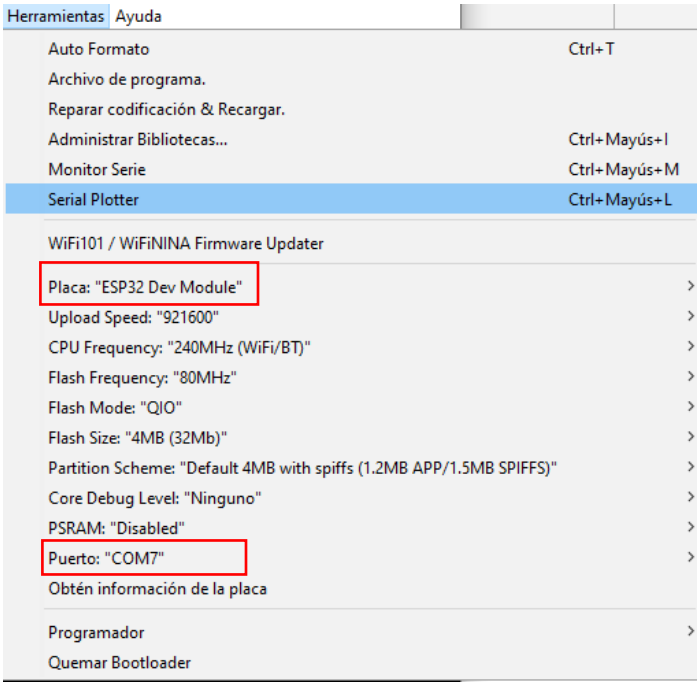

Figura 30. Ventana de “herramientas” de Arduino donde se muestra que se agregarn correctamente las librerías para la placa ESP32

Ya configurado el entorno de Arduino, se ejecutó el código “Blink”, que se encuentra en los ejemplos de la placa de Arduino. El código hace parpadear el led que viene incorporado en la placa y que se encuentra asignado al pin 2 del ESP32, con el propósito de comprobar que la placa se pueda pogramar correctamente con el Arduino IDE. El código se compiló y se subió correctamente a la placa. El código se muestra en la figura 31.

Figura 31. Código "Blink" usado para probar el ESP32

Se realizó una segunda prueba para comprobar si el módulo wi-fi incorporado en el ESP32 funciona y no esta averiado. Para lo cual se probó el código "WifiScan", que se encuentra en el apartado de "WiFi" en los ejemplos de la placa ESP32 DEVKID V1.

3.2.5 Librerías para el controlador CAN de Arduino IDE

Para trabajar con el modelo de comunicación CAN bus en la plataforma de Arduino, se tuvieron que instalar diferentes librerías para probar en las dos placas. Las librerías se descargaron en formato zip, directamente de la página de *Github*, en los siguientes links: [6][8][19][22][39].

Para añadir las librerías o ficheros se debe ir al menú "Programa" – "Incluir librería" –"Añadir biblioteca .ZIP". Como se muestra en la figura 32.

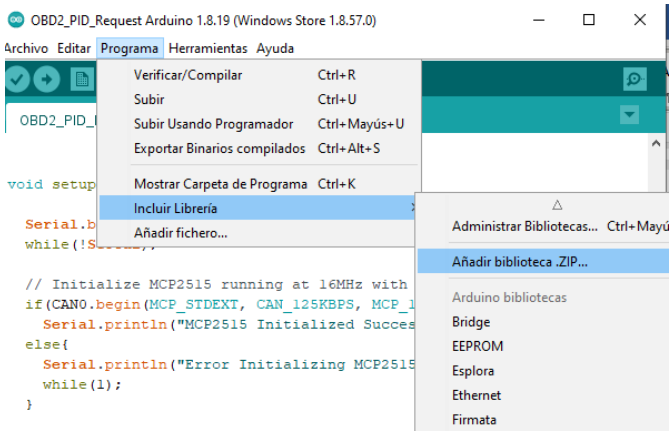

Figura 32. Apartado para añadir las librerías CAN a Arduino.

Las librerías que se agregaron para trabajar con el CAN-bus y posteriormente conectarse al OBD2 se muestran marcadas en la figura 33.

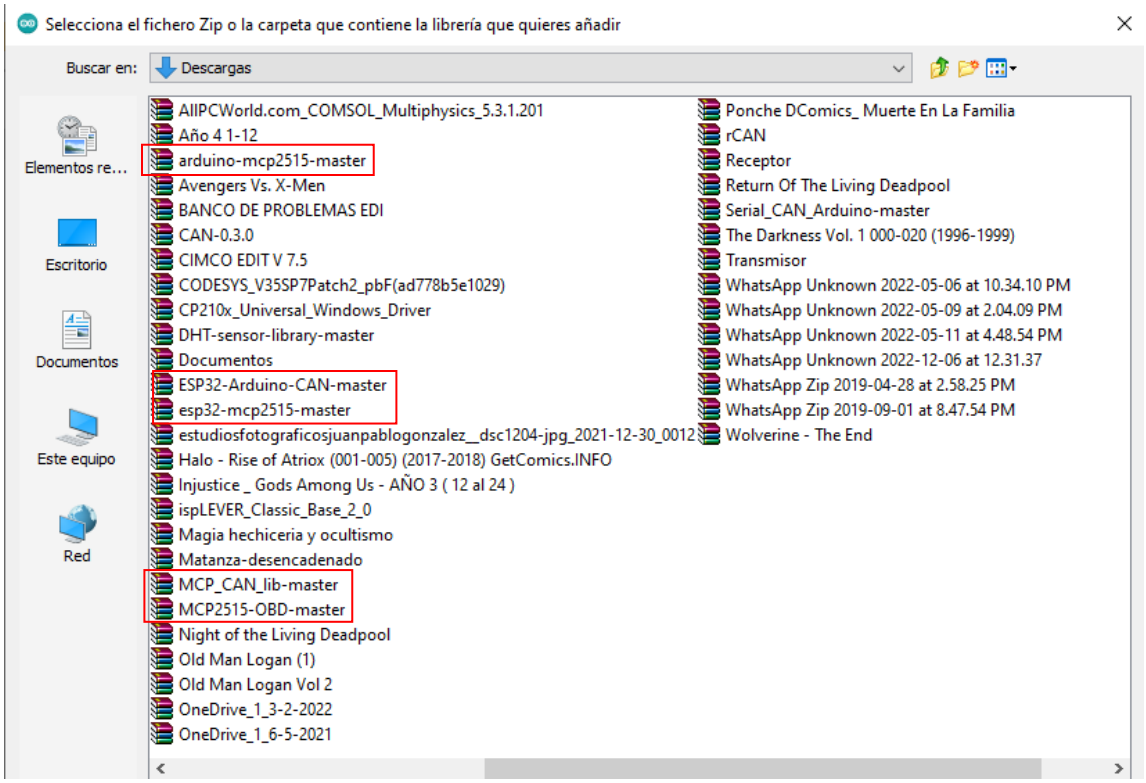

Figura 33. Librerías CAN que se instalaron.

3.2.6 Base de datos (Realtime Database) en Firebase

Como se mencionó en el capítulo 2, existen diferentes servidores que nos permiten trabajar con bases de datos. La principal diferencia entre *Firebase* y otros servidores es que esta se especializa en desarrollar aplicaciones y no en administrar bases de datos, por lo que se decidió utilizar esta base de datos pues se acopla mejor a los objetivos del proyecto, que es el almacenamiento de datos para futuras aplicaciones en IoV, debido a la fácil accesibilidad, comentarios de ayuda que se ofrecen y que se puede ocupar con diferentes dispositivos, como es el caso del ESP32. Además de que es el gestor de base de datos por defecto en *Google Cloud*, que nos ofrece varias herramientas de forma gratuita para probar y sacar aplicaciones y solo se necesita un correo electrónico de *Google* para poder trabajar ahí.

Para trabajar con *Firebase* lo primero que se debe hacer es iniciar sesión con una cuenta de *Google* en la página de *Firebase*, la cual se abre automáticamente si tu dispositivo esta sincronizado con *Google*, posteriormente dirigirse al apartado de consola donde se nos muestra la opción de abrir o crear un nuevo proyecto. Figura 34.

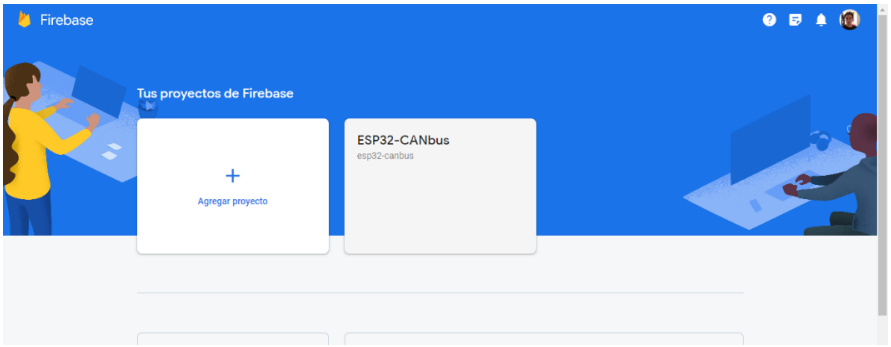

Figura 34. Proyecto ya creado y opción para realizar uno nuevo en Consola de Firebase

Para crear el proyecto simplemente se debe asignar un nombre y activar o desactivar las analíticas, para este proyecto se dejaron desactivadas, sin embargo, si se desea profundizar en el desarrollo de la aplicación es conveniente activarlas. Después de esto se mostrará la siguiente ventana, figura 35. Donde se deberán realizar unas configuraciones para poder ocupar el ESP32 junto con la base de datos. Aquí nos dirigimos al apartado de "authentication", estando ahí seleccionamos "sing-in metod" y nos mostrará lo siguiente, figura 36 y aquí seleccionamos anónimo. Luego únicamente se habilita el anónimo y se guarda.

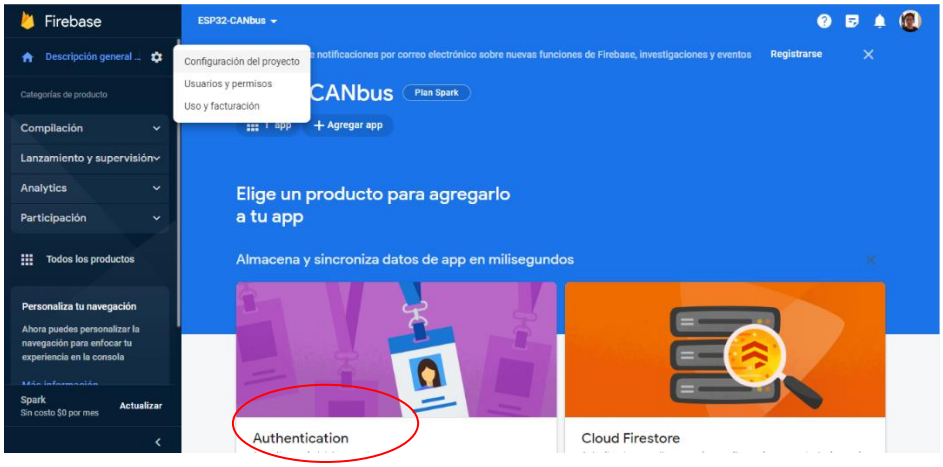

Figura 35. Pantalla principal de Firebase donde se muestra la acción de autentificar.

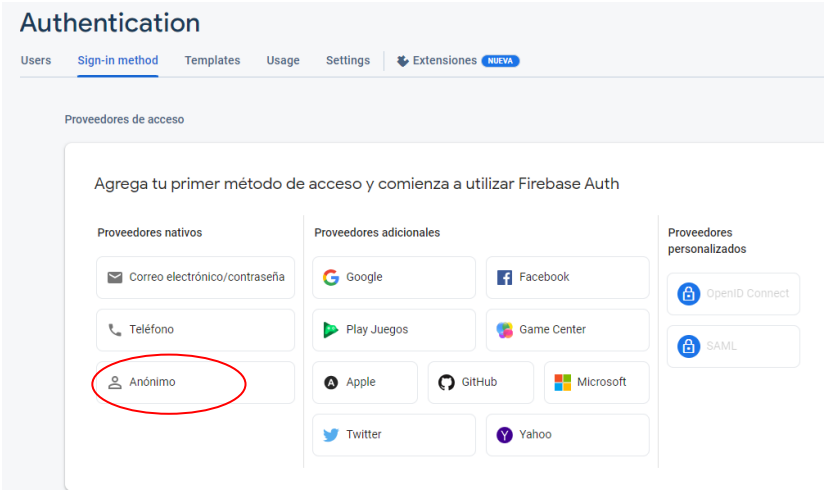

Figura 36. Métodos que se ofrecen para iniciar, seleccionando Anónimo

Luego nos dirigimos al apartado de "realtime database" en la pantalla principal, para crear la base de datos, que consta solo de dos pasos, el primero en donde seleccionamos la ubicación y se nos muestran 3 posibles opciones: Asia, Europa y América(Estados Unidos). El segundo paso nos da a escoger las reglas de seguridad, aquí se usó el modo de prueba, como se muestra en la figura 37 que nos permitirá recibir y enviar datos entre el ESP32 y a la base de datos.

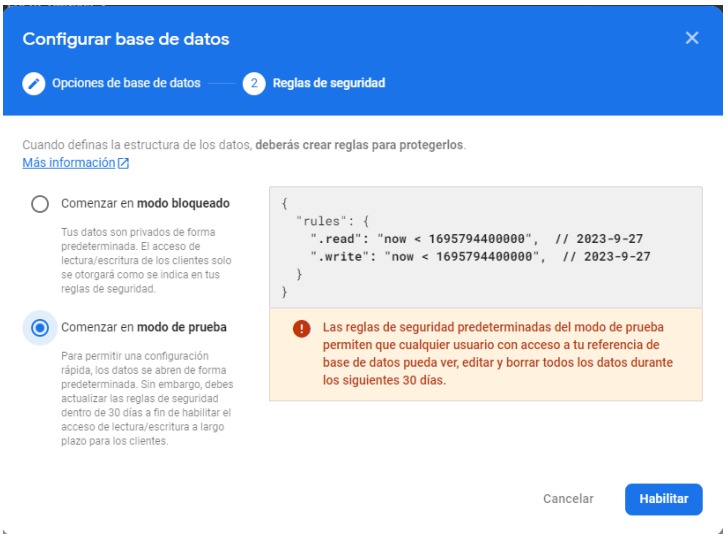

Figura 37. Opciones de reglas de seguridad disponibles para la base de datos.

Una vez creada la base de datos se muestra la pantalla de "Realtime Database", en el apartado de "datos" se muestra el link o URL de la base de datos, como se muestra en la figura 38. Este dato se debe copiar pues es de utilidad para la creación de a base de datos con el ESP32.

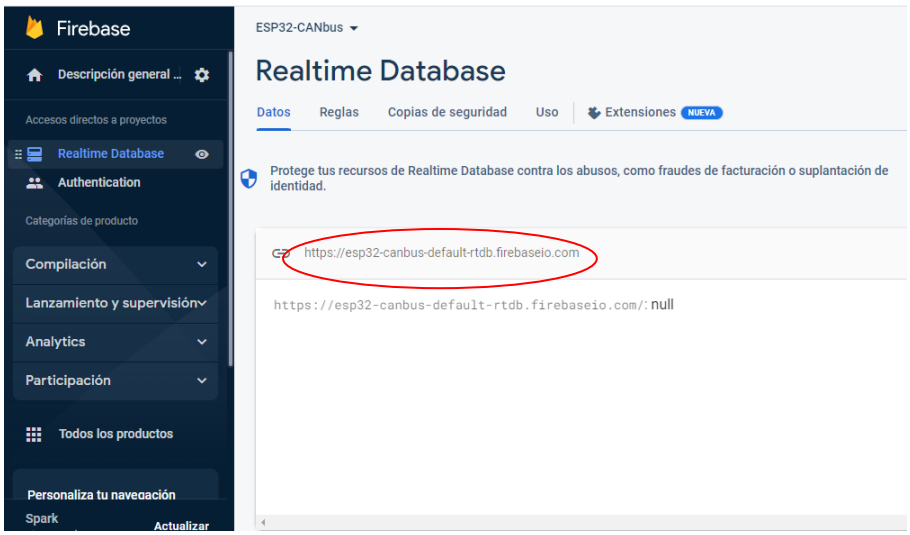

Figura 38. Base de datos y URL de la misma

Luego ahí mismo en la sección de "Realtime Database" nos dirigimos la opción de "Reglas", aquí aparece un pequeño código que genera las reglas que se debe cumplir para poder enviar y recibir información a la base de datos, por lo que se hacen los siguientes ajustes a las reglas como se observa en la figura 39. Y se habilita.

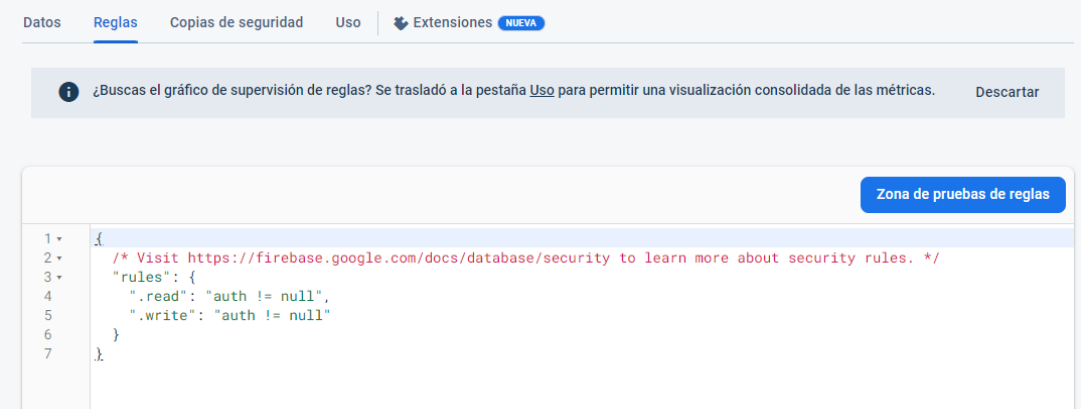

Figura 39. Reglas de la base de datos.

Después, nuevamente en la sección de "authentication" y "sing-in metod" seleccionamos el recuadro de "agregar proveedor nuevo", luego la opción de "correo electrónico/contraseña" y habilitamos y guardamos. Figura 40. Posteriormente nos dirigimos a la opción de "Users" y seleccionamos "Agregar usuario", lo que genera un cuadro, en el que se asigna un correo electrónico y una contraseña. Figura 41. En este caso agregué mi correo electrónico de Gmail "ramonigh.15@gmail.com" y la contraseña "ESPBUAP123", esta información se ocupará más a delante en el código para el ESP32.

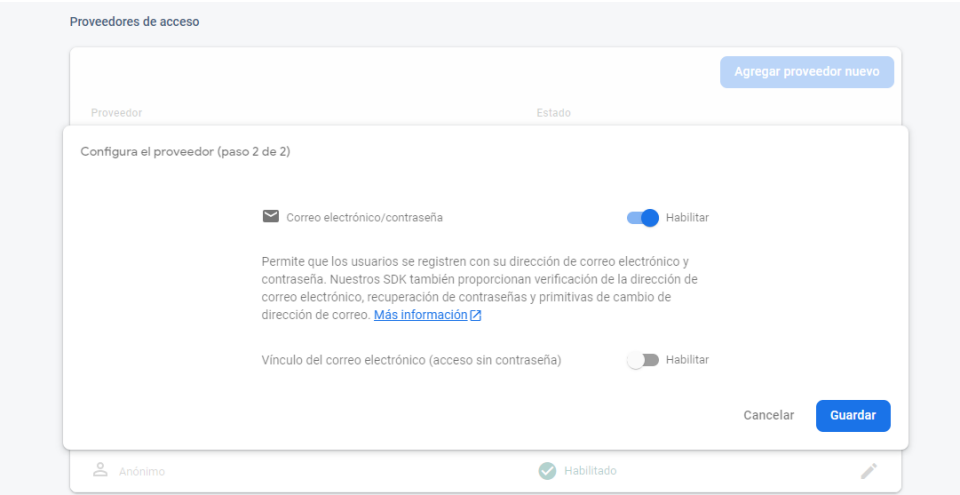

Figura 40. Recuadro para añadir proveedor.

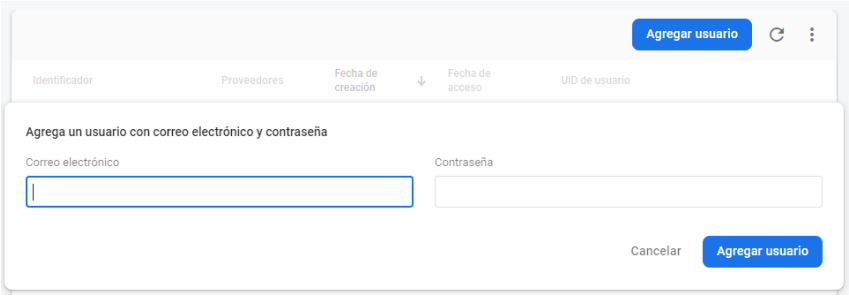

Figura 41. Agregar usuario con correo y contraseña

Antes de dirigirse a Arduino hace falta un último dato necesario para el código, que es la "clave de API web", que nos permite la conexión entre el módulo wifi del ESP32 con el proyecto de Firebase. Esta se encuentra en el aparatado de "configuración de proyecto", en la figura 42 se muestra la API Key del proyecto.

