Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Electrónica
Maestría en Ingeniería Electrónica, opción Instrumentación Electrónica

Tesis para obtener el grado de:
Maestro en Ingeniería Electrónica

Sistema de recolección de datos de un vehículo en tiempo real a través del CAN bus para almacenarlos en la nube utilizando tecnología 4G/LTE

Presenta:
Saul Luna Minor*

Asesor:
M.C. Nicolás Quiroz Hernández VoBo 6/12/22

Co-Asesor:
Dr. Roberto Ambrosio Lázaro

*Becario CONACyT

Puebla, Pue., 6 de diciembre de 2022

Agradecimientos

Agradezco al Consejo Nacional de Ciencias y Tecnología (*CONACyT*) por el apoyo económico que ha hecho posible la conclusión de mis estudios y del trabajo presentado. También agradezco a la Benemérita Universidad Autónoma de Puebla (*BUAP*), a la Facultad de Ciencias de la Electrónica (*FCE*) y a la Maestría en Ingeniería en Electrónica por prestar todas las facilidades durante mi formación académica.

Quiero expresar mi mayor gratitud al director de tesis el M.C. Nicolás Quiroz Hernández, por su enorme ayuda desde la licenciatura, más que un profesor, lo considero como un amigo. Tanto su apoyo y confianza han sido un aporte inconmensurable, no solamente para el desarrollo de esta tesis, sino también en mi formación. Le agradezco también el facilitar los medios necesarios para llevar a cabo las actividades propuestas de esta tesis y sobre ponerse a las dificultades que provocó esta pandemia. Muchas gracias profesor.

También quiero agradecer a mi co-asesor el Dr. Roberto Ambrosio Lázaro, quien contribuyó con su experiencia en el área automotriz. Sus conocimientos han alimentado y mejorado los míos, muchas gracias por su orientación y apoyo. De igual forma, agradezco a mis sinodales, M.C. Ana María Rodríguez Domínguez, M.C. Ricardo Álvarez González y M.C. Rodrigo Lucio Maya Ramírez por sus observaciones que sirvieron para mejorar este trabajo de tesis. También se reconoce el enorme esfuerzo de la M.C Ana María que gestiono la documentación necesaria para el ingreso a la maestría, muchas gracias.

Se agradece a la empresa *INTESC* y en especial a su director el M.C. Carlos García Lucero, por abrirme las puertas para realizar mi estancia, además de ayudarme con su valioso conocimiento y experiencia en el ámbito del desarrollo de *PCBs*.

Gracias a mis padres, quienes me brindaron su asistencia continua y determinante, por el esfuerzo que realizan a diario y que sin ustedes no fueran posibles mis estudios. Sus consejos y experiencias me han ayudado a crecer como persona. Muchas gracias padres por su apoyo y cariño.

Dedicatoria

A mis padres por su constante apoyo para culminar mis estudios, ya que su mayor anhelo era ver a su hijo terminar una carrera universitaria.

A toda mi familia por su poco o mucho apoyo que me brindaron para llegar hasta donde estoy hoy en día.

A mi abuelo Florentino Minor, quien siempre quiso ver a su nieto concluir una licenciatura, sin embargo la muerte le aconteció antes de tiempo.

A mi abuela Reyes Minor, cuyas enseñanzas y consejos me sirvieron a lo largo de mi vida. Ella carece de las facultades para leer, sin embargo alguien le recitara esta dedicatoria para que sepa lo mucho que la estimo.

Dedico esta tesis a mis amigos Jesus Cuvas Limon, Martin Almeida, Jose Juan Mendoza y Cesar Cosme Telis, que me ayudaron desde la licenciatura, por siempre chakboys.

Dedico esta tesis al profesor Nicolás Quiroz, ya que durante mis estudios de licenciatura y maestría, estuvo conmigo brindándome su conocimiento y experiencia. A pesar de su apretada agenda, él encontraba la manera de darme un poco de su valioso tiempo para corregir y mejorar en el trabajo de tesis. Muchas gracias !!!

En memoria del compañero Luis Carlos, que lamentablemente murió durante su trayecto en la maestría, en mi memoria yace el recuerdo de lo mucho que se esforzaba para entender los conceptos de electrónica, ya que él era físico matemático. Descanse en paz.

Y por último, pero no menos importante, a todos mis compañeros de la Maestría, especialmente a Luis Efrain, Adriana Palacios, Jose Jair Medina, Alejandro Mendoza y mi tocayo Saúl Báez, mis mejores deseos para ellos.

Índice general

Índice de figuras

Índice de tablas

Lista de códigos

Abreviaturas

ABS	*Antilock Brake System*, Sistema Antibloqueo de Frenos
ARM	*Advanced RISC Machine*
ASC	*Automatic Stability Control*, Control Automatico de Estabilidad
bps	bits por segundo
CAN	*Controller Area Network*, Red de Area de Controlador
CGW	*Central Gateway*, Gateway Central
CMSIS	*Cortex Microcontroller Software Interface* Standard
CPU	*Central Process Unit*, Unidad Central de Proceso
CRC	*Cyclic Redundancy Check*, Verificación de Reduncancia Cíclica
ECU	*Electronic Control Unit*, Unidad Electronica de Control
ECM	*Engine Control Module*, Módulo de control de motor
GPIO	*General Purpuse Input/Output*, Entrada/Salida de Proposito General
HAL	*Hardware Abstraction Layer*, Capas de Abstracción de Hardware
ID	Identificador CAN bus
IDE	*Integrated Development Enviroment*, Entorno de Desarrollo Integrado
IVN	*In-Vehicle Network*, Red Intra-vehicular
kbps	Kilobits por segundo
LAN	*Local Area Network*, Red de Area Local
LIN	*Local Interconect Network*, Red Local de Interconexión
MAF	*Mass Air Flow*, sensor de flujo de aire
Mbps	Megabits por segundo
MOST	*Media Oriented System Transport*, Sistema de Transporte Orientado a Medios
NRZ	*Non-Return-to-Zero*, No Retorno a Cero
OBD	*On Board Diagnostics*, Diagnostico a Bordo
OSI	*Open System Interconnection*, Interconexión de Sistema Abierto
PCB	*Printed Circuit Boart*, Tarjeta de Circuito Impreso
PID	*Protected ID*, ID Protegido
RTOS	*Real Time Operating System*, Sistema Operativo de Tiempo Real
RTR	*Remote Transmision Request*, Solicitud de Transmisiín Remota
SADO	Sistema de adquisición de datos del OBD-II
SCI	*Serial Communication Interface*, Interfaz de Comunicación Serial
SMD	*Surface Mounted Device*, Dispositivo de Montaje Superficial
TCM	*Transmission Control Module*, Módulo de Control de Transmisión
TDMA	*Time Division Multiple Access*, División de Tiempo de Acceso Múltiple
UART	*Universal Asynchronous Receiver-Transmiter*, Transmisor-Receptor Asincrono Universal