Figura 42. Apartado de Configuración de proyecto donde se muestra la Clave API web

Para trabajar el ESP32 con *Firebase* existe ya una librería en Arduino que nos permite establecer una comunicación entre ambos para enviar y leer información de la base de datos. Así que nuevamente nos

dirigimos a Arduino y al apartado de “Programa”-“incluir librería”-“administrador de bibliotecas”. Esto abre una nueva ventana en donde se muestran todas las librerías, aquí buscamos la de “Firebase ESP32 Client” e instalamos. Figura 43. Ahora ya es posible trabajar en *Firebase* con el ESP32, por lo que se hará una prueba para corroborar que funcione correctamente en el siguiente capitulo.

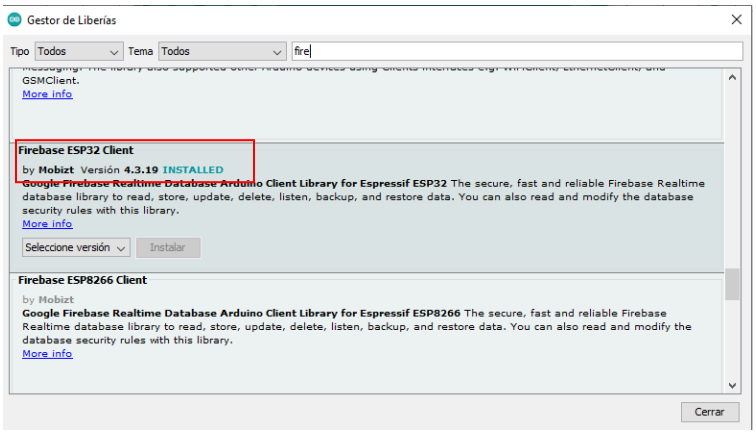

Figura 43. Librería para trabajar con el ESP32 y Firebase ya instalada

3.2.7 Diseño de aplicación móvil en Android Estudio

Como se mencionó en el capítulo anterior se ocupará el programa de *Android Studio* para el diseño de la aplicación móvil, debido a las diferentes herramientas que se ofrecen para desarrollar aplicaciones, con la facilidad de posteriormente subirla a la plataforma de *Play Store*. Algunas herramientas con las que cuenta son: el emulador de diferentes dispositivos móviles que utilizan el sistema *Android*, la interface gráfica que permite ver una pantalla de la aplicación conforme se vaya realizando o incluso realizar la aplicación de manera grafica si no se está familiarizado con la programación, además de que tiene una vinculación directa con servicios de *Google* y *Firebase*.

El programa se encuentra disponible de forma gratuita para *Windows*, junto con una pequeña guía de apoyo en la página de *“*developer.android.com*”*. Una vez abierto el programa se tiene que nombrar el proyecto, así como asignar un “Package” en forma de página web, la locación del proyecto, el lenguaje de programación que puede ser *jaba* o *kotlin* y la versión de Android mínima con la que se puede trabajar, donde se seleccionó la versión “Nougat Android 7.0”, como se muestra en la figura 44, la cual es compatible con aproximadamente 95.4 servidores. En la figura 45 se muestran las diferentes versiones de Android, el porcentaje aproximado de usuarios y una pequeña descripción de la versión. Por razones obvias mientras más reciente es la versión menos usuarios tendrá y como uno de los propósitos del proyecto es que sea accesible se seleccionó la versión antes mencionada.

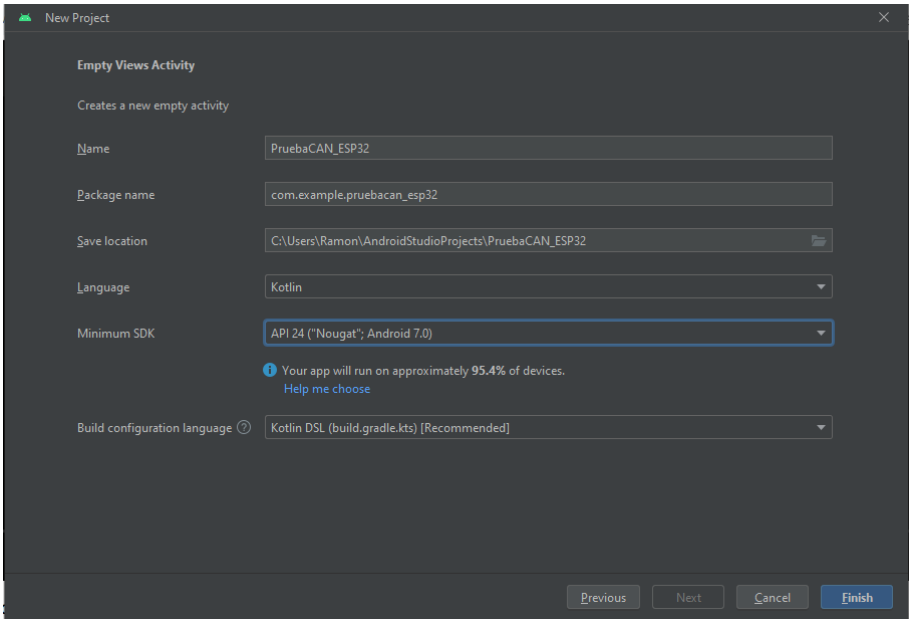

Figura 44. Creación de un proyecto en Android Studio

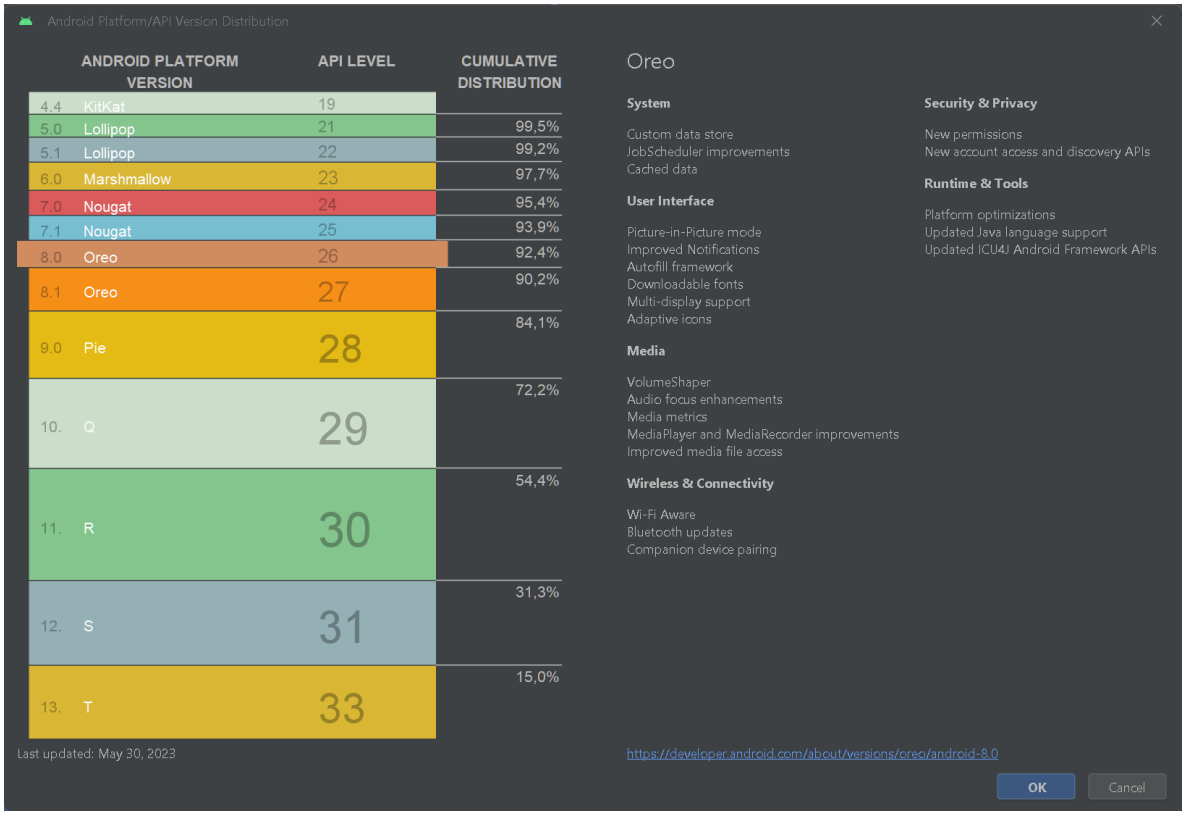

Figura 45. Versiones de Android disponibles

Después de asignar el nombre y las características principales al proyecto, se finaliza y la IDE comienza a crear el nuevo proyecto. En la pantalla principal se muestran los menús y herramientas, en la parte superior y a la izquierda los ficheros del proyecto, en el centro de la pantalla se encuentra el editor y por

encima de este, la pestaña de los archivos abiertos, como se muestra en la figura 45. Aquí se muestra el "Android Manifest.xml" en donde se muestran algunas características y las diferentes actividades con las que cuenta la aplicación, este se encuentra en la carpeta de "manifests" dentro de la carpeta "app".

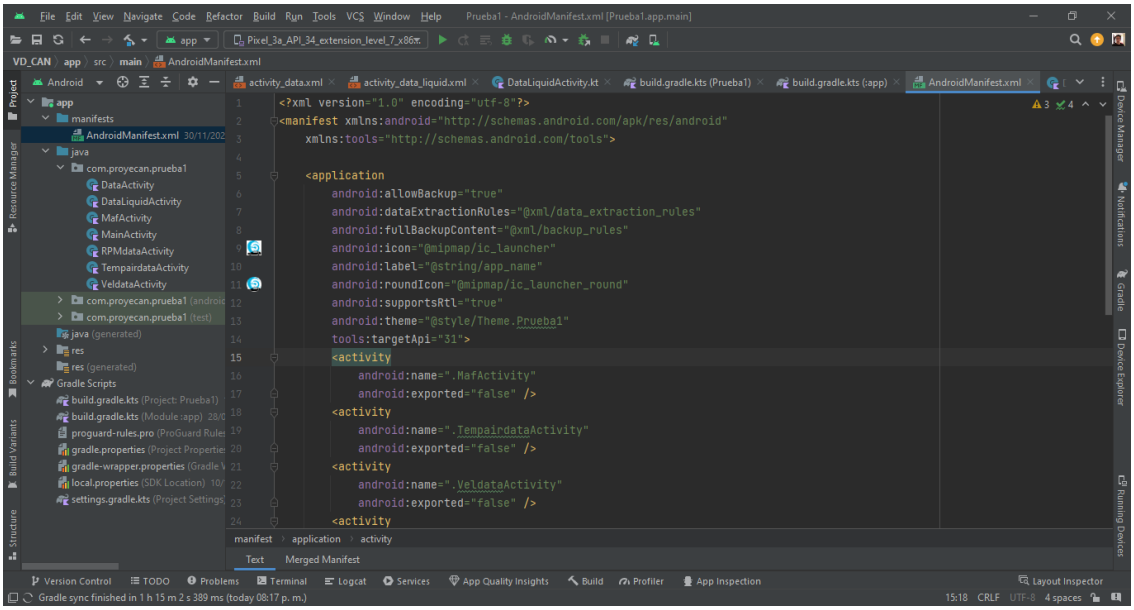

Figura 46. Pantalla de la IDE Android Studio

Una "Activity" funciona en la mayoría de las veces como una pantalla dentro de la aplicación. Por defecto se crea una actividad "MainActivity", que es la que se abrirá cuando se inicie la aplicación, puede estar vacía o tener ya alguna acción para visualizar, dependiendo como se haya elegido. Para crear una nueva actividad se debe dirigirse a la carpeta de "java" y desplegar el menú la carpeta del proyecto, como se muestra en la siguiente figura.

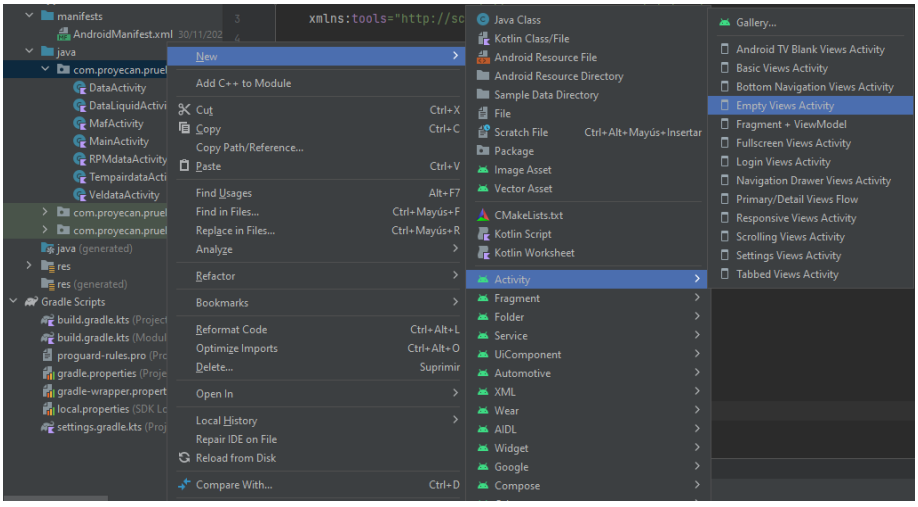

Figura 47. Creación de una nueva actividad

Para el diseño de la interface de la aplicación se creó una actividad para el menú principal, una para la lista de los sensores (PID) y una para mostrar los datos de los diferentes sensores. En la figura 48 se muestran las capturas de la pantalla principal y el listado de los datos de la app desarrollada.

Figura 48. Pantallas capturadas de la aplicación móvil

Se hizo también un sencillo diseño que se mostrará como el icono de la aplicación y para sustituir la imagen que *Android studio* crea por defecto para cualquier aplicación. En la figura siguiente se muestra el diseño de la aplicación y en la figura 50 se muestra la aplicación instalada en el celular con el icono que se creó y nombre asignado VD_CAN “VirtualData_CAN”.

Figura 49. Diseño del icono de la aplicación

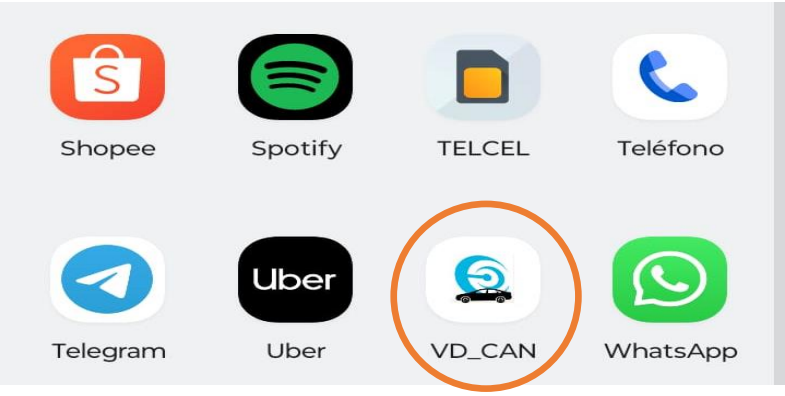

Figura 50. Miniatura de la aplicación instalada

Para poder cambiar el icono de la aplicación existe una forma muy fácil de hacerlo, ya que se necesitan diferentes tamaños y resoluciones para que la densidad del pixel se adapten cada pantalla y cada sistema operativo. Para esto, lo único que se debe hacer es subir la imagen que será el nuevo icono la página con el siguiente link: https://makeappicon.com. Esta web se encarga crear los diferentes tamaños y carpetas asignadas con las que trabaja *Android Studio*. Luego, para asignar el nuevo icono, en *Android Studio* nos dirigimos a la carpeta de "java"-"res"-"mipmap", aquí se encuentran tres carpetas con imágenes del icono por defecto de la app, las cuales tienen el mismo nombre que las carpetas generadas en el sitio web, así que se sustituyen y de esta forma ya la aplicación habrá cambiado su icono.

3.2.7.1 Sincronización del proyecto de Android Studio con Firebase

En la esquina suprior derecha de la pantalla de *Android studio*, al lado del botón de búsqueda se encuentra la opción para iniciar sesión con una cuenta de *Google*. Una vez que se inicia sesión correctamente se muestra lo siguiente:

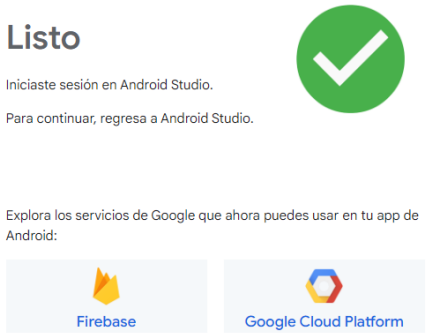

Figura 51. Inicio de sesión en Android Studio con cuenta de Google

Hecho esto, se debe conectar con *Firebase*, lo cual se puede hacer desde la pestaña de "Tools"-"Firebase " en *Android studio*, figura 52, o directamente desde *Firebase* en la opción "agregar app" que se muestra en la figura 53. Aquí se muestran una serie de pasos que se deben seguir para realizar la sincronización, que en resumen consta de agrega el "package" de la app y agregar las dependencias que se brindan al apartado de "build.gradle.tks(app)".

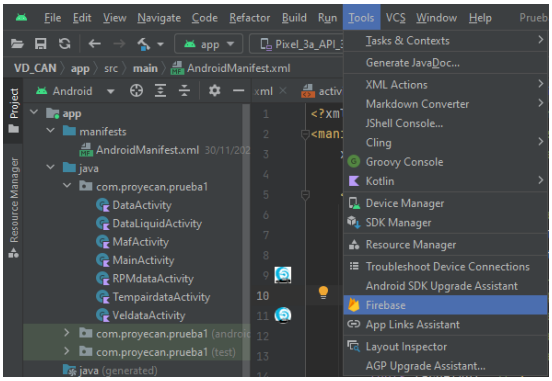

Figura 52. Apartado para incorporar Firebase al proyecto

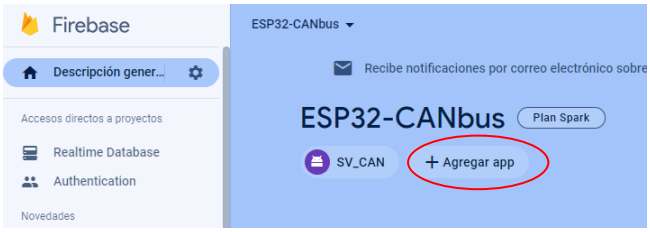

Figura 53. Opción para agregar una app al proyecto

Una vez hecha la conexión con *Firebase* tenemos acceso a las diferentes herramientas que posee, algunas que podeos encontrar se observan en la siguiente figura. Entre las diferentes herramientas se encuentra la conocida *Realtime Database*, la cual se ocupa en este trabajo, al seleccionar esta opción se muestran una serie de puntos, el primero es la sincronización que se realizó anteriormente, por lo que se muestra que ya se ha realizado la conexión, así que en el siguiente punto seleccionamos el botón de "Add the Realtime Database SDK to your app" como se muestra en la figura 54. Esto despliega un cuadro que muestra las líneas que se incorporarán a los ficheros para poder trabajar con "*Realtime Database*", los cuales se muestran en la figura 55 y luego únicamente seleccionamos en "Accept Changes". Hecho esto, ya es posible trabajar con la base de datos "*Realtime Database*".