Resumen

Los avances tecnológicos en el ámbito automotriz referente al internet de las cosas (IoT), han permitido crear bases de datos que se pueden emplear para determinar tráfico en zonas urbanas, niveles de contaminación y en un futuro hacer un diagnóstico preventivo, sobre las posibles fallas y desgastes dentro del vehículo. Sin embargo un 60 % del parque vehicular de México carece de tecnologías que permitan adquirir/enviar datos del automóvil, realizando habitualmente un diagnóstico correctivo, que tiene como principal problemática el encontrar las averías dentro del vehículo, en consecuencia se tienen contratiempos y añadido a esto, se reduce la vida útil de este, ya que no se previene los daños antes que se agraven. Entonces se diseñó un sistema de adquisición de datos vehiculares, que permitirá interactuar con el automóvil por medio del *OBD-II* y se integra un módulo *4G/LTE*, para el envío datos a la nube para que realice aplicaciones enfocadas en el *IoV* (Internet de los vehículos), con el fin de realizar un mantenimiento preventivo, como principal ventaja, es la de detectar averías antes de que estas sucedan, ampliando la vida útil de este. Este sistema está especializado para implementar una comunicación *CAN-BUS*, integrando un microcontrolador *ARM-Cortex M*, el cual posee: un controlador *CAN* (*Controller Area Network*), un transceptor *CAN* MCP2551 que permite acoplarse a los niveles físicos del bus, además de prevenir daños por corto-circuitos. Este trabajo fue desarrollado en el laboratorio de *SLED*, ubicado en la Facultad de Ciencias de la Electrónica. Una vez implementada la comunicación con la nube, se almacenarán los mensajes en una base de datos para posteriormente visualizarse en una interfaz gráfica (página web).

Capítulo 1

Introducción

El crecimiento progresivo de la urbanización ha dado el surgimiento de Megaciudades [17] donde viven más de 10 millones de habitantes. Se pronostica que para el 2030 más de dos terceras partes de la población mundial [18], se encuentren viviendo en dichas ciudades. Tal magnitud de población ocasiona congestionamientos vehiculares masivos en horas pico. Grandes ciudades como Sao Paulo, Beijing, Ciudad de México, París y Moscú sufren problemas de movilidad hoy en día [17].

Con el fin de resolver los problemas de logística y vialidad que surgen actualmente, los servicios de movilidad [17][19], proporcionan al conductor información del tráfico, accidentes viales, rutas alternas, ubicación de gasolineras, entre otros datos que permiten reducir el congestionamiento en vías altamente transitadas [20]. Aplicaciones móviles como *Waze*, *Michellin Navigation*, *Intrix traffic*, *M8*, entre otros más, son de los pioneros que actualmente están incursionando en este ámbito [21].

El desarrollo acelerado de las Tecnologías de la Información y Comunicación (TIC), permite la conexión de diversos dispositivos a internet, a esto se le conoce como el Internet de las cosas [22][23] (IoT). En 2020 se estimaron alrededor de 20 mil millones de dispositivos con IoT [23], de los cuales una gran porción pertenecen a automóviles. Bajo este marco surge el concepto de Internet de los Vehículos (IoV [23] [24] por sus siglas en inglés), enfocándose en la recolección de datos vehiculares y su interacción con la inteligencia humana, las cosas, el entorno y las redes, para proporcionar múltiples servicios para las grandes ciudades [17].

Entre algunos ejemplos del uso de IoV, se encuentran las *Apple CarPlay* [23], que ofrecen una completa integración de los mapas de *Apple* dando seguimiento de voz para llegar al destino. Similarmente, *Google Android Auto* [23] provee una interfaz libre de distracciones que permite a los conductores control de funciones como *GPS* mapeo/navegación, reproducción de música, mensajes (*SMS*), telefonía y búsqueda en la web.

Actualmente los vehículos de alta gama tienen cientos de sensores, actuadores y Unidades de Control Electrónico (*ECUs*) en las redes automotrices [17]. Cada *ECU* realiza una función específica en el automóvil, como controlar los frenos, vigilar el nivel de combustible y la temperatura del motor, entre otros más, así el vehículo es seguro y fiable [25]. Los datos obtenidos de los *ECUs* en conjunto con el IoV, habilitan la implementación de diversas aplicaciones como se muestra en la figura 1 [1]:

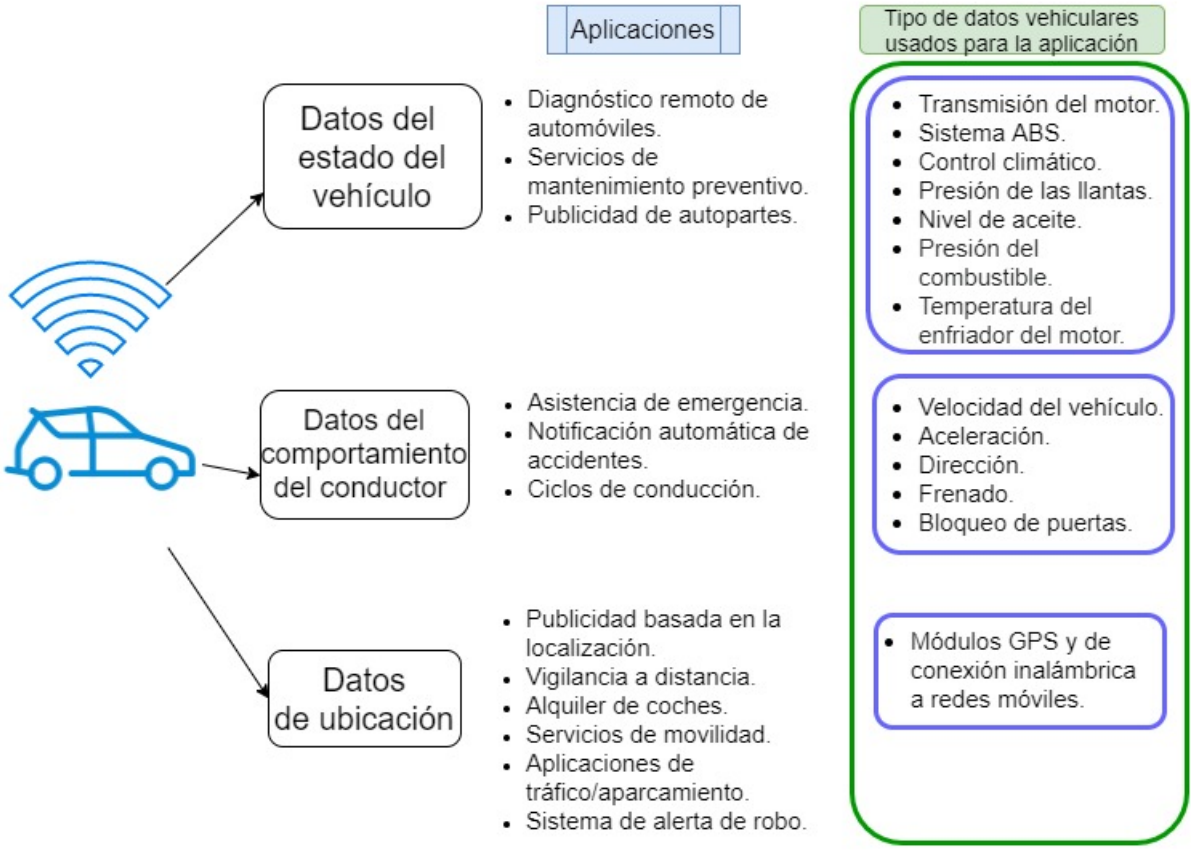


Figura 1.1: Esquema de las aplicaciones que surgen a partir de los datos obtenidos del vehículo [1].

Las datos de la figura 1, requieren una óptima y continua recolección para su almacenamiento en servidores [1]. Se estima que se generan alrededor de 40 terabytes de datos vehiculares cada año [17] [1]. Los analistas establecen que tiene sentido hablar del término *big data* [1] cuando se sobrepasa los 30 terabytes [26], denominado así por el gran volumen de datos estructurados como no estructurados disponibles en internet pudiendo proporcionar información útil del interés de las diferentes organizaciones [26].

Amazon y *eBay* son empresas que compiten dentro del mercado de venta en línea de autopartes [17], utilizando *big data* obtienen información sobre el estado de los componentes que posiblemente en un futuro estén por averiarse o bien tengan que ser reemplazados [26]. A través de campañas publicitarias [26], informan en redes sociales al conductor sobre ofertas de autopartes, con el fin de direccionarlos a su página de ventas en línea (*Amazon* ó *eBay*) y que el usuario compre la refacción que requiere [17]. *General Electric* ha invertido 1500 millones de dólares para adquirir empresas de *software* pequeñas que están dentro del ámbito de *Big Data* y *Analytics* [27] (es un conjunto de *software* inteligentes especializados en recopilar y procesar millones de datos sin dificultad, con el propósito de mejorar la toma de decisiones en las compañías [17]).

Para el 2030 se pronostica que al menos un 30 % de los vehículos serán autónomos en las 'Mega ciudades' [17], donde el conductor observa el comportamiento del automóvil detrás del volante o remotamente, reduciendo el tráfico con rutas óptimas, mediante algoritmos que seleccionen la trayectoria más favorable en colaboración con mapas altamente precisos [25]. Es fundamental la integración de múltiples cámaras, sensores y actuadores que envíen datos con una baja latencia, además de recibir avisos sobre incidentes y situaciones de peligro que se pudiesen presentar en tiempo real [25]. Así que es indispensable una red inalámbrica que trasmita y reciba grandes cantidades de datos en cortos periodos de tiempo [23].

La tecnología inalámbrica celular *2G* [28] permite el envío de audio digital y mensajes cortos a una velocidad de 14.4 kbps. Posteriormente llegó el *3G* [28], que trasmite audio y vídeo a 2 Mbps. Actualmente la red móvil *4G/LTE* [28][29], alcanza 100 Mbps. Las redes de quinta generación *5G* [30][31], tienen como objetivo llegar a velocidades superiores a 1 Gbps, sin embargo esta carece de suficiente infraestructura [31]. A finales de 2018 se estimaron 3.16 billones de usuarios [28] *4G/LTE*, siendo preferible para aplicaciones de *IoV*.

Los sistemas de recolección de datos vehiculares [32], actualmente son cruciales para el envío de las diversas señales eléctricas provenientes de los nodos [33][32] (sensores, actuadores y ECUs) del automóvil, estando conectados por la red *CAN bus* [1][33] (es un protocolo de comunicación serial diferencial asíncrono típicamente usado para la interconexión de ECUs).

Mediante el *OBD-II* [24][34] (sistema que sirve para detectar fallos químicos, mecánicos y eléctricos que afecten las emisiones de gases contaminantes del vehículo al medio ambiente y para identificar cualquier otra avería que presente el automóvil) se obtienen diferentes parámetros, como son las revoluciones por minuto del motor, aceleración, temperatura, nivel de combustible y aceite, entre otros [24], dichos datos son enviados hacia la nube (es una red de servidores remotos de todo el mundo que están interconectados funcionando como un ecosistema) utilizando tecnología *4G/LTE*, permitiendo el desarrollo de aplicaciones como:

- Hacer un diagnóstico en línea del automóvil [24].
- Ofrecer información para organizaciones, como la policía vial sobre patrones de comportamiento individual del conductor [17].
- Informar las causas del desgaste o fallo en una cierta autoparte [17].
- Operar vehículos de limpieza remotamente usando tecnología *4G/LTE* [29].

1.1. Objetivos

1.1.1. General

Desarrollar un dispositivo de recolección de datos que opere en tiempo real conectado al *OBD-II* de un vehículo, con almacenamiento en la nube utilizando tecnología *4G/LTE*, para estudios de diagnóstico automotriz.

1.1.2. Específicos

1. Investigar e identificar las características del protocolo automotriz *CAN bus* para su manejo.
2. Investigar y seleccionar los comandos del *OBD-II* que permitan obtener valores de 5 sensores que se utilizarán en el sistema a diseñar.
3. Determinar los requerimientos eléctricos, mecánicos y características del *OBD-II* para el diseño del dispositivo.
4. Seleccionar un módulo *4G/LTE* compatible con las bandas de frecuencia que se utilizan en México e identificar sus características técnicas y protocolos para el diseño del dispositivo.
5. Desarrollar un sistema empotrado para la recuperación de los valores de 5 sensores utilizando el *CAN bus* mediante los códigos del *OBD-II*, con almacenamiento de datos locales y en la nube vía *4G/LTE*.
6. Implementar y validar un algoritmo para convertir las señales eléctricas recibidas del *CAN bus* en datos que serán enviados por medio del módulo *4G/LTE* a la nube.
7. Probar el sistema completo en un vehículo para validar su correcto funcionamiento.

1.2. Justificación

En el 2019 se registró un parque vehicular de 32.5 millones de unidades en México [35], de las cuales 20 millones tienen entre 6 y 25 años de antigüedad. Representando más del 60 % de los vehículos del país, siendo estos los que carecen de las nuevas tecnologías actuales. El *IoV* esta enfocado principalmente en los automóviles híbridos, eléctricos y autónomos, debido a que contamina menos y predominaran en el 2030 [17], sin embargo en el país se tiene un 2 % de unidades. Una ventaja del IoV es la notificación al conductor de posibles averías y desgastes al disponer del estado parcial del automóvil en tiempo real, permitiendo realizar un mantenimiento preventivo.

Sin embargo, en México al tener un parque vehicular en su mayoría carente de tecnologías para el diagnóstico en línea, el conductor debe llevar su unidad con personal especializado en reparación automotriz, donde las averías son detectadas por algún dispositivo *OBD-II*, realizando así un mantenimiento correctivo, que tiene desventajas, tales como: el vehículo no se supervisa después de la compra, por lo que los fallos son muy imprevisibles, desconociendo que autoparte se necesita para la reparación y en caso de no estar disponible en el país se tendría que importar, incrementando costos y tiempo. Además no se protege ni cuida la unidad, lo que reduce la vida útil de este.