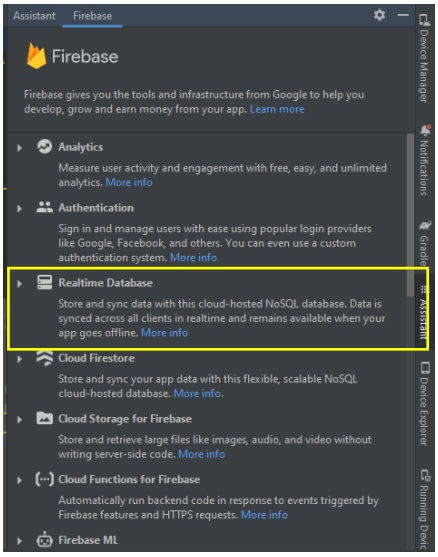

Figura 54. Lista de herramientas proporcionadas por Firebase

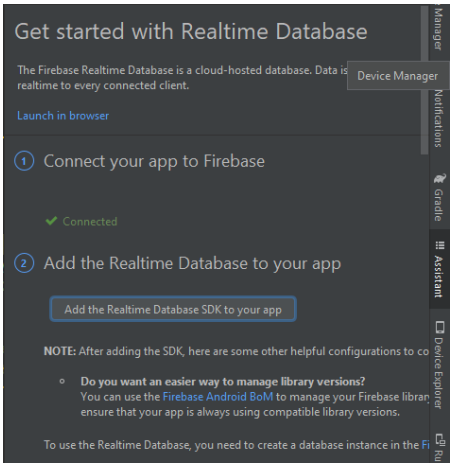

Figura 55. Pasos para iniciar Realtime Database en Android Studio

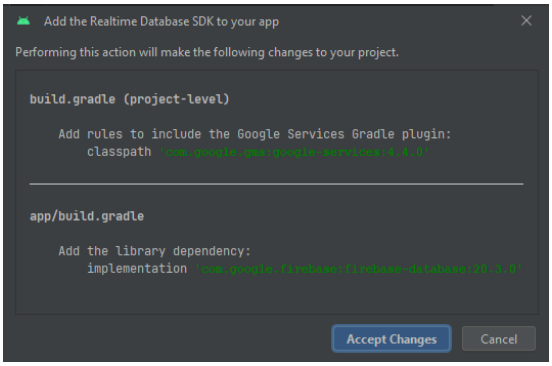

Figura 56. Cambios para agregar el SDK de Realtime Database a la app

3.3 Análisis de las librerías CAN-bus

En este capítulo se mostraron diferentes dispositivos y librerías con los que se puede trabajar una comunicación CAN-bus, por lo que, se decidió examinar la estructura del código de las diferentes librerías, así como las conexiones necesarias entre los dispositivos. Para esto, se diseñó una red de comunicación (receptor y transmisor) CAN usando las placas ESP32 y Arduino. Una vez que se instalaron las librerías necesarias se realizó la programación de las placas. Las redes CAN que se realizaron fueron las siguientes:

- Transmisor y receptor con las placas ESP32 y Arduino, usando la librería de Serial_CAN_Arduino-master.
- Transmisor y receptor de datos con placas Arduino, usando las librerías de Arduino-mcp2515-master y mcp_CAN-lib-master.
- Transmisor y receptor de datos con ESP32, usando las librerías de ESP32-Arduino-CAN-master.
- Transmisor y receptor de datos con ESP32, usando la librería de Esp32-mcp2515-master.

Transmisor y receptor de datos con placas ESP32 y Arduino, usando la librería de CAN.h.

La librería de "CAN.h" se subió a *Github* por el usuario sandeepmistry [22], según la descripción de la librería, esta es compatible con el dispositivo MCP2515 y con el controlador incorporado por el ESP32 en conjunto con el transceptor SN65HVD230. La configuración de los pines de especifica en las siguientes tablas, para el MCP2515 y el ESP32 respectivamente. Esta asignación de pines se usa también con las otras librerías.

Microchip MCP2515	**Arduino**
VCC	5V
GND	GND
SCK	SCK 13
SO	MISO 12
SI	MOSI 11
CS	10
INT	2

Tabla 5. Relación de pines entre el CP2515 y Arduino

Transceptor SN65HVD230	**ESP32**
3V3	3V3
GND	GND
CTX	GPIO_5
CRX	GPIO_4

Tabla 6. Relación de pines entre el SN65HVD230 y el ESP32

Los pines de "CS" e "INT" se pueden cambiar usando *CAN.setPins(cs, irq)* [22]. De igual manera los pines "CTX" y "CRX" usando *CAN.setPins(rx, tx)*. En las siguientes figuras se muestra la conexión entre Arduino y el MCP2515 en físico y en esquemático.

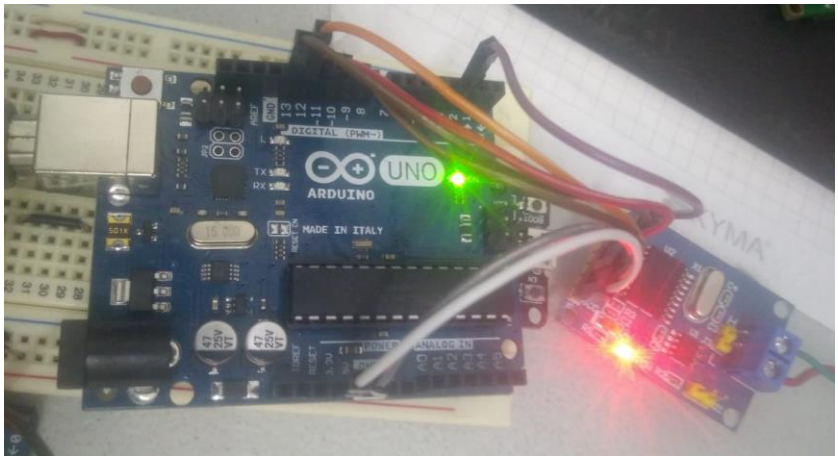

Figura 57. Conexión entre Arduino y el MCP2515

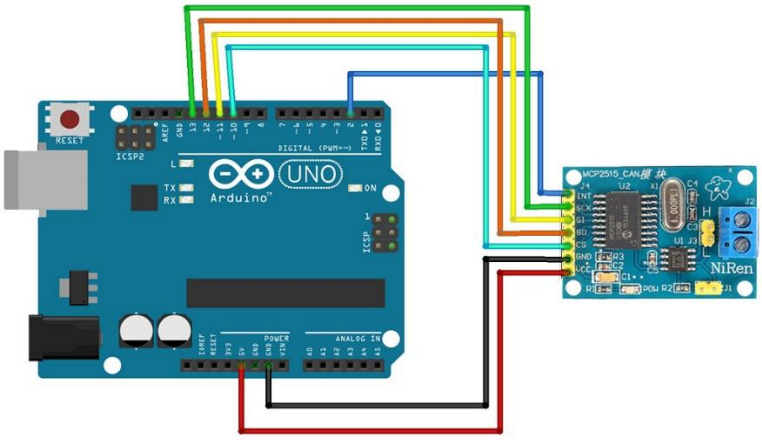

Figura 58. Esquema de la conexión Arduino y el MCP2515

El paquete de la librería cuenta con unos ejemplos para enviar y recibir mensajes, los cuales son los que se probaron con las placas. En ambos nodos lo primero que se debe realizar es iniciar el CAN a una velocidad específica, que en este caso son 500kB por segundo, utilizando la función *CAN.begin()* como se muestra en la figura 59.

```
  // start the CAN bus at 500 kbps
  if (!CAN.begin(500E3)) {
    Serial.println("Starting CAN failed!");
    while (1);
  }
}
```

Figura 59. Inicio del CAN a 500kbps usando CAN.h

El transmisor agrega un identificador estándar con la función *CAN.beginPacket()* y con *CAN.write()* se agrega un argumento a la trama, donde según el protocolo pueden ser hasta 8 bytes, con *CAN.beginExtendedPacket()* se agrega un identificador extendido y luego envía una nota de que se envió la trama, como se muestra en la siguiente figura.

```
CAN.beginPacket(0x12);
CAN.write('h');
CAN.write('e');
CAN.write('l');
CAN.write('l');
CAN.write('o');
CAN.endPacket();

Serial.println("done");

delay(1000);

// send extended packet: id is 29 bits, packet can
Serial.print("Sending extended packet ... ");

CAN.beginExtendedPacket(0xabcdef);
CAN.write('w');
```

Figura 60. Estructura para enviar un mensaje usando CAN.h

Para el receptor, se utiliza la función *CAN.parsePacket* para determinar el tamaño del paquete recibido, luego se determina si la trama es una trama extendida o un "rtr" que corresponde a una trama de solicitud y luego con *CAN.read* se leen los valores recibidos. En la figura 61 se muestra la estructura del receptor y en la figura 62, la imagen del monitor.

```
int packetSize = CAN.parsePacket();

if (packetSize) {
  // received a packet
  Serial.print("Received ");

  if (CAN.packetExtended()) {
    Serial.print("extended ");
  }

  if (CAN.packetRtr()) {
    // Remote transmission request, packet contains no data
    Serial.print("RTR ");
  }
  Serial.print("packet with id 0x");
  Serial.print(CAN.packetId(), HEX);

  if (CAN.packetRtr()) {
    Serial.print(" and requested length ");
    Serial.println(CAN.packetDlc());
  } else {
    Serial.print(" and length ");
    Serial.println(packetSize);
    // only print packet data for non-RTR packets
    while (CAN.available()) {
      Serial.print((char)CAN.read());
```

Figura 61. Estructura del receptor usando CAN.h

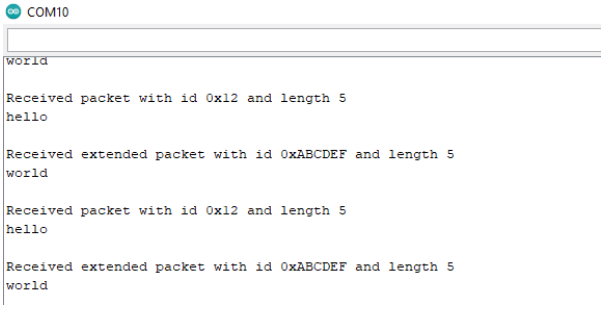

Figura 62. Salida del monitor del receptor con CAN.h

Transmisor y receptor de datos con placas Arduino, usando las librerías de Arduino-mcp2515-master y mcp_CAN-lib-master.

Las siguientes paqueterías usadas fueron las subidas por los usuarios coryjfowler [6] y autowp [8]. Los cuales ocupan las librerías "mcp_can.h" y "mcp2515.h" respectivamente, ambas acompañadas por la librería de "SPI.h". Se ejecutaron los códigos de receptor y transmisor los cuales venían incluidos como ejemplos en el paquete de las librerías, ambas librerías funcionaron correctamente al ejecutar la red CAN. La conexión con el MCP2515 es la misma que con la librería anterior. A continuación, se describen las funciones para enviar y recibir datos.

Al iniciar el código con la librería mpc_can,h, se asigna el "INT" del MCP2515 que corresponde al pin 2 de Arduino en el código del receptor y en ambos nodos, receptor y transmisor, el CS al pin 10, luego se configura el inicio del CAN mediante la función *CAN0.begin* y el modo de funcionamiento del CAN con *CAN0.setMode()*, la cual solicita el tipo de trama en el primer argumento, luego la velocidad y finalmente la frecuencia del buffer, como se muestra en la siguiente figura.

```
#define CAN0_INT 2                              // Set INT to pin 2
MCP_CAN CAN0(10);                               // Set CS to pin 10


void setup()
{
  Serial.begin(115200);

  // Initialize MCP2515 running at 16MHz with a baudrate of 500kb/s and the masks and filters disabled.
  if(CAN0.begin(MCP_ANY, CAN_500KBPS, MCP_16MHZ) == CAN_OK)
    Serial.println("MCP2515 Initialized Successfully!");
  else
    Serial.println("Error Initializing MCP2515...");

  CAN0.setMode(MCP_NORMAL);                     // Set operation mode to normal so the MCP2515 sends acks to received data.

  pinMode(CAN0_INT, INPUT);                            // Configuring pin for /INT input
```

Figura 63. Configuración inicial para mcp_can.h

Para enviar mensajes se utiliza la función *CAN0.sendMsgBuf()* en donde se agrega como primer argumento el identificador en formato sexagesimal, seguido del tipo de trama, el número de datos y finalmente los datos de la trama almacenados en forma de matriz de tipo byte, como se observa en la figura.

```
byte data[8] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};

void loop()
{
  // send data:  ID = 0x100, Standard CAN Frame, Data length = 8 bytes, 'data' = array of data bytes to send
  byte sndStat = CAN0.sendMsgBuf(0x100, 0, 8, data);
  if(sndStat == CAN_OK){
```

Figura 64. Envío de datos con mcp_can.h

En el caso del receptor, lo primero que se hace es declarar unas variables con las que trabaja el código, de la forma que se muestra en la siguiente figura.

```
long unsigned int rxId;
unsigned char len = 0;
unsigned char rxBuf[8];
char msgString[128];
```

Figura 65. asignación de variables para el identificador, largo de la trama, buffer y tamaño.

Para leer los datos recibidos se utiliza la función *CAN0.readMsgBuf()* dentro se escriben los parámetros declarados anteriormente, que son el identificador, tamaño y buffer. Luego se lee el identificador para determinar el tipo de trama y se imprime la trama. Figura 66.

```
void loop()
{
  if(!digitalRead(CAN0_INT))                         // If CAN0_INT pin is low, read receive buffer
  {
    CAN0.readMsgBuf(&rxId, &len, rxBuf);      // Read data: len = data length, buf = data byte(s)

    if((rxId & 0x80000000) == 0x80000000)     // Determine if ID is standard (11 bits) or extended (29 bits)
      sprintf(msgString, "Extended ID: 0x%.8lX  DLC: %1d  Data:", (rxId & 0x1FFFFFFF), len);
    else
      sprintf(msgString, "Standard ID: 0x%.3lX       DLC: %1d  Data:", rxId, len);
```

Figura 66. Estructura para recibir datos con mcp_can.h

En el caso de la librería mcp2515.h se debe establecer una estructura para cada trama CAN que se envíe y una para leer. De igual manera se establece la velocidad de bits, la frecuencia del controlador CAN y el modo con el que trabajará, usando las funciones *mcp2515.setBitrate()* y *mcp2515setNormalMode()* respectivamente, como se muestra en la siguiente figura.

```
struct can_frame canMsg;
MCP2515 mcp2515(10);


void setup() {
  Serial.begin(115200);

  mcp2515.reset();
  mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ);
  mcp2515.setNormalMode();
```

Figura 67. Configuración inicial CAN con mcp2515.h

Para enviar mensajes se utiliza la configuración con la estructura establecida anteriormente *canMsg1* seguido de: *.can_id*, *.can_dlc* o *.data[].* para el asignar el identificador, el DLC y los datos para cada byte de la trama CAN respetivamente, después simplemente se envía con la función *mcp2515sendMessage()*. En la siguiente figura se muestra un ejemplo del transmisor de datos.

```
canMsg1.can_id  = 0x0F6;
canMsg1.can_dlc = 8;
canMsg1.data[0] = 0x8E;
canMsg1.data[1] = 0x87;
canMsg1.data[2] = 0x32;
canMsg1.data[3] = 0xFA;
canMsg1.data[4] = 0x26;
canMsg1.data[5] = 0x8E;
canMsg1.data[6] = 0xBE;
canMsg1.data[7] = 0x86;

mcp2515.sendMessage(&canMsg1);
delay(50);
```

Figura 68. Estructura del transmisor CAN con mcp2515.h

Para recibir los datos se hace uso de la función *mcp2515.readMessage()*, se comprueba que se reciban correctamente los datos con *MCP2515::ERROR_OK* y luego se imprime usando la estructura utilizada con el transmisor, como se muestra en la figura 68.

```
void loop() {
  if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {
    Serial.print(canMsg.can_id, HEX); // print ID
    Serial.print(" ");
    Serial.print(canMsg.can_dlc, HEX); // print DLC
    Serial.print(" ");

    for (int i = 0; i<canMsg.can_dlc; i++)  {  // print the data
      Serial.print(canMsg.data[i],HEX);
      Serial.print(" ");
    }
```

Figura 69. Receptor de datos con mcp2515.h

En la siguiente figura se muestra el monitor serial del receptor estas librerías, donde se pueden ver las tramas de datos con el identificar, DLC y datos.

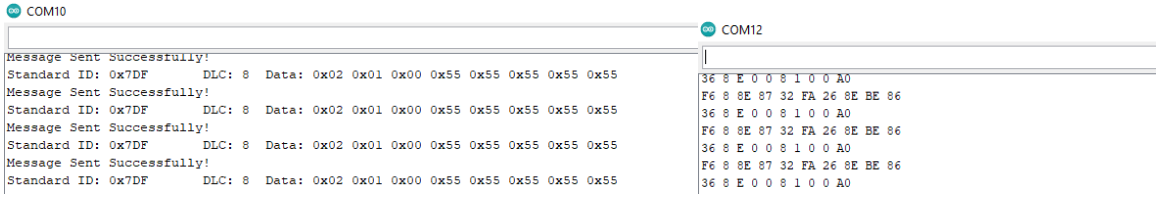

Figura 70. Salidas del monitor de los receptores, a la izquierda con la librería mcp_can.h y a la derecha mcp2515.h

Transmisor y receptor de datos con placas ESP, usando las librerías de ESP32-Arduino-CAN-master.

Como se mencionó anteriormente se usó la librería de "can.h" la cual funciona con el MCP2515 y el ESP. Sin embargo, también se encuentran en *github* librerías que funcionan usando el controlador CAN integrado en placa ESP32 en conjunto con un transceptor CAN, como la que subió el usuario miwagner [39], que ocupa las librerías "Arduino.h", "ESP32CAN.h" y "CAN_config.h" en conjunto. Para hacer la conexión del ESP32 al transceptor se usó la misma configuración de pines que se mencionó anteriormente con la librería "can.h" como se muestra en la tabla 6. En la figura 71 se muestra la conexión física de los nodos. Y, de igual manera, se ejecutó el código de ejemplo contenido en el paquete que contiene lo necesario para enviar y recibir mensajes en un solo código, el cual funcionó correctamente.

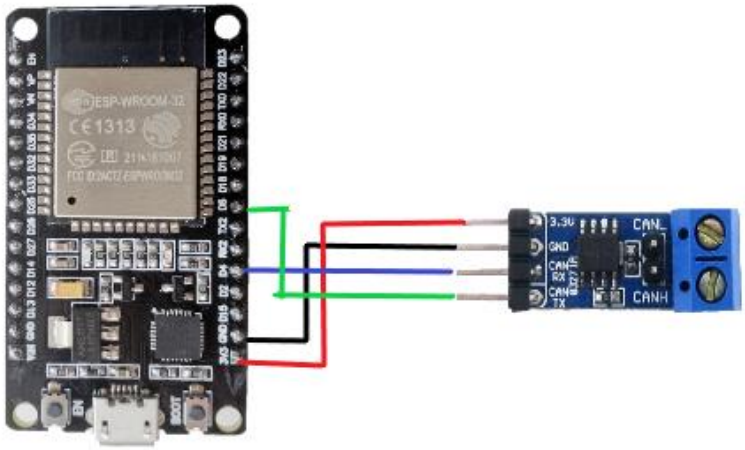

Figura 71. Diagrama de conexión entre la placa ESP32 y el transceptor SN65HVD230.

El código, al igual que el anterior funciona definiendo una estructura mediante *CAN_device_t* como se muestra en la figura, luego con esta estructura se establecen los parámetros con los que trabajará el controlador CAN, como la velocidad, los pines de los canales CAN o el número de tramas que se pueden almacenar en el buffer, con las terminaciones *.speed, .tx,tx_pin_id* y *queue* respectivamente. Luego se inicia con la función *ESP32CAN.CANInit().*

```
CAN_device_t CAN_cfg;               // CAN Config
unsigned long previousMillis = 0;   // will store last time a CAN Message was send
const int interval = 1000;          // interval at which send CAN Messages (milliseconds)
const int rx_queue_size = 10;       // Receive Queue size

void setup() {
  Serial.begin(115200);
  Serial.println("Basic Demo - ESP32-Arduino-CAN");
  CAN_cfg.speed = CAN_SPEED_125KBPS;
  CAN_cfg.tx_pin_id = GPIO_NUM_5;
  CAN_cfg.rx_pin_id = GPIO_NUM_4;
  CAN_cfg.rx_queue = xQueueCreate(rx_queue_size, sizeof(CAN_frame_t));
  // Init CAN Module
  ESP32Can.CANInit();
}
```

Figura 72. Configuración e iniciación CAN para CAN_config.h

Para enviar mensajes se crea la estrustura para la trama de datos CAN con el comando *CAN_FRame_t tx_frame* y al igual que algunas otras librerías, con las terminaciones se definen las partes de la trama, como el tipo, ID, DLC, y datos. Luego se envía el mensaje con la función *ESP32Can.CANWriteFrame().* En la figura 73 se muetra un ejemplo del envío.

```
// Send CAN Message
if (currentMillis - previousMillis >= interval) {
  previousMillis = currentMillis;
  CAN_frame_t tx_frame;
  tx_frame.FIR.B.FF = CAN_frame_std;
  tx_frame.MsgID = 0x001;
  tx_frame.FIR.B.DLC = 8;
  tx_frame.data.u8[0] = 0x00;
  tx_frame.data.u8[1] = 0x01;
  tx_frame.data.u8[2] = 0x02;
  tx_frame.data.u8[3] = 0x03;
  tx_frame.data.u8[4] = 0x04;
  tx_frame.data.u8[5] = 0x05;
  tx_frame.data.u8[6] = 0x06;
  tx_frame.data.u8[7] = 0x07;
  ESP32Can.CANWriteFrame(&tx_frame);
```

Figura 73. Estructura para el envío de datos con ESP32CAN.h

Para leer los mensajes, se checa el almacenamiento del buffer y se determina si es una trama estándar o extendida con el comando *rx_frame.FIR.B.FF == CAN_frame_stad*, luego se comprueba si no es un trama RTR y de no ser así se imprimen los componentes de la trama usando la misma estructura que con el transmisor. A continuación se muestra la estructura para recibir los mensajes.

```
// Receive next CAN frame from queue
if (xQueueReceive(CAN_cfg.rx_queue, &rx_frame, 3 * portTICK_PERIOD_MS) == pdTRUE) {

  if (rx_frame.FIR.B.FF == CAN_frame_std) {
    printf("New standard frame");
  }
  else {
    printf("New extended frame");
  }

  if (rx_frame.FIR.B.RTR == CAN_RTR) {
    printf(" RTR from 0x%08X, DLC %d\r\n", rx_frame.MsgID,  rx_frame.FIR.B.DLC);
  }
  else {
    printf(" from 0x%08X, DLC %d, Data ", rx_frame.MsgID,  rx_frame.FIR.B.DLC);
    for (int i = 0; i < rx_frame.FIR.B.DLC; i++) {
      printf("0x%02X ", rx_frame.data.u8[i]);
    }
```

Figura 74. Estructura para recibir datos con ESP32CAN.h

En la salida del monitor serial del receptor, figura 76, se muestra la trama recibida y en la figura 75 se muestra la conexión física de los nodos con sus respectivos dispositivos.

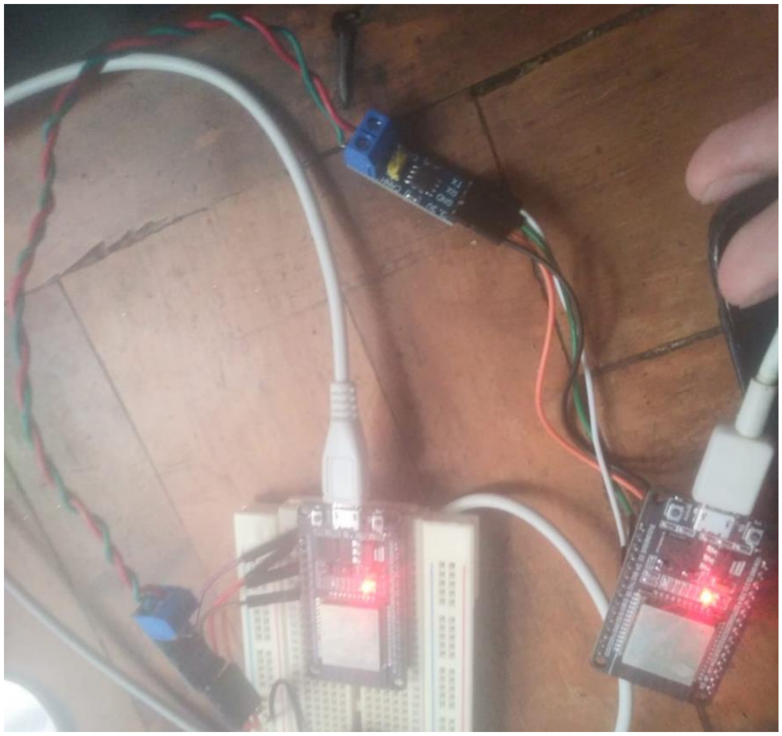

Figura 75. Red CAN-bus usando ESP32 y SN65HVD230.

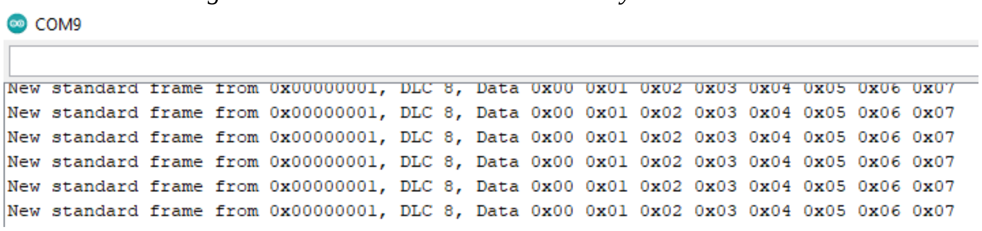

Figura 76. Salida del monito serial del receptor usando las librerías ESP32CAN.h y CAN_config.h

Transmisor y receptor de datos con placas ESP32 y Arduino, usando la librería de Esp32-mcp2515-master.

Al realizar la investigación de las librerías y dispositivos se encontraron dos fuentes en las que se ocupa la ESP32 en conjunto con el MCP2515 para realizar una comunicación CAN-bus [37][38]. Sin embarga en una de estas no se muestra el diagrama de conexión y otra placa que no es la *dev Kit1*. Afortunadamente el otro documento si mostraba un diagrama de conexión entre la ESP32 y el MCP2515, figura 76. Así que se buscó y encontró en *Github* un paquete de librerías subido por el usuario dedalqq [19], el cual adapta la librería del mcp2515.h ya mencionada, para que sea compatible con dispositivos ESP32/ESp8266. Se realizaron pruebas de envío y recepción de datos entre la ESP32 y una de Arduino Uno. En la tabla 7 se muestran los pines designados del ESP32 para la conexión al MCP2515. Como se puede apreciar el pin INT no se conecta a ningún pin del ESP32 por lo que se queda libre y no se ocupa.

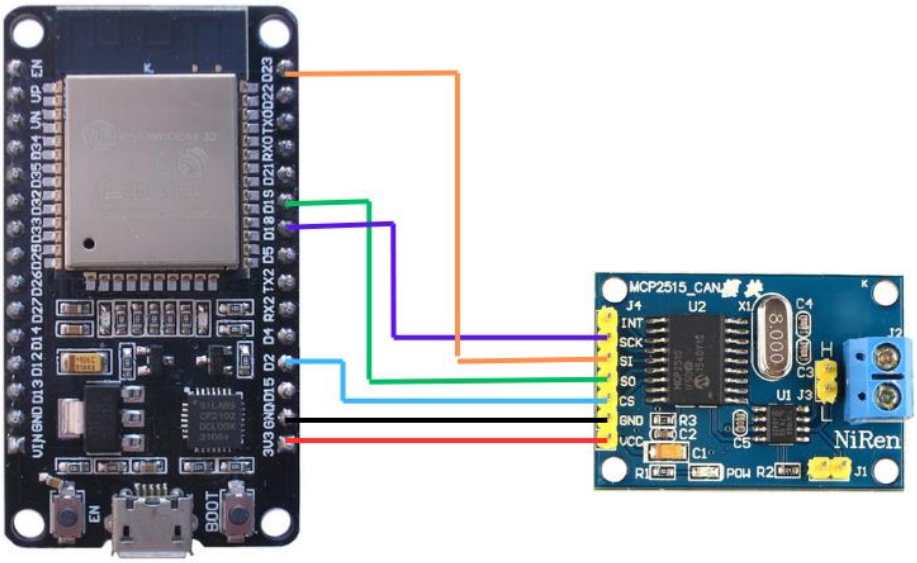

Figura 77. Diagrama de conexión entre la ESP32 y MCP2515 [37]

MCP2515	**ESP32**
VCC	3V3
GND	GND
SCK	GPIO_18
SI	GPIO_23
SO	GPIO_19
CS	GPIO_2

Tabla 7. Relación de pines entre el MCP25215 y el ESP32

La estructura del código es la misma que vio anteriormente, la única diferencia entre el código usado en la ESP32 con el de Arduino es en el comando: "***MCP2515*** *mcp2515(10)*" donde en vez de 10 se coloca un 2 que es el pin al que se conecta el CS y aun que el voltaje de funcionamiento del MCP sea de 5V al parecer este funciona correctamente con los 3V del ESP32. En la siguiente figura se muestra la conexión entre el ESP32 y Arduino a través del can bus del MCP2515.

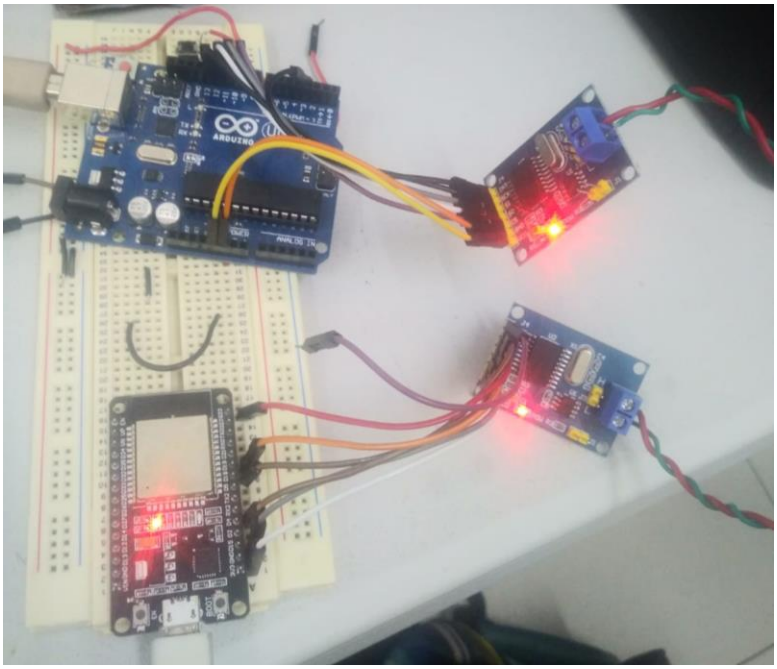

Figura 78. Conexión física entre las placas ESP32 y Arduino usando el MCP2515

Se armó la red CAN-bus usando el MCP2515 con ambas placas ESP32 y también ESP32 con Arduino y ambas funcionaron correctamente. En la siguiente figura se muestra el monitor serial de una placa Arduino recibiendo datos de una placa ESP32, como se puede observar en la imagen a pesar de que el ESP32 funciona con 3.3V los datos enviados no son distorsionados al ser recibidos por el receptor de Arduino.

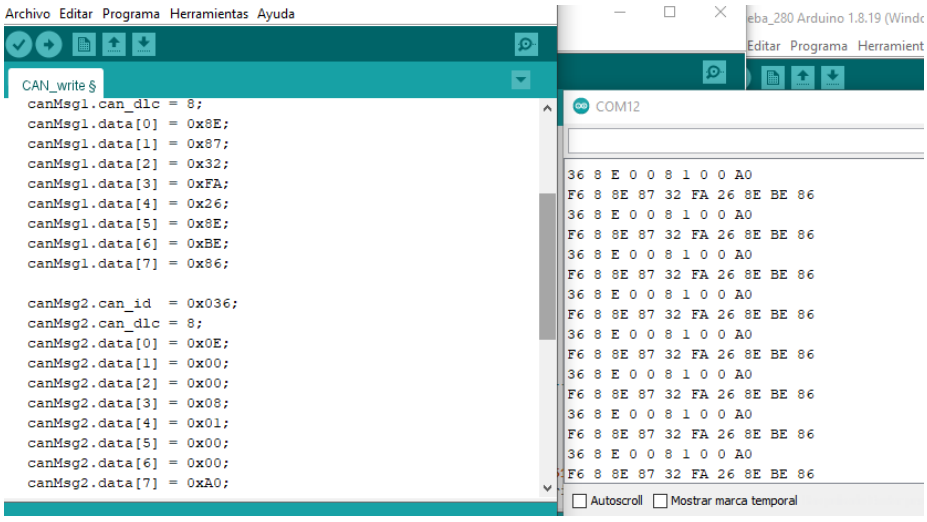

Figura 79. comparación de datos enviados por la ESP32 y recibidos por un receptor Arduino

3.4 Proceso de recepción y obtención de datos

El siguiente paso fue configurar el código para enviar una trama de petición de PID, que según la norma SAE1939 debe tener la siguiente estructura: ID de 7DE, DLC de 8, en el primer byte va el número de bytes extras, en este caso 2, el siguiente, byte 1 que especifica el modo del OBDII, los cuales se especificaron en el capítulo 2, por lo que se usó el modo 1 que corresponde al envío de datos en tiempo real y el siguiente que se ocupa, el byte 2 contiene el PID que se requiera, en el apartado de anexos C, la lista de PID de los cuales se ocuparon los siguientes:

- 05. Correspondiente a la temperatura del líquido de enfriamiento del motor.
- 0C. Indica las revoluciones por minuto.
- 0D. Que corresponde a la velocidad del vehículo.
- 0F. La temperatura del aire del colector de admisión.
- 10. Salida del sensor de flujo de aire MAF.

Mediante la función "*switch-case*" se programó el microcontrolador para que leyera el byte 2 del mansaje recibido que corresponde al PID, para que al recibir un PID la función lo cambie por el siguiente, en la figura 80 se muestra este proceso. Se probó nuevamente en el OBDII del *beetle* el cual inesperadamente respondió enviando los mensajes con la información solicitada. Esto pudo deberse a que el OBDII cuenta con un protocolo de seguridad que impide que se reciba información a menos que sea mediante una solicitud como se especifica en la normal SAE1939.

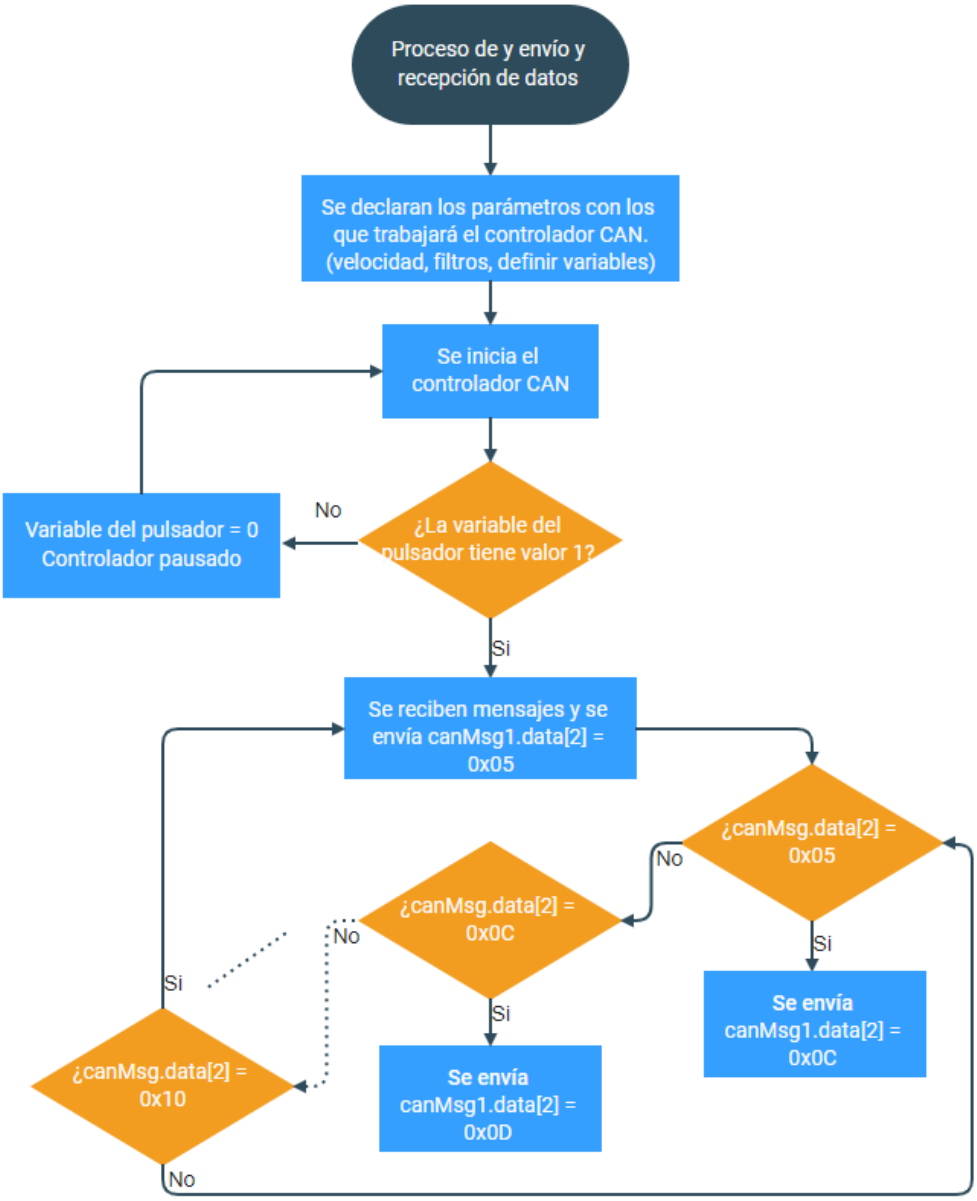

Figura 80. Diagrama de datos del proceso para enviar y recibir tramas de mensaje de solicitud de PID

Capítulo 4

Pruebas de funcionamiento

4.1 Prueba de comunicación con cuadro de instrumentos (clúster)

Después de realizar las redes CAN y analizar los diferentes códigos, se comprobó que el módulo CAN con los distintos códigos se comunique correctamente en el CAN-bus de un auto, sin embargo, para tener un mejor espacio y comodidad al hacer las pruebas se utilizó primero un cuadro de instrumentos. Se ocupó el clúster 6J0920801X del vehículo Ibiza, con el que cuenta el laboratorio automotriz, para realizar pruebas y comprobar si se puede recibir y enviar mensajes CAN. El clúster cuenta con un módulo de interfaz CAN con el que se comunica con el resto de módulos y sensores en el vehículo. Las librerías ocupadas fueron la "ESP32CAN.h", "mcpCAN.h" y la "mcp2515.h", debido a que puede modificar la velocidad de transmisión, agregar máscaras y filtros y modificar el ID, DLC y datos de cada trama.

Lo primero es identificar el conector del clúster, el cual se encuentra en la parte de atrás y está fijado al cuadro de instrumentos, este es de forma rectangular y cuenta con 32 pines, como se muestra en la figura 81. Con ayuda del trabajo de tesis de la lic. Lady en la cual se trabajó con el mismo clúster sabemos que los pines los pines 28 y 29 son para el CAN High y CAN Low respectivamente, el pin 31 tierra de la batería y los pines 16 y 32 van a la alimentación. El clúster se alimentó usando una fuente de corriente directa y con un voltaje de 12V, para simular el voltaje de la batería del vehicula con la cual funciona.

Figura 81. Conector del del cuadro de instrumentos

Para hacer las pruebas se ocuparon 2 identificadores obtenidos del clúster usando el PXI, como se especifica en el trabajo de Lady mencionado anteriormente, los identificadores son el 280 que corresponde a las RPM y el 470 que se encarga de encender algunas luces testigo en el cuadro de instrumentos. Ahí también se menciona que identificador 280 tiene el valor de las revoluciones en el byte de datos 3 y en el caso del 470 el primer byte corresponde a las luces direccionales, el byte 0 corresponde a las luces direccionales, el byte 01 enciende el testigo de que hay puertas abiertas, el byte 02 modifica la luminosidad del tablero, el byte 03 no envía datos, el byte 04 notifica un problema en una bombilla, el byte 05 y 06 están libre s y el byte 07 enciende el testigo de la luz antiniebla, carretera o ambas.

Se implementó un código con cada una de las librerías mencionadas en capitulo, 3 para enviar una trama CAN cada 100 milisegundos con identificador 280 y un valor para las revoluciones en el byte asignado, para lo cual se utilizó un potenciómetro y la lectura analógica de Arduino y ESP32 para obtener un valor de revoluciones el cual se agrega en el byte 3 de la trama. En los códigos se configura en controlador CAN a una velocidad de transmisión de bits a 500kbps y la frecuencia de oscilación del cristal a 8MHz, que es la velocidad con la que trabaja el CAN tracción, según la SAE J1979 (norma que establece la transferencia, conexión y códigos para trabajar con el OBDII). La conexión entre el clúster y el transceptor se muestra en la figura 82. Para realizar los códigos se tomó en cuenta que cada una de las librerías tiene su propia estructura para trabajar la comunicación CAN, como se vio anteriormente. Una vez cargados los códigos al microcontrolador se realizó la conexión entre el clúster, con el transceptor y microcontrolador junto con la resistencia de 120ohms. Como se muestra la siguiente figura 83.