Bajo la problemática antes mencionada este trabajo de tesis desarrollará un sistema empotrado que integre el *OBD-II*, proporcionando datos vehiculares que se enviarán a la nube por medio de un módulo *4G/LTE*, teniendo a disposición el estado parcial del vehículo para un diagnóstico en línea, mediante una aplicación intuitiva para un dispositivo portátil (*smartphone*) accesible para el usuario y con información aprovechable para personal del área automotriz. Posibilita un mantenimiento preventivo que evite que se produzcan un fallo, notificando al conductor oportunamente y permite que la unidad dispongan de las ventajas existentes del IoV [23].

Actualmente se puede encontrar dispositivos con *OBD-II* con *4G/LTE* comercialmente, sin embargo al ser productos que tiene un armazón de plástico que no se puede modificar y dentro de ellos se encuentra una placa de circuito impreso (*PCB*) donde todos los componentes ya están soldados, dificulta agregar mejoras en su *hardware* y *firmware*, adicionalmente si este dispositivo se fabricó en un país distinto a México, donde su parque vehicular y sus usuarios no tiene las mismas necesidades, delimita el realizar plenamente ciertas investigaciones en el ámbito automotriz, también es importante notar que el *CAN bus* es una red no cifrada, vulnerable a un ataque cibernético o un mal uso de los datos.

Por lo que, es útil tener a disposición un *hardware* que se adapte a los requerimientos del proyecto, el parque vehicular del país y permita interactuar con el automóvil con un cifrado específico que asegure una comunicación segura con los servicios de nube, además de servir como base en el posterior desarrollo de diferentes aplicaciones e investigaciones en la Benemérita Universidad Autónoma de Puebla (BUAP) en el ámbito automotriz.

1.3. Estado del arte

Tras una revisión sistemática de la literatura referente a los sistemas de adquisición de datos vehiculares con comunicación *4G/LTE*, se encontraron varios trabajos que contribuyen al desarrollo de la solución propuesta. En el artículo de Y. Zhou *et al* [36] se presenta un trabajo donde se utilizó un microcontrolador *ARM-Cortex TM-A8* y un módulo *4G/LTE Quectel EC20* producido por *Shanghai Mobile*, que soporta redes *GSM/GPRS*. En vez de usar una red *CAN Bus* se plantea el uso del protocolo *FlexRay* que más adelante se describirá brevemente, el cual tiene ventajas como gran ancho de banda, alta fiabilidad y rendimiento en tiempo real, sin embargo como el trabajo de tesis va enfocado en automóviles con más de 6 años de antigüedad y estos no disponen de *FlexRay*, se descarta el uso de este protocolo.

Para enviar los datos a la nube, utilizaron el protocolo *HTTP* debido a su alta velocidad y bajo índice de pérdida de datos, almacenados en una base de datos realizada en *MySQL*. Después fueron transferidos los datos vehiculares en formato *JSON* desde el servidor hacia una aplicación para el *smartphone* para ser visualizados. Para esto se utilizó el kit de herramientas *Gson* de *Google* para realizar la conversión y el análisis sintáctico de *JSON*.

En el trabajo revisado I. S. BULUT *et al* [20] se plantea una propuesta para detectar el tráfico en ciertas zonas de Istanbul, Turquía. En contraste con el trabajo anterior se utilizó el escáner comercial *OBD-II Renault Megane*, este envía por *Bluetooth* los datos vehiculares a un *smartphone*. Entonces mediante una aplicación diseñada en *Android* establece comunicación con la nube, usando el protocolo *HTTP* con formato *JSON*, muy similar al trabajo anterior.

Sin embargo, otros sistemas como el descrito en el artículo A. Bin Masoud *et al* [37] hace una adquisición de datos vehiculares usando un escáner *OBD-II ELM327* comercial, comunicándose con una tarjeta de desarrollo *Raspberry Pi*, a través de un programa desarrollado en lenguaje *Python*. La ventaja de usar esta tarjeta de desarrollo es de transferir los datos al servidor usando el protocolo *MQTT*, que es ideal para aplicaciones del *IoT* por la taza de datos reducida que emplea. Para la programación del protocolo se utiliza *Node-red* que es bastante práctico, ya que solo utiliza bloques que se interconectan entre sí, con el fin de generar los pasos a seguir para en tablar conexión con la nube.

Por otra parte, el proyecto de A. Srinivasan *et al* [38], se asemeja con el de A. Bin Masoud *et al* [37], utiliza una tarjeta *Raspberry Pi 3*, pero se enfoca en la parte del procesamiento de *Machine Learning* utilizando algoritmos de aprendizaje automático como *KNN* y *Naive Bayes*, para determinar el estado del vehículo y la vida útil de varios actuadores como el motor, el refrigerante, etc.

1.4. Descripción

El sistema de recolección de datos automotrices se compone de distintas partes, como se puede ver en la figura 2, la primera es la red eléctrica del vehículo es donde están ubicados los sensores, actuadores y ECUs, conectados por la red *CAN bus*. A través del *OBD-II* se accede a este protocolo y a la alimentación del vehículo. Después se tiene el sistema empotrado encargado de almacenar temporalmente estos datos, para posteriormente transmitirlos hacia al módulo *4G/LTE* que será el encargado de establecer comunicación con el servicio de nube en específico. Mediante un dispositivo portátil se solicitarán estos datos para visualizarlos usando una aplicación. Del lado derecho se muestra el *firmware* del sistema empotrado.

Para interactuar con la red *CAN bus* este debe tener dos componentes:

1.- El transceptor [1] [33] encargado de acoplar los niveles de voltaje del controlador *CAN bus* con los manejados por la red. Usualmente este tiene un voltaje de alimentación de 5 V.

2.- El controlador [33] dispone de todas las funciones del protocolo *CAN bus*, como son la tasa de transmisión de datos, detección de errores por redundancia cíclica, etc. Este se puede encontrar incrustado dentro del microcontrolador, como en la figura 2 o en caso contrario se puede adquirir uno externo, controlándolo mediante comunicación *SPI*.