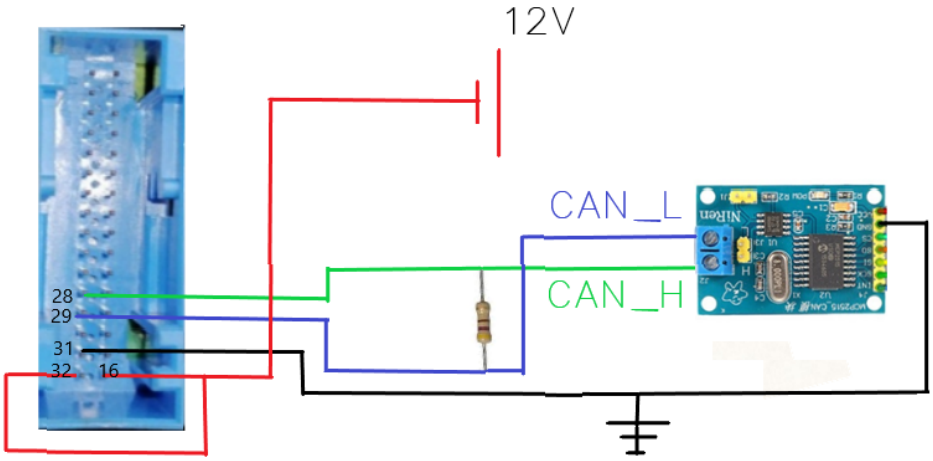

Figura 82. Diagrama de conexión al CAN-bus

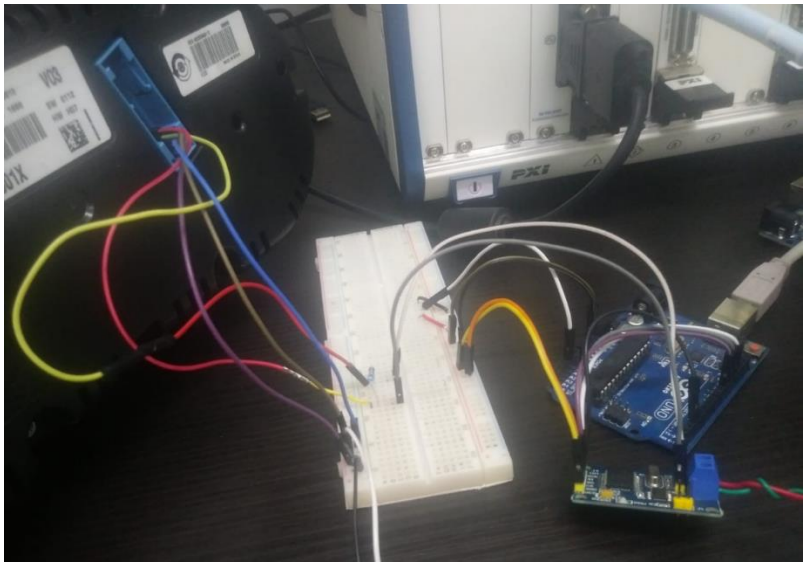

Figura 83. Conexión CAN entre el clúster y el MCP2515 en conjunto con Arduino

Desafortunadamente al hacer las pruebas y enviar el mensaje al clúster, este no recibía las tramas enviadas por los dispositivos, por lo que se conectó al osciloscopio automotriz para ver si este reconocía los mensajes y aunque sí mostraba una señal similar a un mensaje CAN, no se reconocían los mensajes. Este caso fue el mismo para todas las librerías, únicamente reconoció los mensajes enviados por la librería "mcp2515.h", lo que provocó que la ajuga indicadora de las revoluciones cambie su posición, como se muestra en la figura 84. De igual manera se comprobó por el osciloscopio para ver si se tenía el mismo error, pero este si logró codificar la trama, figura 85.

Figura 84. Cuadro de instrumentos, a la izquierda sin recibir mensa y a la derecha recibiendo el valor de las revoluciones en la trama CAN

Figura 85. Trama de datos CAN enviada vista con el osciloscopio

De igual manera se realizó un programa con el identificador de 470, para ver si el clúster encendía algunas de las luces testigos, los datos enviados son los siguientes; en el byte 0 un valor de 3, en el byte 01 de 21, en el byte 02 de 32, en el byte 03 de 0, del byte 04 al 06 un valor de 1 y en el byte 07 un valor de 40.

Se observa que el clúster reconoció el mensaje y encendió algunas luces testigo indicadas en la trama del identificador 470. Como la direccional derecha, el testigo de puerta abierta y el de luces antiniebla, como se muestra en la figura siguiente. Y a pesar de que los testigos diferían un poco de lo mencionado en el trabajo de Lady, funcionó para realizar la prueba. Los códigos se muestran en el apartado de anexos.

Figura 86. Testigos encendidos con el ID 470

Afortunadamente la librería que funcionó correctamente es compatible con el ESP32, como se menciona en el capítulo anterior. Se realizó la misma prueba con resultados positivos. El clúster reconoció los datos provenientes del ESP32. En la figura 87 se muestra la conexión del ESP32 al CAN bus del clúster a través del MCP2515 y a otro MCP2515 conectado a Arduino con un código receptor. El código se encuentra en el apartado B de los anexos. Esto con el objetivo de observar los mensajes enviados por el clúster a través del CAN-bus y ver si funciona la recepción de mensajes.

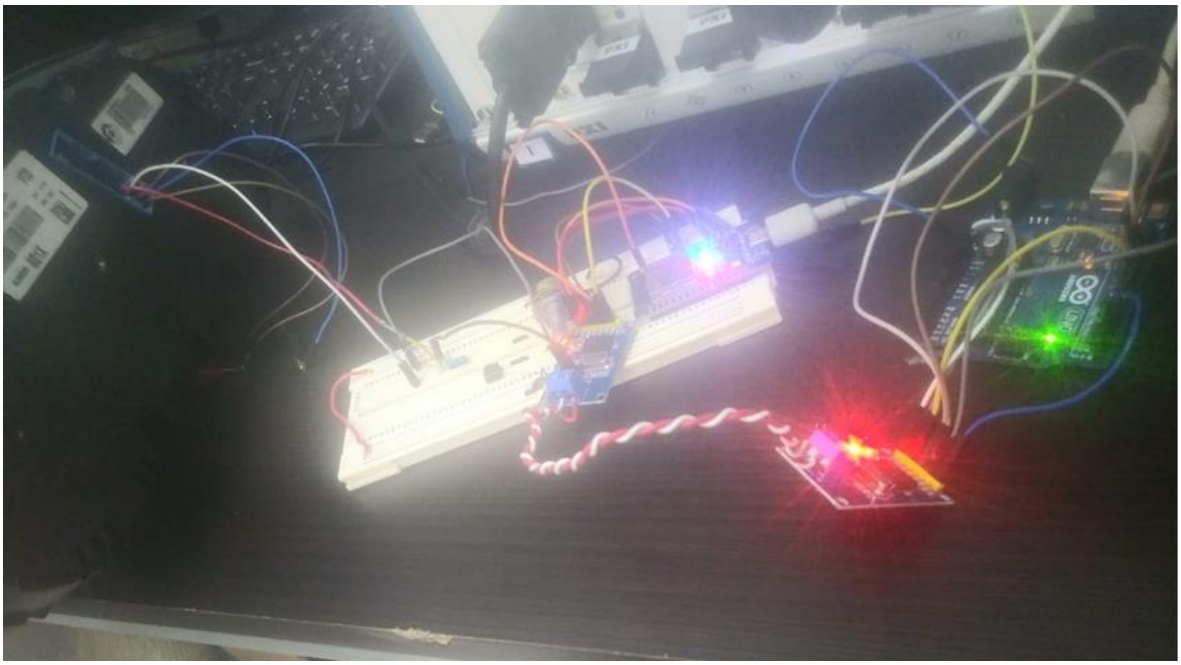

Figura 87. Red CAN clúster, ESP32 y Arduino

En la figura 88 se observa el monitor serial del Arduino con el código receptor, donde se muestran las tramas de datos recibidas, donde el primer dato corresponde al ID, el segundo al DLC y los que siguen a los bytes de datos del mensaje, entre los mensajes recibidos se encuentra la trama emitida por el ESP32 con el identificador 280 y los diferentes valores para los rpm.

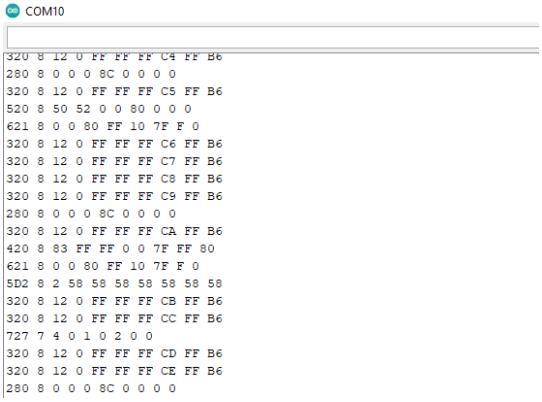

Figura 88. Tramas CAN recibidas del clúster y el ESP32 por Arduino

4.2 Prueba de la base de datos en tiempo real y el ESP32

La librería instalada en el capítulo anterior cuenta con varios ejemplos para trabajar con *Firebase*, entre ellos se encuentra el de “basic” el cual se usó para probar el funcionamiento de la base de datos en tiempo real *Realtime Database*. En la figura 89 se muestra en donde se encuentra el ejemplo utilizado.

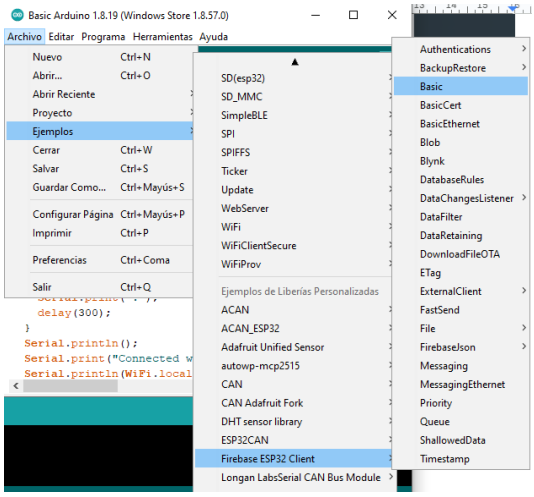

Figura 89. Proyecto de prueba “basic”

Al código se le agrega la información de *firebase* identificada anteriormente, junto con los datos del internet para el módulo wifi, como se muestra en la figura 90. Aquí se agregan entre comillas y donde corresponde el nombre de la red wifi, la contraseña, la llave API, el URL de la base de datos, el Email y la contraseña asignada a este.

```
/* 1. Define the WiFi credentials */
#define WIFI_SSID "WIFI_AP"
#define WIFI_PASSWORD "WIFI_PASSWORD"

// For the following credentials, see examples/Authentications/SignInAsUser/EmailPassword/EmailPassword.ino

/* 2. Define the API Key */
#define API_KEY "API_KEY"

/* 3. Define the RTDB URL */
#define DATABASE_URL "URL" //<databaseName>.firebaseio.com or <databaseName>.<region>.firebasedatabase.app

/* 4. Define the user Email and password that alreadey registerd or added in your project */
#define USER_EMAIL "USER_EMAIL"
#define USER_PASSWORD "USER_PASSWORD"
```

Figura 90. Datos ingresados para trabajar con Realtime Database

El código se conecta a una red wifi y a la base de datos con los parámetros especificados. Luego envía diferentes tipos de valores (bool, int, float, double y string) a la base de datos y los lee cada 15 segundos, “true” o “false” para el booleano, la variable “count” para enteros, “count + 10.2” para flotantes, “count + 35.17…” para dobles y “Hello word” para letras. Como se observa en la figura 91. Se utiliza la función *Firebase.get()* junto el tipo de variable para leer los datos y *Firebase.set*() para enviar los valores, la estructura de la función va de la siguiente manera: (dato de firebase, ubicación en donde se verá el dato, valor).

```
void loop()
{
  if (Firebase.ready() && (millis() - sendDataPrevMillis > 15000 || sendDataPrevMillis == 0))
  {
    sendDataPrevMillis = millis();
    Serial.printf("Set bool... %s\n", Firebase.setBool(fbdo, F("/test/bool"), count % 2 == 0) ? "ok" : fbdo.errorReason().c_str());
    Serial.printf("Get bool... %s\n", Firebase.getBool(fbdo, FPSTR("/test/bool")) ? fbdo.to<bool>() ? "true" : "false" : fbdo.errorReason().c_str());
    bool bVal;
    Serial.printf("Get bool ref... %s\n", Firebase.getBool(fbdo, F("/test/bool"), &bVal) ? bVal ? "true" : "false" : fbdo.errorReason().c_str());
    Serial.printf("Set int... %s\n", Firebase.setInt(fbdo, F("/test/int"), count) ? "ok" : fbdo.errorReason().c_str());
    Serial.printf("Get int... %s\n", Firebase.getInt(fbdo, F("/test/int")) ? String(fbdo.to<int>()).c_str() : fbdo.errorReason().c_str());
    int iVal = 0;
    Serial.printf("Get int ref... %s\n", Firebase.getInt(fbdo, F("/test/int"), &iVal) ? String(iVal).c_str() : fbdo.errorReason().c_str());
    Serial.printf("Set float... %s\n", Firebase.setFloat(fbdo, F("/test/float"), count + 10.2) ? "ok" : fbdo.errorReason().c_str());
    Serial.printf("Get float... %s\n", Firebase.getFloat(fbdo, F("/test/float")) ? String(fbdo.to<float>()).c_str() : fbdo.errorReason().c_str());
    Serial.printf("Set double... %s\n", Firebase.setDouble(fbdo, F("/test/double"), count + 35.517549723765) ? "ok" : fbdo.errorReason().c_str());
    Serial.printf("Get double... %s\n", Firebase.getDouble(fbdo, F("/test/double")) ? String(fbdo.to<double>()).c_str() : fbdo.errorReason().c_str());
    Serial.printf("Set string... %s\n", Firebase.setString(fbdo, F("/test/string"), "Hello World!") ? "ok" : fbdo.errorReason().c_str());
    Serial.printf("Get string... %s\n", Firebase.getString(fbdo, F("/test/string")) ? fbdo.to<const char *>() : fbdo.errorReason().c_str());
```

Figura 91. Estructura para enviar y recibir datos de Realtime Database

También se envía información en formato “Json” donde si el valor de “count” es igual a 0 se escribe “cool” y si es diferente se escribe “Smart. Figura 92.

```
  FirebaseJson json;

  if (count == 0)
  {
    json.set("value/round/" + String(count), F("cool!"));
    json.set(F("vaue/ts/.sv"), F("timestamp"));
    Serial.printf("Set json... %s\n", Firebase.set(fbdo, F("/test/json"), json) ? "ok" : fbdo.errorReason().c_str());
  }
  else
  {
    json.add(String(count), "smart!");
    Serial.printf("Update node... %s\n", Firebase.updateNode(fbdo, F("/test/json/value/round"), json) ? "ok" : fbdo.errorReason().c_str());
  }
```

Figura 92. Envio de datos en formato Json

Después se sube el código al ESP32 y en *firebase* en el apartado de “realtime database” y en la primera opción que se muestra “datos”, podemos observar que se creó una base de datos con el nodo “test”, los diferentes sub-nodos dependiendo del tipo de dato y los diferentes datos enviados en el código del ESP32, figura 93. Por lo cual el código funciona correctamente y se usará más adelante para enviar la información del automóvil.

Figura 93. Datos recibidos y almacenados en Realtime Database

4.3 Prueba de envío y recepción de datos con el proyecto (app) en Android Studio y Realtime Database

Para realizar las pruebas y aunque *Android Studio* cuenta con un emulador con diferentes modelos de celular y diferentes versiones de Android, se trabajó instalando la aplicación en el dispositivo *moto g*

power. Debido a que la velocidad de la computadora con la que se trabajó el proyecto disminuía su rendimiento considerablemente, por lo que resultó más fácil usar el celular personal para ver el funcionamiento de la app.

Lo primero es importar las paqueterías necesarias para trabajar con *Realtime database* las cuales se muestran en la siguiente figura.

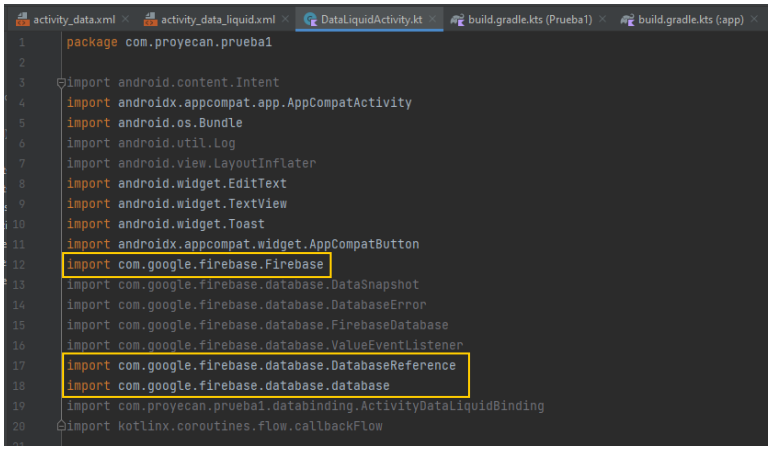

Figura 94. Paqueterías usadas en una actividad de la app

Como se comentó en el capítulo 3.2.7, se creó una actividad para los datos de cada sensor, los cuales se deben adquirir de la base de datos. Para estas actividades se hicieron pruebas para leer datos de la nube y para añadir datos a la misma. Por defecto se crea una función tipo "class" en la actividad, aquí se deber crear una variable a la que se haga referencia la base de datos, usando ":DatabaseReference", de igual manera se debe asignar una variable a los textos y botones de la actividad, usando el comando "findViewById<*elemento que se busca*>(R.id.*id asignado*)". Para escribir los datos se agrega un nodo nuevo a la base de datos con el comando ".child()", luego se agrega un texto con el comando ".setValue()" y finalmente con la función ".addOnSuccessListener{}" agrega un mensaje para el caso en el que se haya subido el texto correctamente o en caso contrario un mensaje de error, figura 95.

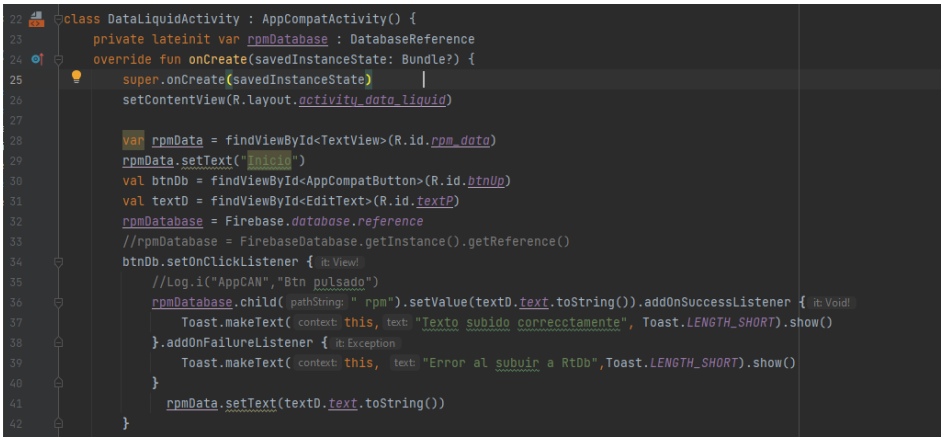

Figura 95. Código para escribir en la base de datos (realtime database)

Para leer información de la base de datos se hace la referencia y se agrega el nodo que se desea leer, al igual que en el caso de la escritura, luego se usa la función ": ValueEventListener{}" en conjunto con la función "onDataChange{}" para que al momento en que cambie la ruta del valor que se solicita en la base de datos, este se lea y actualice. Esta función genera una salida de tipo "dataSnapshot", la cual se almacena en una variable y junto con el comando ".getValue()" almacena el valor de la base de datos, el cual se muestra en un texto con el comando ".setText".

```
//      val rpmListener = object : ValueEventListener {
//          override fun onDataChange(dataSnapshot: DataSnapshot) {
//              // Get Post object and use the values to update the UI
//              if (dataSnapshot.exists()) {
//                  val list = dataSnapshot.getValue()
//                  rpmData.setText(list.toString())
//              }
//          }
//          override fun onCancelled(databaseError: DatabaseError) {
//              rpmData.setText("Error")
//          }
//          // ...
//
//      }
//      rpmDatabase.addValueEventListener(rpmListener)
```

Figura 96. Código para leer datos de la base de datos (realtime database)

4.4 Prueba con un vehículo

Finalmente se realizaron pruebas en un vehículo real para verificar que el sistema funcione correctamente, para lo cual se necesita un auto con CAN bus, condición con la que cuenta cualquier vehículo que sea al menos modelo 2000. Para esto, el laboratorio automotriz cuenta con 2 vehículos, un *Jetta Hybrid.* Figura 97. Y un *Volkswagen Beetle.* Figura 98.

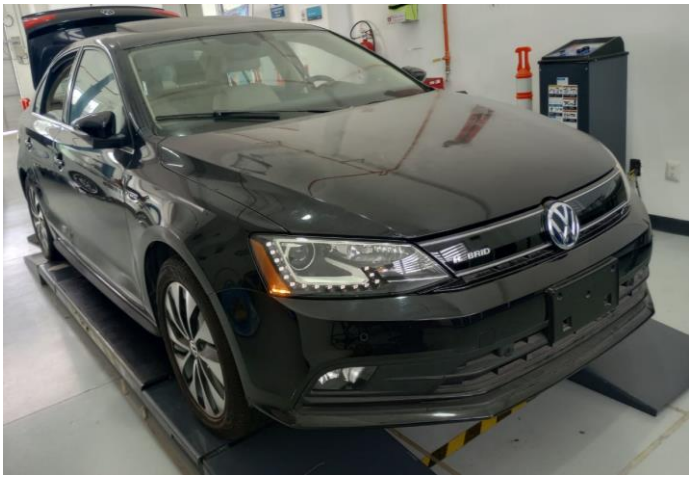

Figura 97. Jetta Hybrid.

Figura 98. Volkswagen Beetle

Se probo un código para asegura que se pueden recibir tramas de datos CAN, para esto se utilizó el código receptor usado anteriormente para recibir información del clúster, (apartado B de anexos). Este código se probó con Arduino uno y ESP32. Primero se buscó una conexión en el *beetle* a través del OBDII, el cual se encuentra debajo del tablero de instrumentos, a la izquierda, cerca de la puerta como se ve en la figura 99. Se identificaron los pines en los que se encuentra el CAN Low y en Can High, los cuales se encuentran en los pines 6 y 14 respectivamente, como se vio en el capítulo 2, figura 3.

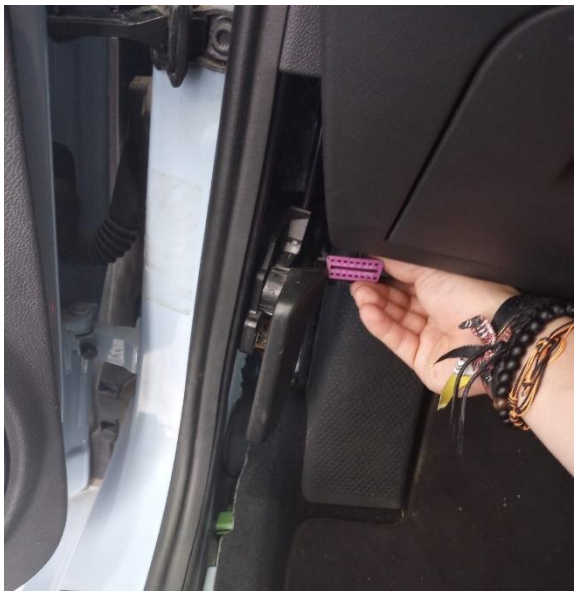

Figura 99. Puerto OBDII

Se ocupó un par de caimanes con cable en los extremos para conectarse en los pines y en la protoboard que contaba con la resistencia de 120ohm y a la cual se conectó el receptor CAN con el código subido, luego se abrió el "suich" del vehículo, hecho esto se esperaba que se mostraran las tramas de datos en el monitor como en el caso del clúster, sin embargo, la pantalla no mostraba nada.

Se probó con el *Jetta*, en el que se hizo la conexión al CAN bus mediante un conector ubicado de debajo del asiento del copiloto y se muestra en la figura 100. Al hacer la conexión y activar el receptor CAN se

mostraron en el monitor serial las diferentes tramas de mensajes enviadas en el bus por los diferentes módulos, como se observa en la figura 101.

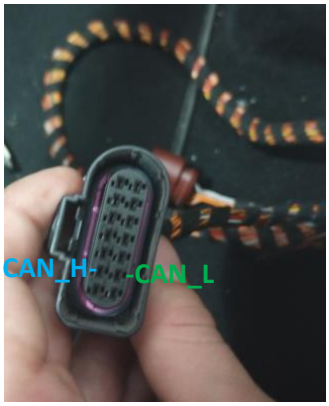

Figura 100. Conector con CAN en el Jetta Hybrid

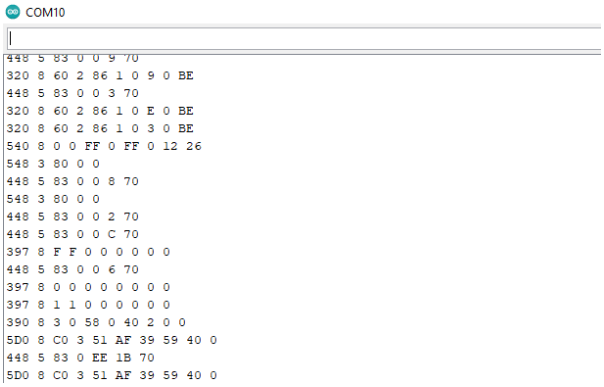

Figura 101. Tramas de mensaje recibidas del CAN-bus

Asegurando que se recibían mensajes provenientes del CAN bus del automóvil, se configuró el código para agregar máscaras y filtros, puesto que sabemos que el ID de respuesta será de entre 7E1 a 7E9. Para esto se agrega la función "*mcp2515.setConfigMode*", para configurar el controlador CAN, seguido de las funciones "*mcp2515.setFilterMask*" y "*mcp2515.setFilter*", las entradas de la función son las siguientes: (número de la máscara o filtro, trama normal (false) o extendida (true), máscara o filtro). Como se observa en la siguiente figura 102.

```
mcp2515.setConfigMode();
mcp2515.setFilterMask(MCP2515::MASK0, false, 0x7FF);      // Primer marcara
mcp2515.setFilterMask(MCP2515::MASK1, false, 0x7FF);      // Segunda marcara
mcp2515.setFilter(MCP2515::RXF0, false, 0x280);           // Filtro 1
  mcp2515.setFilter(MCP2515::RXF1, false, 0x0F6);       // Filtro 2
```

Figura 102. Asignar máscaras y filtros CAN

Se ocupó el logaritmo mencionado en el capítulo 3.4 para que los PID solicitas cambien cíclicamente y agregaron también las diferentes conversiones, usando los valores recibidos en los bytes, para así obtener la variable física del PID que se solicitó, los cuales son los que posteriormente se enviaran a la base de datos en la nube, a continuación se muestran las fórmulas que se ocuparon:

1. Temperatura del líquido de enfriamiento del motor:

$$TemperaturaM[C^o] = A - 40$$

2. Revoluciones por minuto (rpm):

$$RPM[rpm] = \frac{(256 * A) + B}{4}$$

3. Velocidad del vehículo:

$$V\left[\frac{km}{h}\right] = A$$

4. Temperatura de aire del colector de admisión:

$$TemperaturaCAd[C^o] = A - 40$$

5. Salida del sensor de flujo de aire MAF:

$$MAF[gr/seg] = \frac{(256 * A) + B}{100}$$

Habiendo agregado las fórmulas al código y probarlo en el vehículo, se obtuvo en el monitor serial la trama de datos con los diferentes PIDs seguido del valor obtenido aplicando las fórmulas. En la figura 103 se muestra la conexión al OBDII usando la placa ESP32 y la lectura de la trama recibida en el monitor serial y en la figura 104 se observa la captura del monitor, donde se observan las respuestas a los PIDs solicitados junto con el valor en decimal de los sensores.

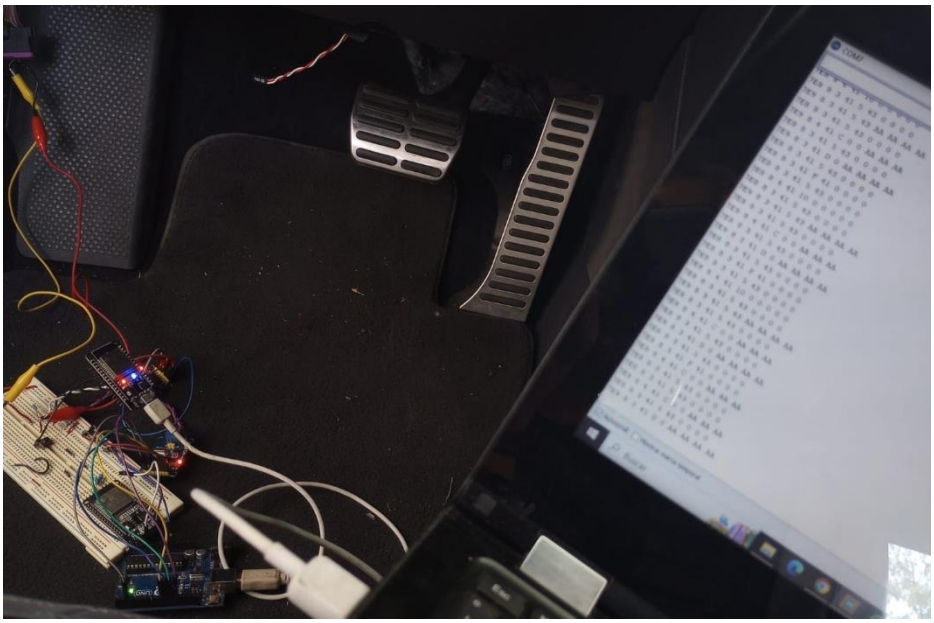

Figura 103. Conexión de los dispositivos al OBDII y los ratos recibidos

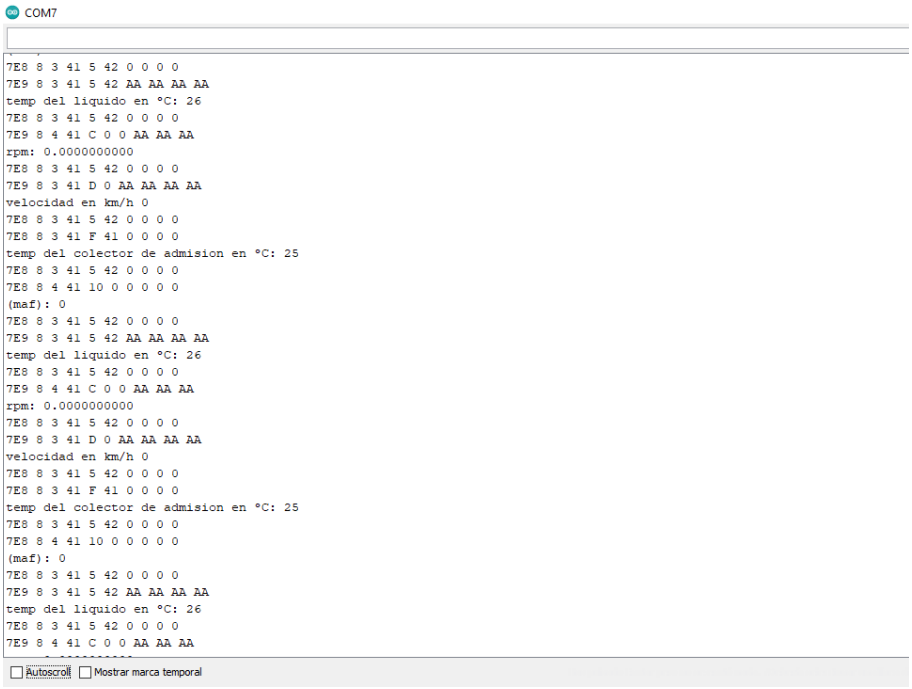

Figura 104. Tramas recibidas del OBDII y valor del sensor.

Se agregaron al código las líneas para enviar datos a la base de datos en tiempo real, como se vio en el capítulo 3, usando la librería de *realtime database*. En la base de datos se almacenan los valores de los sensores solicitados, cuyos valores se actualizan automáticamente cuando exista un cambio en el mismo. Como se muestra en la figura 105. Se restó un valor aleatorio entre 0 y 5 para ver que se actualizarán los datos subidos a la tabla, ya que se ocupó el auto sin encender el motor. El código completo se encuentra en el apartado de anexos D.

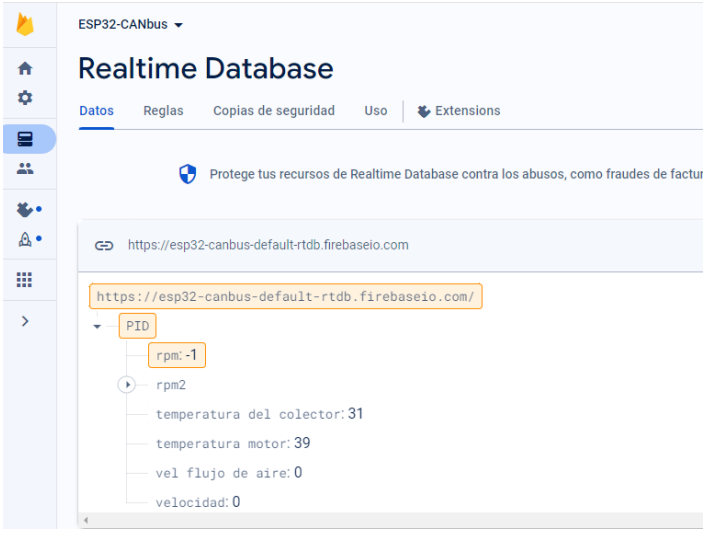

Figura 105. Datos obtenidos del auto almacenados en la nube

Capítulo 5

Resultados y Conclusiones

5.1 Resultados del proyecto

Los resultados obtenidos fueron satisfactorios en cuanto a la solicitud y adquisición de los datos del automóvil por medio del CAN-bus y el OBDII. Así como una base de datos que almacena la información en un servidor en la nube, la cual recibe la información en tiempo real. En la siguiente figura se muestra la comparación de los datos obtenidos entre la pantalla del monitor serial del microcontrolador y la base de datos.

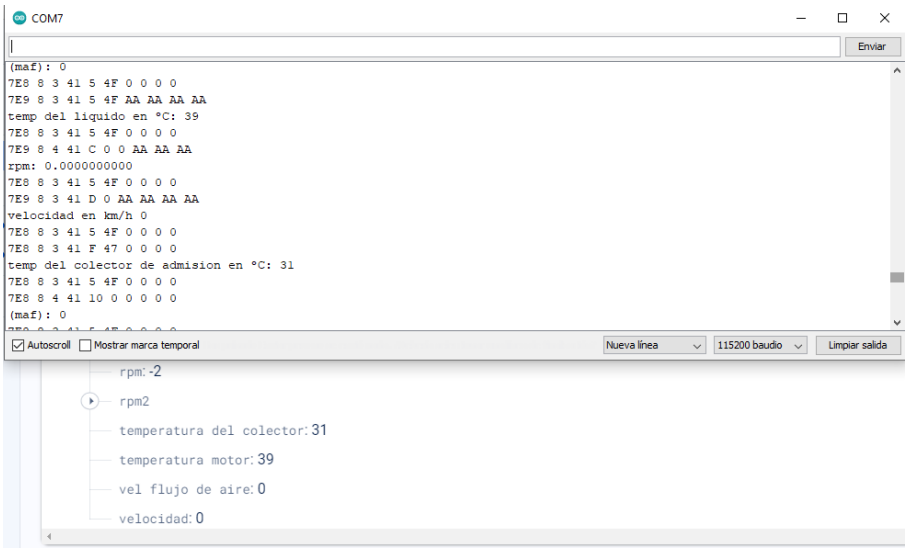

Figura 106. Comparación de los datos obtenidos del monitor seria con los que se muestran en la nube

En resumen, se logró hacer un sistema que permite adquirir, almacenar y visualizar la información solicitada en un almacenamiento en la nube, dentro de una base de datos en tiempo real, la cual se presenta como un árbol de nodos donde se puede agregar un nombre y un valor de cualquier tipo (booleano, entero, escritura, etc.), la estructura de la base de datos se amacena en formato Json, el cual se puede descargar. Además de las herramientas que ofrece *Firebase*, específicamente la acción de compartir la base de datos en algún sitio web o aplicación móvil, tanto para *Android* cono *IOs*. Sin embargo, en cuanto a la visualización de datos mediante una aplicación móvil no se obtuvieron resultados favorables, pues al realizar las pruebas no se logró establecer una comunicación entre *Firebase* y *Android Studio*.

5.2 Conclusiones

Se diseñó un sistema que solicita y recibe datos específicos del automóvil por medio de una conexión a la red CAN-bus y aplicando el formato de trama especificado en el protocolo OBDII para solicitar los datos de los sensores. Los datos obtenidos en este caso fueron los PID: 0C que corresponde a las revoluciones del motor, el 05 que corresponde a la temperatura del líquido del motor, el 0D a la velocidad, el 0F a la temperatura en el colector d admisión y el 10 a la velocidad de flujo de aire (sensor MAF), pero si se requiere, también se disponen de los PID que se muestran en el apartado C de los anexos, establecidos en la norma SAE. Para esto se ocupó un microcontrolador en conjunto con un controlador y transceptor CAN, se realizó un código que solicite repetidamente cada una de las tramas correspondientes a cada PID, como se dijo en el capítulo 3.4. Dicho código funciona tanto para Arduino UNO, como para la placa ESP32, pero al final se ocupó esta última por el módulo wifi que tiene integrado, el cual funcionó para enviar la información obtenida a una base de datos en tiempo real, la cual, se obtuvo y configuró de manera gratuita con el servidor Firebase de Google. Dicha información, en teoría, se tenía que compartir y visualizar en una aplicación móvil que se desarrolló y configuró en el software Android Studio.

Después de realizar el trabajo se puede concluir que se cumplió con los objetivos de extraer información del automóvil a través del CAN-bus y almacenarla en la nube, sin embargo, no se logró conseguir el objetivo de diseñar y sincronizar una aplicación móvil con Firebase para así visualizar la información de la base de datos en el celular.

Cabe resaltar que el proyecto se realizó de manera que resultara económico y sencillo, se usaron componentes comerciales que sean de fácil acceso, así como librerías y programas que sean accesibles a cualquier público, lo que da la posibilidad de continuar con el proyecto y trabajar en los muchos apartados en los que se puede mejorar.

Es importante mencionar las ventajas de tener la información almacenada en una base de datos en tiempo real, en especial para trabajar con proyectos IoV, además de las potencial desarrollo de proyectos con terceros, como en este caso de Google, que si se quisiera implementar de una forma comercial, da las ventajas para subir la aplicación en la tienda de “Play Store”, así como diversas herramientas de monitoreo del proyecto y en caso de ser necesario, pagar por todas las herramientas que se ofrecen. El proyecto, por lo tanto, queda para futuros trabajos y proyectos enfocados al IoV y el manejo de la información para propósitos de confort, seguridad e Infotainment.

Como experiencia personal al realizar el proyecto adquirí diferentes conocimientos relacionados a la industria automotriz y al IoV, como lo es el funcionamiento del protocolo CAN-bus, así como las diferentes normas y códigos establecidos para la solicitud de información, además de contemplar las diferentes posibilidades para crear un sistema similar al de este proyecto, existiendo diferentes dispositivos y códigos que permitan conectarse y subir información a la web. Un último comentario y recomendación es que se cuente con un buen equipo para trabajar, pues los softwares como *Android studio* con el que se te trabajó, alentan mucho el equipo, lo que dificultó el trabajo con el mismo. El proyecto queda pues para futuras mejoras como se menciona en el siguiente apartado.

5.3 Trabajos futuros

Una vez concluido el trabajo, es evidente el potencial de esta área en futuros proyectos y mejoras, pues actualmente se cuenta con una gran cantidad de herramientas y facilidades para este tipo de proyectos y sin embargo no se encuentran tantos proyectos relacionados, con excepción del escáner ELM327, que cumple y supera el proyecto en cuanto a adquisición de datos y precio, pero no se ha desarrollado para uso potencial en IoV y almacenamiento en la nube.

Algunas mejoras que se pueden realizar en el trabajo, serian el adaptar un conector OBDII para alimentar la placa y el módulo transceptor con la energía proporcionada por la batería del vehículo a la que nos da acceso el puerto OBDII, también es posible reducir el tamaño agregando solo los componentes necesarios del sistema a una PCB o del tipo. Por otro lado, en cuanto a la adquisición de datos, se tiene acceso a diferentes códigos establecidos por la norma SAE, que nos permite obtener diferentes datos e incluso saber los que se encuentran disponibles en el modelo del automóvil, por lo que se puede adquirir la información en tiempo real del sensor que esté disponible, según el propósito del proyecto.

En cuanto a lo relacionado al IoV, se cuentan con bastantes opciones para trabajar, tanto módulos para establecer una conexión a internet, diferentes servidores en la nube, así como tipos de bases de datos que se ofrecen, tanto SQL como no, todos los cuales se pueden encontrar de manera gratuita, con un costo el cual dependerá de las herramientas y enfoque al que vaya dirigido el proyecto. En este caso se usó *Google* puesto que es el más accesible y cuenta con herramientas y convenios con empresas y programas relacionados la creación de proyectos IoT, también cuenta almacenamiento en la nube, cuentas y desarrollo de aplicaciones, las cuales se encuentra de manera gratuita y aunque sea básico sirve para proyectos primerizos, pero que, al cambiar a un formato de pago brinda muchas facilidades para sacar al mercado y a la red diferentes proyectos informáticos. Aquí mismo se pueden realizar diferentes mejoras, ocupando más a fondo las herramientas que ofrece *Realtime Database*, como lo es, el manejo de diferentes usuarios, datos y autorizaciones que son necesarias para llevar un proyecto de este tipo al mercado. Así como la sincronización con el dispositivo conectado al CAN, el cual dependiendo del enfoque del proyecto, puede crear una estructura de datos más compleja o solicitar unos datos más específicos, asi como en la misma base de datos se pueden restringir y realizar acciones por medio de las reglas programables de la base de datos (*Realtime Database*). Otra cosa en la que se puede profundizar es en los diferentes convenios que tiene *Firebase* con diferentes empresas como el caso de *Android Studio* con el que se decidió trabajar.