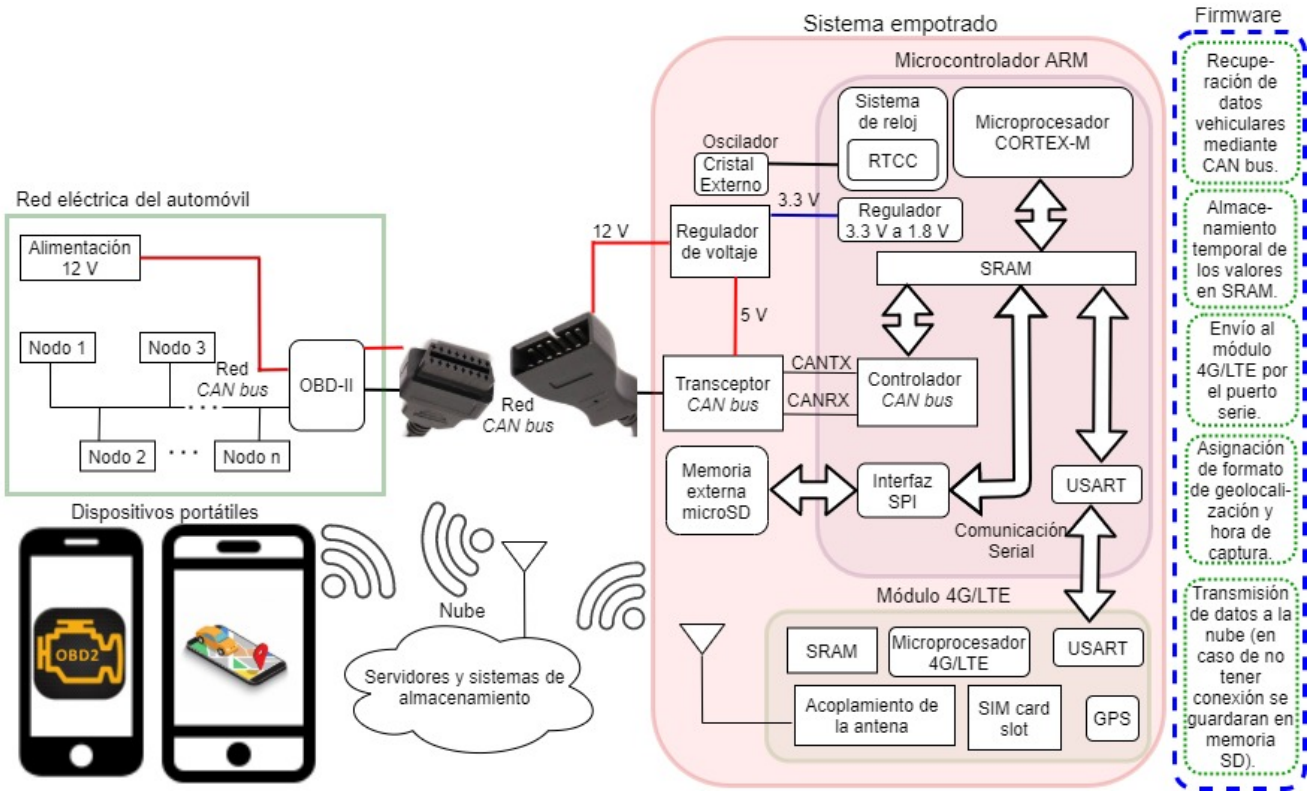


Figura 1.2: Esquema del sistema a desarrollar.

Cada dato es almacenado primeramente en la memoria estática de acceso aleatorio (*SRAM*) [39] del microcontrolador, después se le asigna un formato de hora, minuto y segundo de su captura, definida por medio de un reloj de tiempo real (*RTCC*) [40]. En caso de fallar la conexión con la nube o la trasmisión de los datos al módulo *4G/LTE* se propone almacenarlos en una memoria externa *microSD*, interactuando con esta, mediante la interfaz *SPI* [41] (ver figura 2).

Se ha optado por usar un microcontrolador *ARM* [42][43], ya que actualmente son utilizados en una gran variedad de aplicaciones automotrices, ordenadores, tablets, smartphones, etc. Este tiene una arquitectura *RISC* (del inglés *Reduced Instruction Set Computer*) de 32 bits, aceptando solo instrucciones compactas y ortogonales [42] que se puedan ejecutar en un único ciclo. Además de tener un bajo consumo de energía [43] y poseer varios periféricos, que posibilitan la integración de los elementos antes mencionados, como se ve en la figura 2.

La *USART* [44] es un dispositivo de comunicación serial altamente flexible, posibilita la transmisión-recepción de datos con el módulo *4G/LTE*. Una vez almacenados se les asigna un formato de geolocalización definido por el *GPS* [45] que integra el módulo como se puede apreciar en la figura 2. El *SIM card slot* [46], es una ranura para introducir una tarjeta *SIM* que opere con las compañías móviles del país y así enviar los datos a la nube.

1.5. Diagrama metodológico de actividades

Para el desarrollo del sistema empotrado, se usa una metodología en cascada que consta de 4 etapas (ver figura 3). La primera es para determinar cuales componentes y estándares se requieren para realizar el proyecto haciendo una revisión dispositivos comerciales y en fase de prototipo similares al que se desea diseñar.

Después se consulta la bibliografía que respecta a la red eléctrica de automóvil para así seleccionar y analizar los parámetros de 5 sensores con los cuales interactuar. En la tercera etapa se diseña en un *PCB*, el sistema empotrado para su posterior fabricación y construcción. Una vez terminado, se someterá a pruebas para determinar si todos los componentes están bien conectados y soldados.

Por último, se realizaran pruebas con un automóvil real, al introducir la placa dentro de un dispositivo que facilite la conexión al *OBD-II* del vehículo. Como se aprecia en la figura 2, se envía los datos hacia la nube, después son obtenidos y visualizados mediante una aplicación para un dispositivo portátil compatible con la red móvil *4G/LTE*. Posteriormente se plantea la colaboración con personas del área de Computación para un diseño detallado que integre varias funcionalidades.