En cuanto al desarrollo de la aplicación móvil usando *Android Studio* todavía hay mucha área por trabajar en este IDE, pues al realizar el trabajo por falta de tiempo y funcionamiento del equipo de trabajo, no se profundizó a detalle las diferentes acciones y funciones que se pueden utilizar para tener un mejor diseño de la interface, así como una mejor disposición y muestra de la información. También se pueden desarrollar aplicaciones para *Android Auto* lo que puede ser de utilidad para proyectos de este estilo. Finalmente se queda para un trabajo futuro el poder obtener y mostrar los datos de *Firebase* de una manera eficiente y agradable a cierto público al que vaya dirigido, para que si se requiere profundizar y difundir poder sacar la aplicación a la tienda de *Play Store*.

Bibliografía

[1] Agarwal, D., & Chowdhury, A. (2014). The basics automotive cluster device architecturesand aplications, part I.

[2] Anchapaxi, A. (2016). RECOLECCIÓN DE DATOS DEL SISTEMA OBD II DE UN AUTOMÓVIL USANDO UN DISPOSITIVO ANDROID. *[Tesis de licenciatura]. Escuela politécnica Nacional de Ecuador* .

[3] Birau, E. (2018).Adquisiscion de datos remota mediante el sistema OBDII y un microcontrolador. *[Tesis de Licensiatura]. Universidad Autonoma de Nuevo Leon*.

[4] Borges, E. (2019, Mrazo 17). *Infranetworking*. Retrieved from Servidor Base de Datos: https://blog.infranetworking.com/servidor-base-de-datos/

[5] *Codigosobd2.net*. (2023, Enero 25). Retrieved from Mejor Escaner Automotriz de 2021: Guía de Compra Definitiva. : https://codigosobd2.net/escaner-automotriz/

[6] Coryjfowler. (2020, Junio 24). *GitHub*. Retrieved from MCP_CAN Library: https://github.com/coryjfowler/MCP_CAN_lib

[7] *Developers*. (2023, 09 28). Retrieved from Introducción a Android Studio: https://developer.android.com/studio/intro?hl=es-419

[8] Dmitry. (2020, marzo 5). *GitHub*. Retrieved from Arduino MCP2515 CAN interface library: https://github.com/autowp/arduino-mcp2515

[9] Eduard, V. (2018). Development of a CAN-Wifi converter based on a ESP32. *[Tesis de maestría]. Universidad de Barcelona de ingeniería industrial.*

[10] *ElectroniscHub*. (2018, Agosto 23). Retrieved from Arduino MCP2515 CAN Bus Interface Tutorial: https://www.electronicshub.org/arduino-mcp2515-can-bus-tutorial/

[11] *Espressif*. (2023, Enero 20). Retrieved from Get started: https://docs.espressif.com/projects/esp-idf/en/release-v3.3/get-started/index.html#

[12] Firebase. (2023, Junio 17). Retrieved from https://firebase.google.com/?gad=1&gclid=CjwKCAjwgsqoBhBNEiwAwe5w04F0odP7YKRtoWL26EDFJ1Ts4KZdMeq5oLCXgAiRpY34kK56T4e6kRoCPhQQAvD_BwE&gclsrc=aw.ds&hl=es-419

[13] *Firebase*. (2023, 08 24). Retrieved from Cómo elegir tu base de datos: Cloud Firestore o Realtime Database: https://firebase.google.com/docs/database/rtdb-vs-firestore?hl=es-419

[14] *github*. (2023, enero 20). Retrieved from https://raw.githubusercontent.com/playelek/pinout-doit-32devkitv1/master/pinoutDOIT32devkitv1.png

[15] *GoogleCloud*. (2023, 08 12). Retrieved from Precios de Identity Platform: https://cloud.google.com/identity-platform/pricing?hl=es-419

[16] Herrera, J. (2022, 04 25). *Paradigma*. Retrieved from ¿Cómo crear una base de datos y conectarla a nuestra app con Firebase?: https://www.paradigmadigital.com/dev/crear-base-datos-firebase/

[17] Ibáñez, M. (2015). Integración de un sistema para l obtención de datos de vehículos automotores basado en los protocolos de CAN bus y OBDII. *[Tesis de licenciatura]. Universidad Nacional Autónoma de México.*

[18] *INEGI*. (2023, Septiembre 12). Retrieved from Economía y Sectores Productivos "Parque vehicular": https://www.inegi.org.mx/temas/vehiculos/

[19] Kovalev, T. (2021, Abril 1). *Github*. Retrieved from MCP2515 CAN interface library for using on esp32/esp8266: https://github.com/dedalqq/esp32-mcp2515

[20] Medina, A. (2018). "IoT aplicado al sector automotriz". *XIX Aniversario del Instituto Tecnológico Superior de RioVerde*, pp. 4-15.

[21] *Microchip Technology*. (2010, Marzo 26). Retrieved from MCP25215 Stand-Alome CAN Contoller With SPI Interface : http://www.microchip.com/downloads/en/DeviceDoc/21801e.pdf

[22] Mistry, S. (2018, Marzo 8). *GitHub*. Retrieved from An Arduino library for sending and receiving data using CAN bus.: https://github.com/sandeepmistry/arduino-CAN

[23] *National Instruments*. (2023, 08 23). Retrieved from Configuring PXI Systems Using MXI-Express: https://www.ni.com/en/shop/pxi/configuring-pxi-systems-using-mxi-express.html

[24] *National Instruments*. (2023, 03 16). Retrieved from Sistemas PXI: https://www.ni.com/es/shop/pxi.html

[25] Nkenyereye, L., & Jang, J. (2017). Integration of big data for querying can bus data from connected car. pp. 945-952.

[26] Obare, E. (2013, agosto 15). *Comprensión de las bases de datos en la nube: Amazon RDS, Google Cloud DB y más.* Retrieved from geekflare: https://geekflare.com/es/understanding-cloud-databases/

[27] Rodríguez, A., Vento, J., & Inouyr, R. (2018). Implementation of an OBD-II diagnostics tool over CAN-BUS with Arduino. *Sistemas & Telemática*, pp. 31-43.

[28] Signoretti, G., Silva, M., Dias, A., Silva, I., Silva, D., & Ferrari, P. (2018). Performance evaluation of an edge obd-ii device for industry 4.0. *ICCES*, pp. 273 - 278.

[29] *Silicon Labs*. (2023, Enero 22). Retrieved from CP210x USB to UART Bridge VCP Drivers: https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers

[30] Suárez, R. (2023, Enero 15). *Blog de Tecnologías*. Retrieved from Arduino: https://www3.gobiernodecanarias.org/medusa/ecoblog/rsuagued/arduino/

[31] Sudip, V., Amin, A., Moushumi, S., & Hovsepian, K. (2015). Estimating Drivers' Stress from GPS Traces. *IEEE*.

[32] Sun, S., Bi, J., Guillen, M., & Ana M., P. (2020). Assessing Driving Risk Using Internet of Vehicles Data: An Analysis Based on Generalized Linear Models. *MDPI Journal*, pp. 3-20.

[33] *Texas Instruments*. (2023, Enero 13). Retrieved from SN65HVD23x 3.3-V CAN Bus Transceivers: https://www.ti.com/lit/ds/symlink/sn65hvd230.pdf?ts=1676955248191&ref_url=https%253A%252F%252Fwww.google.com%252F

[34] Tolocka, H. (2023, Febrero 02). *Profe Tolocka*. Retrieved from Programando el ESP32 con el Arduino IDE: https://www.profetolocka.com.ar/2020/07/09/programando-el-esp-32-con-el-arduino-ide/

[35] *UM*. (2023, Febrero 1). Retrieved from Introducción al bus CAN: https://www.um.es/documents/4874468/19345367/ssee-da-t03-02.pdf/c4bc7d45-bb4a-4cbe-b9fb-7835ae37d2ac

[36] *Unit Electronics*. (2023, Enero 20). Retrieved from ESP32 DEVKIT V1 30 Pines USB-C/MicroUSB: https://uelectronics.com/producto/esp32-devkit-v1-30-pines-usb-c-microusb/

[37] Veliz, A., Riffo, S., Aguilar, M., González, C. (2022). RT Box card for studying the control communication impacts on microgrid performance and stability. *Elsevier*. pp 3-6.

[38] Voss, W. (2023, Febrero 5). *Copperhill Technologies*. Retrieved from ESP32 Triple CAN Bus Application Through Adding Two MCP2515 Ports: https://copperhilltech.com/blog/esp32-triple-can-bus-application-through-adding-two-mcp2515-ports/

[39] Wager, M. (2019, Mayo 8). *GitHub*. Retrieved from An Arduino CAN-Bus library for ESP32: https://github.com/miwagner/ESP32-Arduino-CAN

[40] Winkelhake, U. (2018). The digital transformation of the automotive industry: catalysts, roadmap, practice. *IEEE*, pp. 2-20.

[41] Yang, F., Wang, S., Li, J., & Sun, Q. (2014). An overview of internet of vehicles. *Chin comunication*, 1-14.

Anexos

A. Códigos de prueba con ID 280 para enviar al clúster mediante CAN-bus

Código usando la librería mcp2515.h

```
 #include <SPI.h>           //Libreria para usar pa comunicación SPI
#include <mcp2515.h>        //Libreria para usar comunicación CAN

struct can_frame canMsg;
  MCP2515 mcp2515(2);

void setup()
{
while (!Serial);
Serial.begin(115200);
SPI.begin();                //Begins SPI communication
mcp2515.reset();
mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ);    //configura el CAN a una
velocidad de 500KBPS
//y el reloj 8MHz
mcp2515.setNormalMode();
}

 void loop()
{
int digitalRPM = 0;
int RPM= analogRead(34);
digitalRPM = map(RPM,0,1023,0,4000);      //Valor RPM
canMsg.can_id = 0x280;                   //ID correspondiente a las RPM
canMsg.can_dlc = 8;                      //CAN con 8 bytes de datos
canMsg.data[0] = 0x00;
canMsg.data[1] = 0x00;
canMsg.data[2] = 0x00;
canMsg.data[3] = digitalRPM;             // Byte asignado al valor de las RPM
canMsg.data[4] = 0x00;
canMsg.data[5] = 0x00;
canMsg.data[6] = 0x00;
canMsg.data[7] = 0x00;
mcp2515.sendMessage(&canMsg);          //Envia el mensaje CAN
Serial.println("Messages sent");
delay(100);
}
```

Código para enviar rpm usando la librería mcp_can.h

```
#include <mcp_can.h> // Libreria para comunicarse con el bus
#include <SPI.h>  //Libreria que permite la comunicacion entre el MCP y Aduino

// Funciones creadas para ordenar el codigo
//void MCP2515_ISR(); // Interrupcion
void Enviar(); //Enviar mensajes
void Leer(); //Leer mensajes

// CA  N Variables
unsigned char flagRecv = 0; //Bandera de recepción del mensaj3
unsigned long int rxId;
// byte txData[] = {0x02,0x01,0x00,0x55,0x55,0x55,0x55,0x55};

// CAN RX Variables
unsigned char lon = 0; // longut de la trama de datos (0-8)
unsigned char Buf[8];    // "Bufer" almacenamiento del mensaje
char msgString[128];                        // Array to store serial string

// CAN Interrupt and Chip Select Pins
#define CAN0_INT 2                              // Inicialización del MCP en
el pin 2
MCP_CAN CAN0(10);                               // Pin para la comunicación
serial

void setup() {

  Serial.begin(115200);

  // Initialize MCP2515 running at 16MHz with a baudrate of 500kb/s and the
masks and filters disabled.
  if (CAN0.begin(MCP_STDEXT, CAN_500KBPS, MCP_8MHZ) == CAN_OK)
  {
    Serial.println("CAN bus iniciado correctamente!");

  } else {
    Serial.println("Error al iniciar CAN bus");
    //    attachInterrupt(0, MCP2515_ISR, FALLING);
    while (1);
  }
  /* while (CAN_OK != CAN.begin(CAN_500KBPS))
      { Serial.println("Fallo de inicio del CAN bus");
      delay(100); }
      Serial.println("Inicio del CAN bus");
      attachInterrupt(0, MCP2515_ISR, FALLING);
  */

  CAN0.setMode(MCP_NORMAL);                     // Configura el MCP en modo
normal para enviar y recibir mensajes
  pinMode(CAN0_INT, INPUT);                           // Configura el pin de
inicio como entrada
```

```
}

//void MCP2515_ISR()  //Activa el ISR
//{flagRecv = 1; }    //Durante la interrupción se alza la bander de recv


void loop() {

  // if(flagRecv)
  // {flagRecv = 0;
  // while (CAN_MSGAVAIL == CAN0.checkReceive())
  Enviar();
   delay(100);
  //Leer();
}

void Enviar() {
  Revoluciones();
  unsigned char datos[8] = {0, 0, 0, Revoluciones(), 0, 0, 0, 0};
  byte sndStat = CAN0.sendMsgBuf(0x280, 0, 8, datos);

  if (sndStat == CAN_OK) {
    Serial.print("Mensaje enviado: ");
  }
  else {
    Serial.println("Error Sending Message...");
  }

  Serial.println();

  delay(200);

 }

void Leer() {
  if (!digitalRead(CAN0_INT)) {
    CAN0.readMsgBuf(&rxId, &lon, Buf);
    Serial.print("Mensaje recivido: ");
    sprintf(msgString, "Standard ID: 0x%.3lX, DLC: %1d, Data: ", rxId, lon);
    Serial.print(msgString);

    for (byte i = 0; i < lon; i++) {
      sprintf(msgString, " 0x%.2X", Buf[i]);
      Serial.print(msgString);
    }
    Serial.println("");
  }
}

int Revoluciones()
{ const int analogp = 13;
  int potv = analogRead(analogp);
  const int valor = map(potv, 0, 1023, 0, 800);
  return (valor);
}
```

Código para enviar rpm usando la librería ESP32CAN.h

```
#include <Arduino.h>
#include <ESP32CAN.h>
#include <CAN_config.h>

CAN_device_t CAN_cfg;               // CAN Config
unsigned long previousMillis = 0;   // will store last time a CAN Message was
send
const int interval = 100;          // interval at which send CAN Messages
(milliseconds)
const int rx_queue_size = 10;       // Receive Queue size

void setup() {
  Serial.begin(115200);
  Serial.println("Basic Demo - ESP32-Arduino-CAN");
  CAN_cfg.speed = CAN_SPEED_500KBPS;
  CAN_cfg.tx_pin_id = GPIO_NUM_5;
  CAN_cfg.rx_pin_id = GPIO_NUM_4;
  CAN_cfg.rx_queue = xQueueCreate(rx_queue_size, sizeof(CAN_frame_t));
  // Init CAN Module
  ESP32Can.CANInit();
}

void loop() {

  CAN_frame_t rx_frame;

  unsigned long currentMillis = millis();

  // Receive next CAN frame from queue
  if (xQueueReceive(CAN_cfg.rx_queue, &rx_frame, 3 * portTICK_PERIOD_MS) ==
pdTRUE) {

    if (rx_frame.FIR.B.FF == CAN_frame_std) {
      printf("New standard frame");
    }
    else {
      printf("New extended frame");
    }

    if (rx_frame.FIR.B.RTR == CAN_RTR) {
      printf(" RTR from 0x%08X, DLC %d\r\n", rx_frame.MsgID,
 rx_frame.FIR.B.DLC);
    }
    else {
      printf(" from 0x%08X, DLC %d, Data ", rx_frame.MsgID,
 rx_frame.FIR.B.DLC);
      for (int i = 0; i < rx_frame.FIR.B.DLC; i++) {
        printf("0x%02X ", rx_frame.data.u8[i]);
      }
      printf("\n");
    }
  }
  int digitalRPM = 0;
```

```
int RPM= analogRead(34);
digitalRPM = map(RPM,0,1023,0,4000);
  // Send CAN Message
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
    CAN_frame_t tx_frame;
    tx_frame.FIR.B.FF = CAN_frame_std;
    tx_frame.MsgID = 0x280;
    tx_frame.FIR.B.DLC = 8;
    tx_frame.data.u8[0] = 0x00;
    tx_frame.data.u8[1] = 0x01;
    tx_frame.data.u8[2] = 0x02;
    tx_frame.data.u8[3] = digitalRPM;
    tx_frame.data.u8[4] = 0x04;
    tx_frame.data.u8[5] = 0x05;
    tx_frame.data.u8[6] = 0x06;
    tx_frame.data.u8[7] = 0x07;
    ESP32Can.CANWriteFrame(&tx_frame);
  }
```

B. Código para recibir mensajes del CAN-bus

```
#include <SPI.h>
#include <mcp2515.h>

struct can_frame canMsg;
struct can_frame canMsg1;      // Estructura de trama CAN
volatile bool interrupt = 0;
const int boton = 4;             // se define el Pin del pushbotton
int estadoBoton = 0;            // variable donde se guarda si el  botón ya fue
precionado
int valor = 0;                   // variable donde se guarda la entrada del
boton
int val_old = 0;

MCP2515 mcp2515(2);            // ESP32
// MCP2515 mcp2515(10);        // Arduino UNO

void setup() {
  while (!Serial);
  Serial.begin(115200);
  SPI.begin();

  mcp2515.reset();
  mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ);  // velocidad de trabajo del CAN


 Serial.println("ID  DLC   DATA");


  mcp2515.setNormalMode();
    attachInterrupt(digitalPinToInterrupt(21), irqHandler, FALLING);
}                                       //Función de interrupcion

void irqHandler() {
    interrupt = 1;
}

void loop() {

valor= digitalRead(boton); // lee el estado del Boton
if ((valor == HIGH) && (val_old == LOW)){
estadoBoton=1-estadoBoton;    // Se tiene un valor de 0 o 1 si el boton ya fue
presionado
delay(10);
}
val_old = valor; // valor del antiguo estado
if (estadoBoton==1){
interrupt = 1; // inicia la comunicación CAN
 // canMsg1.data[2] = 0x0c;
}
else{
interrupt = 0; // apagar la comunicación CAN
}
```

```
   if (interrupt) {
      interrupt = 0;
        uint8_t irq = mcp2515.getInterrupts();

        if (irq & MCP2515::CANINTF_RX0IF) {   // Recibe los mensajes CAN
            if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {

                // frame contains received from RXB0 message
            Serial.print(canMsg.can_id, HEX); // print ID
    Serial.print(" ");
    Serial.print(canMsg.can_dc, HEX); // print DLC
    Serial.print(" ");

    for (int i = 0; i<canMsg.can_dlc; i++)  {  // print the data
      Serial.print(canMsg.data[i],HEX);
      Serial.print(" ");
    } }
        Serial.println();
        }

        if (irq & MCP2515::CANINTF_RX1IF) {
            if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {
                // frame contains received from RXB1 message
            Serial.print(canMsg.can_id, HEX); // print ID
    Serial.print(" ");
    Serial.print(canMsg.can_dc, HEX); // print DLC
    Serial.print(" ");

    for (int i = 0; i<canMsg.can_dlc; i++)  {  // print the data
      Serial.print(canMsg.data[i],HEX);
      Serial.print(" ");
    }}
       Serial.println();
        }
delay(50);

}
}
```

C. Tabla de códigos PID

PID (hex)	Bytes de respuesta	Descripción	Valor mínimo	Valor máximo	Unidad	Fórmula
00	4	PIDs implementados [01 - 20]				Cada bit indica si los siguientes 32 PID están implementados (1) o no (0): [A7..D0] == [PID 01..20]
01	4	Estado de los monitores de diagnóstico desde que se borraron los códigos de fallas DTC; incluye el estado de la luz indicadora de fallas, MIL, y la cantidad de códigos de fallas DTC				Codificación en bits
02	2	Almacena los códigos de fallas de diagnóstico DTC de un evento				
03	2	Estado del sistema de combustible				Codificación en bits
04	1	Carga calculada del motor	0	100	%	(==)
05	1	Temperatura del líquido de enfriamiento del motor	-40	215	°C	
06	1	Ajuste de combustible a corto	-100	99.2	%	

		plazo—Banco 1	(Reducción de combustible: muy rico)	(Aumento de combustible: muy magro)		(or)
07	1	Ajuste de combustible a largo plazo—Banco 1				
08	1	Ajuste de combustible a corto plazo—Banco 2				
09	1	Ajuste de combustible a largo plazo—Banco 2				
0A	1	Presión del combustible	0	765	kPa	
0B	1	Presión absoluta del colector de admisión	0	255	kPa	
0C	2	RPM del motor	0	16,383.75	rpm	
0D	1	Velocidad del vehículo	0	255	km/h	
0E	1	Avance del tiempo	-64	63.5	° antes TDC	
0F	1	Temperatura del aire del colector de admisión	-40	215	°C	
10	2	Velocidad del flujo del aire MAF	0	655.35	gr/sec	