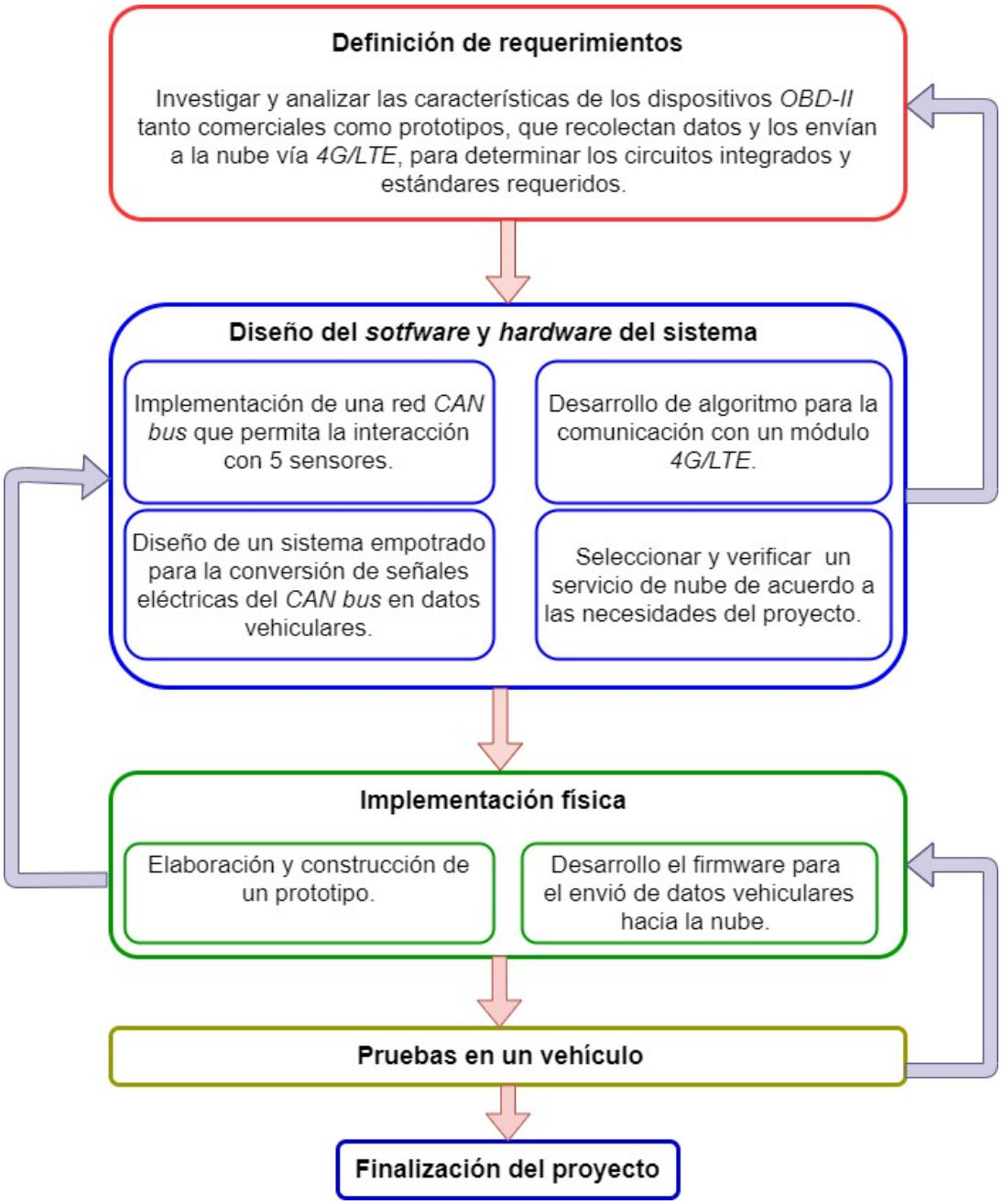


Figura 1.3: Diagrama metodológico de actividades.

1.6. Organización del documento

El presente trabajo se encuentra conformado de la siguiente manera. En el capítulo 2, se realizó una revisión de los fundamentos teóricos más importantes, relacionados con el *CAN bus* y la transmisión de datos vehiculares hacia la nube para un posterior diagnóstico en línea. Se partió desde la descripción del sistema electrónico del vehículo, donde se encuentra el sistema *OBD-II* encargado de supervisar constantemente todos los sistemas funcionales del automóvil y almacenar ciertos códigos de errores referentes a fallas que ocurran.

Así mismo, se realizó una comparativa de las tecnologías existentes en la transmisión de datos inalámbricos (*4G/LTE*, *Sigfox* y *LoRa*), también se compraron los diferentes microcontroladores (*ARM*, *PIC* y *Node MCU*) actuales que permiten una adquisición de datos mediante el protocolo *CAN* y las nubes actuales (*AWS*, *Google Cluod* y *Hive MQTT*), con el fin de determinar los mejores tecnologías para el proyecto.

El desarrollo del firmware del sistema empotrado se muestra en el capítulo 3, mostrándose las pruebas realizadas para validar el sistema y establecer comunicación con la nube. Al no tener acceso al laboratorio de sistemas automotrices para obtener los datos del vehículo, se emulo un *OBD-II* esto conforme al estándar *SAE1939*. Después al acceder al laboratorio correspondiente, se realizaron las pruebas correspondientes de validación del sistema.

En el capítulo 4, se muestran el diseño y fabricación del hardware, con base en las tecnologías seleccionadas. En el capítulo 5 finalmente, se muestra las conclusiones obtenidas a partir de los resultados obtenidos con el trabajo de tesis. Además, se presenta el trabajo futuro a realizar en los campos de investigación relacionados el diagnóstico en línea.

Capítulo 2

Tecnologías para realizar un diagnóstico en línea para vehículos

En este capítulo se enuncian los fundamentos teóricos esenciales para el diseño del dispositivo a realizar, comenzando desde el sistema electrónico del automóvil, las partes que lo conforman, enfocándonos principalmente en el *OBD-II*, donde se obtendrán los datos para realizar un diagnóstico en línea. También se muestran las tecnologías para diseñar, ensamblar y validar el dispositivo, comparando diferentes componentes, nubes y tecnologías de comunicación inalámbricas existentes, a fin de determinar la que mejor cumpla con los requisitos del proyecto.

Se añade a esta investigación temas como el protocolo *CAN bus* y su relevancia en la tesis, los protocolos para el *IoT* como son *MQTT* y *HTTP*, así también una parte relacionada con la nube y sus beneficios hoy en día para la comunicación, almacenamiento y procesamiento de datos en tiempo real.

2.1. Sistema electrónico del automóvil

El sistema electrónico se encarga de comunicar los diversos sensores y actuadores de todo el vehículo, para un correcto funcionamiento, para cumplir con este objetivo, se necesitan de la Unidad de Control Electrónico (por sus siglas en inglés *ECU*) [34] , este es un pequeño dispositivo encargado de controlar sensores y actuadores específicos del automóvil, así como su comunicación con los demás componentes del vehículo. Este término tiende a ser muy general, llegando a confundirse en la literatura, por lo que se le sugiere al lector revisar las abreviaturas puestas antes del primer capítulo del libro.

2.1.1. La Unidad de Control Electrónico (*ECU*) como dispositivo para la detección de fallas/averías en vehículos

Este dispositivo controla funciones específicas del vehículo, como son asientos, el motor, bolsas de aire, la dirección asistida, puertas, las ventanas eléctricas, sistema de frenado, etc [46]. La unidad tiene generalmente un sistema empotrado con un firmware enfocado a cumplir una tarea en específico, por lo que involucra comunicarse con otros *ECUs* que están ubicadas a lo largo del vehículo necesitando protocolos automotrices especializados, como es el *CAN bus*, *LIN*, *FlexRay* y *MOST*, los cuales se profundizará a detalle más adelante.

En vehículos actuales llegan a tener más de 100 *ECUs* [31], que interactúan con diversos sensores o actuadores, como por ejemplo, para bloquear una puerta, se recibe una señal cuando un pasajero presiona el botón de bloqueo/desbloqueo de un automóvil ó llavero inalámbrico, entonces la *ECU* correspondiente interviene convirtiendo la señal en un dato que trasmitirá por medio de un protocolo automotriz, hasta que llegue a un actuador, que activará el sistema mecánico de bloqueo de puerta.

Importante destacar que toda *ECU* debe cumplir con lo necesario para soportar ambientes robustos, con lo cual deben cumplir ciertos requisitos como son: temperaturas que van desde los -40 °C hasta los 125 °C, tener compatibilidad electromagnética (por sus siglas en inglés *EMC* [37], que describe la capacidad de un dispositivo puede funcionar sin complicaciones en su entorno electromagnético predeterminado) y tener protección a fluidos como el aceite, gasolina, entre otros. A continuación se describirán tres *ECUs* que son principales para el tema de tesis, ya que mediante estos se puede obtener información para hacer un diagnóstico en línea.

2.1.2. Principales ECUs en un automóvil

A continuación se presentarán 3 diferentes *ECU* dentro del automóvil, dejando en claro que existen otras más, pero estas tiene un papel relevante dentro del tema de tesis.

Módulo de control de motor (ECM)

El módulo de control de motor (en la figura 2.1 se puede observar una *ECM* de la marca *BOSCH*), reconocido mundialmente por sus siglas en inglés *ECM* (*Engine Control Module*) [4] [47], es un dispositivo encargado de gestionar diversos aspectos relacionados con el funcionamiento del motor, controlar las emisiones de gases contaminantes y facilitar la detección de fallos o averías. De esta *ECU* en particular obtendremos los datos de los 5 sensores a utilizar, que son:

- Sensor de temperatura del líquido de enfriamiento del motor.
- Sensor de las *RPMs* del motor.
- Sensor de velocidad del vehículo.
- Velocidad de flujo del aire (*MAF*).
- Sensor de temperatura del aire del colector de admisión.

El *ECM* recibe señales de entrada analógica de los sensores y controla varios dispositivos tales como inyectores de combustible, electroválvulas y válvula *IAC* [4]. Esta válvula en particular esta compuesta por un motor de pasos que controla el aire que fluye hacia las cámaras de combustión, según lo indique el *ECM*. Cuando se enciende el vehículo en frío, esta válvula es abierta en su totalidad para permitir una gran entrada de aire haciendo que el motor tenga alrededor de 1200 *RPM* (revoluciones por minuto), después esta se irá cerrando conforme el motor alcance una temperatura de 82°C, como consecuencia reduciendo las *RPM* del motor entre unos 800 y 900 [4] .

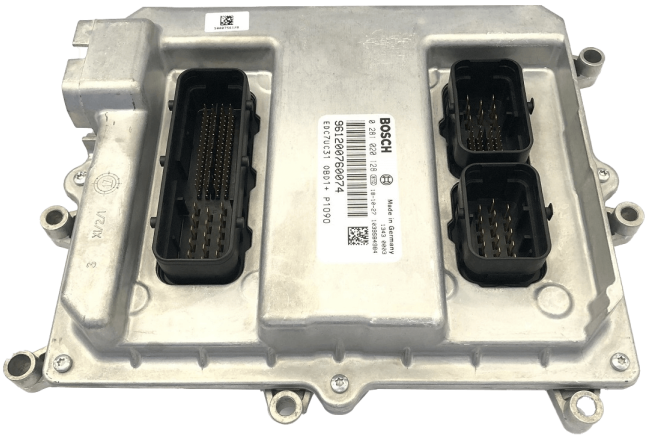


Figura 2.1: *ECM* fabricada por Bosh [2] .

Módulo control de transmisión (TCM)

Por sus siglas en inglés *TCM* (*transmission control module*) es un *ECU* encargado de controlar los cambios, en vehículos de transmisión automática, ubicado en la parte trasera del motor en el área de cortafuego [48], obsérvese que en la figura 2.2 se tiene un modelo fabricado por la empresa *MAZDA*. La *ECM* esta en constante comunicación con la *TCM*, ya que ambos se deben coordinar para que se ejecuten los cambios con seguridad y precisión. *TCM* tiene a disposición un módulo de autodiagnóstico que detecta fallas, las cuales son almacenadas en formato de código *DTC* [26], para posteriormente identificar áreas que puedan ocasionar averías en el transeje [48].

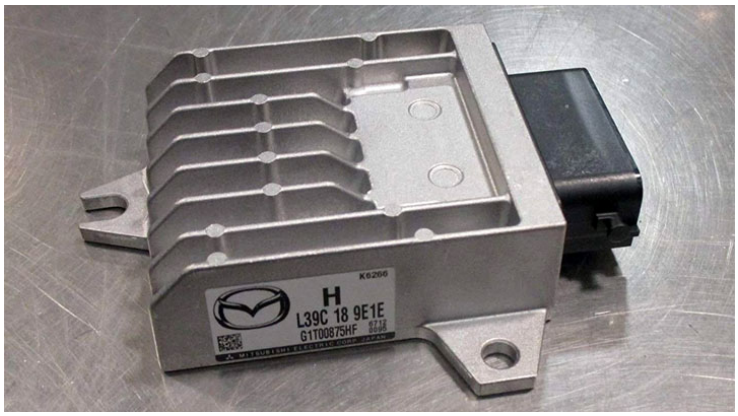


Figura 2.2: *TCM* de la empresa *MAZDA* [3] .

Sistema Antibloqueo de Frenos (*ABS*)

Enfocado en la seguridad del vehículo en función del frenado, evitando colisiones y accidentes. Por sus siglas en inglés *ABS* (*Anti-lock Braking System*) [4], tiene un circuito hidráulico como se muestra en la figura 2.3, que permita detener o disminuir la velocidad del automóvil (punto 3 y 6 de la figura 2.3), ejerciendo fricción en las ruedas. Mediante unos sensores magnéticos instalados en las ruedas sobre una corona dentada se obtiene la información necesaria para determinar la velocidad del automóvil (punto 7 de la figura 2.3) en todo momento [4]. Al pasar los dientes de la corona por el sensor producen una variación en el campo magnético, ocasionado una tensión en la bobina que rodea al sensor magnético. Estas variaciones de tensión conllevan una frecuencia, dando a la unidad de control electrónica la información necesaria para determinar la velocidad del automóvil en todo momento [4].

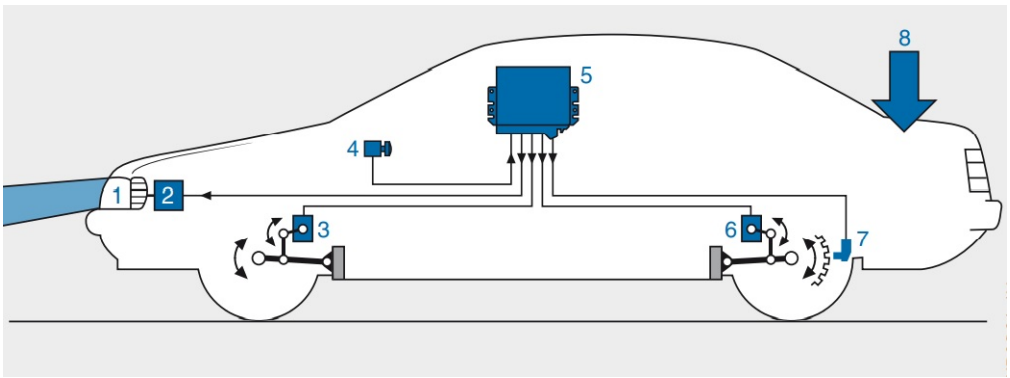


Figura 2.3: Esquema general del *ABS* en un vehículo [4]
divido en secciones.