11	1	Posición del acelerador	0	100	%	
12	1	Estado del aire secundario controlado				Codificación en bits
13	1	Presencia de sensores de oxígeno (en 2 bancos)				[A0..A3] == Banco 1, sensores 1-4. [A4..A7] == Banco 2...
14	2	Sensor de oxígeno 1 A: Voltaje B: Ajuste de combustible a corto plazo	0 -100	1.275 99.2	voltios %	(si B==FF, entonces el sensor no se usa en el cálculo del ajuste)
15	2	Sensor de oxígeno 2 A: Voltaje B: Ajuste de combustible a corto plazo				
16	2	Sensor de oxígeno 3 A: Voltaje B: Ajuste de combustible a corto plazo				
17	2	Sensor de oxígeno A: Voltaje B: Ajuste de combustible a corto plazo				
18	2	Sensor de oxígeno 5 A: Voltaje B: Ajuste de combustible a corto plazo				
19	2	Sensor de oxígeno 6 A: Voltaje B: Ajuste de combustible a corto plazo				

1A	2	Sensor de oxígeno 7 A: Voltaje B: Ajuste de combustible a corto plazo				
1B	2	Sensor de oxígeno 8 A: Voltaje B: Ajuste de combustible a corto plazo				
1C	1	Estándar OBD implementado en este vehículo				Codificación en bits
1D	1	Sensores de oxígenos presentes en el banco 4				Similar a PID 13, pero [A0..A7] == [B1S1, B1S2, B2S1, B2S2, B3S1, B3S2, B4S1, B4S2]
1E	1	Estado de las entradas auxiliares				A0 == Estado de **P**ower **T**ake **O**ff, **PTO** (1 == activo) [A1..A7] sin uso
1F	2	Tiempo desde que se puso en marcha el motor	0	65,535	sec	
20	4	PID implementados [21 - 40]				Cada bit indica si los siguientes 32 PID están implementados (1) o no (0): [A7..D0] == [PID 21..40]
21	2	Distancia recorrida con la luz indicadora de falla (**M**alfunction **I**ndicator **L**amp, **MIL**) encendida	0	65,535	km	
22	2	Presión del tren de combustible, relativa al colector de vacío	0	5177.265	kPa	

23	2	Presión del medidor del tren de combustible (Diesel o inyección directa de gasolina)	0	655,350	kPa	
24	4	Sensor de oxígeno 1 AB: Relación equivalente de combustible - aire CD: Voltaje	0 0	< 2 < 8	prop. V	
25	4	Sensor de oxígeno 2 AB: Relación equivalente de combustible - aire CD: Voltaje				
26	4	Sensor de oxígeno 3 AB: Relación equivalente de combustible - aire CD: Voltaje				
27	4	Sensor de oxígeno 4 AB: Relación equivalente de combustible - aire CD: Voltaje				
28	4	Sensor de oxígeno 5 AB: Relación equivalente de combustible - aire CD: Voltaje				
29	4	Sensor de oxígeno 6 AB: Relación equivalente de combustible - aire CD: Voltaje				
2A	4	Sensor de oxígeno 7 AB: Relación equivalente de combustible - aire CD: Voltaje				
2B	4	Sensor de oxígeno 8 AB: Relación equivalente de combustible - aire CD: Voltaje				

2C	1	EGR comandado	0	100	%	
2D	1	falla EGR	-100	99.2	%	
2E	1	Purga evaporativa comandada	0	100	%	
2F	1	Nivel de entrada del tanque de combustible	0	100	%	
30	1	Cantidad de calentamientos desde que se borraron los fallas	0	255	cuenta	
31	2	Distancia recorrida desde que se borraron los fallas	0	65,535	km	
32	2	Presión de vapor del sistema evaporativo	-8,192	8191.75	Pa	(AB es número binario en complemento de dos con signo)
33	1	Presión barométrica absoluta	0	255	kPa	
34	4	Sensor de oxígeno 1 AB: Relación equivalente de combustible - aire CD: Actual	0 -128	< 2 <128	prop. mA	or
35	4	Sensor de oxígeno 2 AB: Relación equivalente de combustible - aire CD: Actual				
36	4	Sensor de oxígeno 3 AB: Relación equivalente de combustible - aire CD: Actual				
37	4	Sensor de oxígeno 4				

		AB: Relación equivalente de combustible - aire CD: Actual				
38	4	Sensor de oxígeno 5 AB: Relación equivalente de combustible - aire CD: Actual				
39	4	Sensor de oxígeno 6 AB: Relación equivalente de combustible - aire CD: Actual				
3A	4	Sensor de oxígeno 7 AB: Relación equivalente de combustible - aire CD: Actual				
3B	4	Sensor de oxígeno 8 AB: Relación equivalente de combustible - aire CD: Actual				
3C	2	Temperatura del catalizador: Banco 1, Sensor 1	-40	6,513.5	°C	
3D	2	Temperatura del catalizador: Banco 1, Sensor 1				
3E	2	Temperatura del catalizador: Banco 1, Sensor 2				
3F	2	Temperatura del catalizador: Banco 2, Sensor 2				
40	4	PID implementados [41 - 60]				Cada bit indica si los siguientes 32 PID están implementados (1) o no (0): [A7..D0] == [PID 41..60]

41	4	Estado de los monitores en este ciclo de manejo				Codificación en bits
42	2	Voltaje del módulo de control	0	65.535	V	
43	2	Valor absoluto de carga	0	25,700	%	
44	2	Relación equivaliente comandada de combustible - aire	0	< 2	prop.	
45	1	Posición relativa del acelerador	0	100	%	
46	1	Temperatura del aire ambiental	-40	215	°C	
47	1	Posición absoluta del acelerador B	0	100	%	
48	1	Posición absoluta del acelerador C				
49	1	Posición del pedal acelerador D				
4A	1	Posición del pedal acelerador E				
4B	1	Posición del pedal acelerador F				
4C	1	Actuador comandando del acelerador				
4D	2	Tiempo transcurrido con MIL encendido	0	65,535	min	
4E	2	Tiempo transcurrido desde que se borraron los códigops de fallas				
4F	4	Valor máximo de la relación de	0, 0, 0, 0	255,	prop.,	A, B, C, D*10

		equivalencia de combustible - aire, voltaje del sensor de oxígenos, corriente del sensor de oxígenos y presión absoluta del colector de entrada		255, 255, 2550	V, mA, kPa	
50	4	Valor máximo de la velocidad de flujo de aire del sensor de flujo de aire masivo	0	2550	g/s	A*10, B, C, y D están reservados para uso futuro
51	1	Tipo de combustible				Ver tabla
52	1	Porcentaje de combustible Ethanol	0	100	%	
53	2	Presión absoluta del vapor del sistema de evaporación	0	327.675	kPa	
54	2	Presión del vapor del sistema de evaporación	-32,767	32,768	Pa	((A*256)+B)-32767
55	2	Ajuste del sensor de oxígeno secundario de plazo corto. A: banco 1. B: banco 3	-100	99.2	%	
56	2	Ajuste del sensor de oxígeno secundario de plazo largo. A: banco 1. B: banco 3				
57	2	Ajuste del sensor de oxígeno secundario de plazo corto. A: banco 2. B: banco 4				
58	2	Ajuste del sensor de oxígeno secundario de plazo largo. A: banco 2. B: banco 4				
59	2	Presión absoluta del tren de combustible	0	655,350	kPa	
5A	1	Posición relativa del pedal del acelerador	0	100	%	

5B	1	Tiempo de vida del banco de baterías híbridas	0	100	%	
5C	1	Temperatura del aceite del motor	-40	210	°C	
5D	2	Sincronización de la inyección de combustible	-210.00	301.992	°	
5E	2	Velocidad del combustible del motor	0	3276.75	L/h	
5F	1	Requisitos de emisiones para los que el vehículo fue diseñado				Codificación en bits
60	4	PID implementados [61 - 80]				Cada bit indica si los siguientes 32 PID están implementados (1) o no (0): [A7..D0] == [PID 61..80]
61	1	Porcentaje de torque solicitado por el conductor	-125	125	%	A-125
62	1	Porcentaje de torque actual del motor	-125	125	%	A-125
63	2	Torque de referencia del motor	0	65,535	Nm	256A+B
64	5	Datos del porcentaje de torque del motor	-125	125	%	A-125 Ocioso B-125 Motor punto 1 C-125 Motor punto 2 D-125 Motor punto 3 E-125 Motor punto 4
65	2	Entrada / salida auxiliar implementada				Codificación en bits

66	5	Sensor de flujo de aire masivo				
67	3	Temperatura del enfriador del motor				
68	7	Sensor de temperatura de aire de entrada				
69	7	EGR comandado y falla de EGR				
6A	5	Control comandado del flujo de aire de entrada de Diesel y posición relativa de la entrada del flujo de aire				
6B	5	Temperatura de recirculación del gas del escape				
6C	5	Control comandado del actuador del acelerador y posición relativa del acelerador				
6D	6	Sistema de control de presión del combustible				
6E	5	Sistema de control de presión de inyección				
6F	3	Presión de entrada del compresor del turbocargador				
70	9	Control de presión de aumento				
71	5	Control del turbo de geometría variable (**V**ariable **G**eometry **T**urbo, **VG T**)				
72	5	Control de la compuerta de desperdicio				
73	5	Presión del escape				

74	5	RPM del turbocargador				
75	7	Temperatura del turbocargador				
76	7	Temperatura del turbocargador				
77	5	Temperatura del enfriador del aire de carga (**C**harge **A**ir **C**ooler **T**emperature, **CACT**)				
78	9	Temperatura del gas del escape (**E**xhaust **G**as **T**emperature, **EGT**) Banco 1				PID especial
79	9	Temperatura del gas del escape (**E**xhaust **G**as **T**emperature, **EGT**) Banco 2				PID especial
7A	7	Filtro de partículas Diesel (**D**iesel **P**articulate **F**ilter, **DPF**)				
7B	7	Filtro de partículas Diesel (**D**iesel **P**articulate **F**ilter, **DPF**)				
7C	9	Temperatura del filtro de partículas Diesel (**D**iesel **P**articulate **F**ilter, **DPF**)				
7D	1	Estado del área de control NOx NTE				
7E	1	Estado del área de control PM NTE				
7F	13	Tiempo que el motor ha estado en marcha				
80	4	PID implementados [81 - A0]				Cada bit indica si los siguientes 32 PID están implementados (1)

						o no (0): [A7..D0] == [PID 81..A0]
81	21	Tiempo de marcha del motor para el dispositivo auxiliar de control de emisiones (**A**uxiliary **E**missions **C**ontrol **D**evice, **AECD**)				
82	21	Tiempo de marcha del motor para el dispositivo auxiliar de control de emisiones (**A**uxiliary **E**missions **C**ontrol **D**evice, **AECD**)				
83	5	Sensor de NOx				
84		Temperatura de superficie del colector				
85		Sistema reactivo NOx				
86		Sensor de partículos (**P**articular **M**atter, **PM**)				
87		Presión absoluta del colector de admisión				
A0	4	PID implementados [A1 - C0]				Cada bit indica si los siguientes 32 PID están implementados (1) o no (0): [A7..D0] == [PID A1..C0]
C0	4	PID implemantados [C1 - E0]				Cada bit indica si los siguientes 32 PID están implementados (1) o no (0): [A7..D0] == [PID C1..E0]
C3	?	?	?	?	?	Muestra datos, incluye ID de la

						condición del motor y su velocidad
C4	?	?	?	?	?	B5: Solicitud de poner el motor en estado ocioso B6: Solicitud de detener el motor

D. Código completo para adquirir y enviar datos del automóvil a la base de datos en Firebase

```
#include <SPI.h>            //Librería para usar SPI
#include <mcp2515.h>        //Librería para la comunicaión CAN
#include <Arduino.h>
#include <WiFi.h>
#include <FirebaseESP32.h>                                  // librerías
necesarias para usar el modulo wi-fi inorporado en el ESP32, Arduino y
comunicación con FireBase
#include <addons/TokenHelper.h>                              //libreria para
Proporcionar la información del proceso de generación del token.
#include <addons/RTDBHelper.h>                               //libreria para
Proporcionar información de impresión de carga útil RTDB y otras funciones
auxiliares.

#define WIFI_SSID "BUAP_Estudiantes"                       //"Totalplay-
80A5_EXT_plus"
#define WIFI_PASSWORD "f85ac21de4"                      //"80A5F733Mn38yzXp"
                  //se definen las credeciales para la cenexion wifi. Nombre
y contraseña
#define API_KEY "AIzaSyA7nRZppsJCu1tMV7UmT1_WFTnfca5vY_E"   //Se define la
llave API del usuario
#define DATABASE_URL "esp32-canbus-default-rtdb.firebaseio.com"       //Se
define la URL de la base de datos con la siguiente estructura

                                                                  //<databaseName>.
firebaseio.com or <databaseName>.<region>.firebasedatabase.app
#define USER_EMAIL "ramonigh.15@gmail.com"
#define USER_PASSWORD "ESPBUAP123"

FirebaseData fbdo;
FirebaseAuth auth;
FirebaseConfig config;                                      //Declaramos los
"Firebase Data object"

unsigned long sendDataPrevMillis = 0;
struct can_frame canMsg;
struct can_frame canMsg1;   //Se defime la estructura de los mensajes
                            //como una trama CAN-bus
volatile int interrupt = 0;      //se define la variable de interrupcion
const int boton = 4;            // se difine el Pin del pushbotton
int estadoBoton = 0;           // variable donde se guarda si el  boton ya fue
precionado
int valor = 0;                  // varibla donde se guarda la entrada del
boton
int val_old = 0;

     MCP2515 mcp2515(2);                  // CS del MCP2515 al pin 10

void setup() {
  while (!Serial);
  Serial.begin(115200);
   SPI.begin();
```

```
WiFi.begin(WIFI_SSID, WIFI_PASSWORD);                 //Iniciamos el modulo
WiFi
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED)
  {
    Serial.print(".");
    delay(300);
  }
  Serial.println();
  Serial.print("Connected with IP: ");
  Serial.println(WiFi.localIP());
  Serial.println();

  Serial.printf("Firebase Client v%s\n\n", FIREBASE_CLIENT_VERSION);

  config.api_key = API_KEY;
  auth.user.email = USER_EMAIL;
  auth.user.password = USER_PASSWORD;
  config.database_url = DATABASE_URL;                          //Asignamos las
variables ya definidas a la configuracion estalecida en el coigo de la
librerías de FireBaseESP32
  config.token_status_callback = tokenStatusCallback;         //configuracion
para llamar el estado del token

  //Asegúrese de que el espacio libre del dispositivo no sea inferior a 80 k
para ESP32 y 10 k para ESP8266.
  // de lo contrario la conexión SSL fallará

 ////////////////////////////////////////////////////////////////////////////
/////////////////

  Firebase.begin(&config, &auth);                        //Inicia Firebase
  // Comente o pase un valor falso cuando la reconexión WiFi se controle
mediante su código o biblioteca de terceros
  Firebase.reconnectWiFi(true);
  Firebase.setDoubleDigits(5);                          //digitos despues del
punto en variables tipo doble


  mcp2515.reset();
  mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ);      //configura el CAN a
velocidad de 500KBPS
                                                  //y la frecuencua del
ozcilador en 8MHz
  Serial.println("------- Receptor  CAN ----------");
  Serial.println("ID  DLC   DATA");
pinMode(boton, INPUT);                              // de define "boton" como
entrada

//  mcp2515.setConfigMode();
//  mcp2515.setFilterMask(MCP2515::MASK0, false, 0x7FF);      // Primer
marcara
//  mcp2515.setFilterMask(MCP2515::MASK1, false, 0x7FF);      // Segunda
marcara
//  mcp2515.setFilter(MCP2515::RXF0, false, 0x280);           // Filtro 1
//    mcp2515.setFilter(MCP2515::RXF1, false, 0x0F6);       // Filtro 2
```

```
//
  mcp2515.setNormalMode();
    attachInterrupt(digitalPinToInterrupt(21), irqHandler, FALLING);
  //Funcción para activar la interrupción
    //El primer termino es el pin de interrupción,
    // Función de la interrupción
    // Activa la funcion de la interrupcion, cuando hay un cambio de alto a
bajo
}

void irqHandler() {                                  // Funcion implemenatada por
la interrupcion
    interrupt=1;                                   // variable interrucion a
positivo
  }

void loop() {

valor= digitalRead(boton); // lee el estado del Boton
if ((valor == HIGH) && (val_old == LOW)){
estadoBoton=1-estadoBoton;    // Se tiene un valor de 0 o 1 si el boton ya fue
presionado
delay(10);
}
val_old = valor; // valor del antiguo estado
if (estadoBoton==1){
interrupt = 1; // inicia la comunicación CAN
 // canMsg1.data[2] = 0x0c;
}
else{
interrupt = 0; // apagar la comunicación CAN
}

canMsg1.can_id = 0x7DF;
canMsg1.can_dlc = 8;
canMsg1.data[0] = 0x02;
canMsg1.data[1] = 0x01;
canMsg1.data[2] = 0x05;
canMsg1.data[3] = 0x00;
canMsg1.data[4] = 0x00;
canMsg1.data[5] = 0x00;
canMsg1.data[6] = 0x00;
canMsg1.data[7] = 0x00;

  if (interrupt) {              // si interrupcion es verdadero se leen los
mensajes del CAN
        interrupt = 0;
        uint8_t irq = mcp2515.getInterrupts();

if (irq & MCP2515::CANINTF_RX0IF) {
            if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {      //
Recibe los mensajes de RXB0

            Serial.print(canMsg.can_id, HEX);                          //
imprime ID
    Serial.print(" ");
```

```
    Serial.print(canMsg.can_dlc, HEX);                                    //
imprime DLC
    Serial.print(" ");

    for (int i = 0; i<canMsg.can_dlc; i++)  {                              //
imprime los datos
      Serial.print(canMsg.data[i],HEX);
      Serial.print(" ");
    }
    Serial.println();
        }
            }

 if (irq & MCP2515::CANINTF_RX1IF) {
            if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {       //
Recibe los ensajes de RXB1

            Serial.print(canMsg.can_id, HEX);
    Serial.print(" ");
    Serial.print(canMsg.can_dlc, HEX);
    Serial.print(" ");

    for (int i = 0; i<canMsg.can_dlc; i++)  {
      Serial.print(canMsg.data[i],HEX);
      Serial.print(" ");
    }   }
       Serial.println();
        }
delay(100);
mcp2515.sendMessage(&canMsg1);

   switch (canMsg.data[2]) {
   case 0x05:
    uint16_t Temp;
    Temp = canMsg.data[3]-40;
    Serial.print("temp del liquido en °C: ");
    Serial.println(Temp, DEC);
       if (Firebase.ready() )
  {
    Firebase.setFloat(fbdo, F("/PID/temperatura motor"), (Temp));
  }

    delay(20);
     canMsg1.data[2] = 0x0C;      //PID RPM
     mcp2515.sendMessage(&canMsg1);
     delay(50);
   break;
   case 0x0C:
   float RPM;
    RPM = ((256 * canMsg.data[3]) + canMsg.data[4]) / 4;
     Serial.print("rpm: ");
          Serial.println(RPM, DEC);
    if (Firebase.ready())
  {
    Firebase.setFloat(fbdo, F("/PID/rpm"), (RPM - random(5)));
    delay (100);
```

```
    //Firebase.push(fbdo, F("/PID/rpm2"), (RPM ));
  }

        delay(20);
    canMsg1.data[2] = 0x0D;      //PID velocidad
    mcp2515.sendMessage(&canMsg1);
    delay(50);
   break;
   case 0x0D:
      int Vel;
    Vel = canMsg.data[3];
    Serial.print("velocidad en km/h ");
    Serial.println(Vel);
        if (Firebase.ready())
  {
    Firebase.setFloat(fbdo, F("/PID/velocidad"), (Vel));
  }
    delay(20);
     canMsg1.data[2] = 0x0F;    //PID temperatura del colector de admisión
     mcp2515.sendMessage(&canMsg1);
     delay(50);
   break;
   case 0x0F:
       uint16_t Temp1;
    Temp1 = canMsg.data[3]-40;
    Serial.print("temp del colector de admision en °C: ");
    Serial.println(Temp1, DEC);
        if (Firebase.ready())
  {
    Firebase.setFloat(fbdo, F("/PID/temperatura del colector"), (Temp1));
  }
    delay(20);
     canMsg1.data[2] = 0x10;    //PID Presión del combustible
     mcp2515.sendMessage(&canMsg1);
     delay(50);
   break;
   case 0x10:
     uint16_t Pre;
    Pre = 3 * canMsg.data[3];
    Serial.print("(maf): ");
          Serial.println(Pre, DEC);
        if (Firebase.ready())
  {
    Firebase.setFloat(fbdo, F("/PID/vel flujo de aire"), (Pre));
  }
        delay(20);
    canMsg1.data[2] = 0x05;    //PID temperatura del motor
    mcp2515.sendMessage(&canMsg1);
    delay(50);
   break;
   }
  }
}
```