2.2. El *OBD-II* como sistema para efectuar un diagnostico en vehículos

El sistema *OBD-II* tiene como funcionalidad la detección de fallos mecánicos, eléctricos y químicos, ya que estas averías pueden provocar emisiones de gases contaminantes hacia el medio ambiente. Además de identificar cualquier otra daño que se presentase en el automóvil. El *OBD-II* revisa que los sensores implicados en las emisiones de gases funcionen bien. Por ejemplo, en el catalizador, se tienen sensores de oxígeno, los cuales deben certificar un buen funcionamiento químico [34], de no ser así, el *OBD-II* registra una falla en el sistema para que el usuario revise u corrija posteriormente, añadido a esto, inmediatamente enciende la luz de falla, llamada *MIL* (por sus siglas en inglés *Malfunction Indication Lamp*) o Lámpara de indicación de mal funcionamiento, ubicada en el cuadro de instrumentos. Entonces el sistema *OBD-II* guarda un registro de fallas encontradas en el vehículo y en que condiciones ocurrieron. En general el conector *OBD-II* se ubica abajo del volante de piloto o en algunos modelos, esta a un lado del asiento del copiloto. Tanto para vehículos alimentados por gasolina y diésel (aunque también para algún otro combustible) las normativas de *OBD-II* son aplicables.
Empresas como *Volkswagen*, *Audi*, *Ford*, entre otras, desde el año 1996 hasta la fecha, están obligadas [49] a instalar en sus automóviles el sistema *OBD-II*. El *CAN* es uno de los tres protocolos incluidos en el estándar *OBD-II* para el diagnóstico de automóviles.

2.2.1. Representación de los datos del *OBD-II*

De acuerdo con la norma *SAE J1979* [50] y su equivalente técnico europeo el *ISO 15031-5* [50], establece para todos los fabricantes automotrices un estándar de códigos de averías. De esta forma es indistinto tanto el vehículo, como el equipo de diagnóstico empleado, ya que las pruebas se realizarán siempre de la misma forma. Sin embargo, no todos los fabricantes están obligados a implementar en su totalidad los códigos e inclusive pueden incluir los suyos, aunque estos adicionales son privados.

2.2.2. Modos de diagnóstico

También conocidos en inglés *Test Modes* [50] son comunes a todos los vehículos y permiten registrar datos para su verificación, extraer códigos de averías, borrarlos y realizar pruebas dinámicas de actuadores. Para facilitar la presentación y comunicación de datos, se tiene hardware de diagnóstico, que integra microcontroladores, conexiones y firmware apegado a la norma *SAE J1979*. A continuación se presentan los diferentes modos existentes dentro del estándar del *OBD-II*:

- **Modo 1**
  En este modo se usa la Identificación de Parámetro (por sus siglas en inglés *PID*), en el cual se envían un código en específico, que será por el protocolo *CAN bus*, para así acceder a valores analógicos o digitales de salidas y entradas a la *ECU*. Este modo es también llamado flujo de datos. Aquí es consultar, por ejemplo, la temperatura del motor o el voltaje generado por una sonda lambda.

- **Modo 2**
  Acceso a Cuadro de Datos Congelados. Esta es una función muy útil del *OBD-II* porque la *ECU* toma una muestra de todos los valores relacionados con las emisiones, en el momento exacto de ocurrir un fallo. De esta manera, al recuperar estos datos, se pueden conocer las condiciones exactas en las que ocurrió dicha avería. Solo existe un cuadro de datos que corresponde al primer fallo detectado.

- **Modo 3**
  Este modo permite extraer de la memoria de la *ECU* todos los códigos de fallo (*DTC - Diagnostic Trouble Codes* [51]) almacenados.

- **Modo 4**
  Con este modo se pueden borrar todos los códigos almacenados en la *PCM* (Módulo de control del tren motriz) [24] , incluyendo los *DTCs* y el cuadro de datos grabados.

- **Modo 5**
  Este modo devuelve los resultados de las pruebas realizadas a los sensores de oxígeno para determinar el funcionamiento de estos y la eficiencia del convertidor catalítico.

- **Modo 6**
  Este modo permite obtener los resultados de todas las pruebas de abordo.

- **Modo 7**
  Este modo permite leer de la memoria de la *ECU* todos los *DTCs* pendientes.

- **Modo 8**
  Este modo permite realizar la prueba de actuadores. Con esta función, el mecánico automotriz puede activar y desactivar actuadores como bombas de combustible, válvula de ralentí, etc.

2.2.3. Principales protocolos de comunicación automotriz

La cantidad de componentes de un automóvil que están en red han incrementado considerablemente los requerimientos de los sistemas de control del vehículo para comunicarse entre sí. La multiplexación [52] es un proceso de combinar varios mensajes para su transmisión a lo largo de la ruta de la misma señal. La ruta de la señal se llama bus de datos. Este bus de datos es básicamente solo un par de cables que conectan las unidades de control entre sí. Un bus de datos consta de un comunicador o cable de señales y un retorno a tierra, que da servicio a todos los nodos del sistema multiplex. Existen diferentes protocolos de comunicación que emplean el bus de datos antes mencionado. A continuación se mencionarán algunos que tiene el automóvil, pero se hará más énfasis en el *CAN bus* [1] debido a su relevancia en el trabajo de tesis.

LIN

Una red de interconexión local (*LIN*) [52] es un sistema de bus en serie conjuntado específicamente para sensores y actuadores dentro de un subsistema. Es un concepto para redes automotrices de costo bajo, el cual complementa las redes multiplex automotrices existentes como *CAN*. *LIN* capacita la implementación de una red jerárquica del vehículo. Este permite una mayor mejora en la calidad y una reducción de costo en los vehículos. *LIN* garantiza la interoperabilidad (capacidad de las plataformas digitales para intercambiar información, ya sean datos, documentos u otros objetos digitales, de manera uniforme y eficiente) [52] de los nodos de la red desde el punto de vista de hardware y software, y el comportamiento predecible de la compatibilidad electro-magnética (*LIN*).

FlexRay

Entre el año 2000 al 2009 se ha establecido el protocolo automotriz *FlexRay*, el cual tiene un ancho de banda máximo de 10 *Mbits/s* [52] [36], superando a *CAN bus* que soló alcanza 1 *Mbits/s*. Sin embargo al ser reciente, no es considerado un estándar en comunicaciones dentro del automóvil y por ende no está en el *OBD-II* como un protocolo para el diagnóstico automotriz. Este se compone de un controlador, que integra todas las funciones para generar la trama y de un transceptor para acoplarse a los niveles de voltajes de la red. El bus esta conformado por un par trenzado, con una correspondiente resistencia en paralelo a los cables, que va desde los 80 Ω hasta los 110 Ω [52]. Su principal función es trabajar en conjunto con el *CAN bus* en el tren motriz [52] para aplicaciones donde es necesario un mayor ancho de banda.

MOST

Por sus siglas en inglés *Media Oriented Systems Transport* [52] , es un protocolo destinado a la comunicación de dispositivos multimedia (vídeo, música, *GPS*, *4G/LTE*, etc.) en vehículos, pero a diferencia de los demás protocolos existentes, este utiliza la fibra óptica como medio para transmitir información, superando inclusive al protocolo *FlexRay* en velocidad. Sin embargo, no es tolerante a fallos y no podría ser utilizado en otras áreas como el *ABS* dentro del automóvil.

2.2.4. El protocolo *CAN bus* y su relevancia para la detección de fallos por medio del *OBD-II*

El *CAN bus* [4] [33] es un protocolo automotriz aceptado en 1986 en el congreso *SAE* (Sociedad de Ingenieros Automotrices) para el uso en el ámbito automotriz. Este protocolo vino a revolucionar la forma en como se transmitían los datos de los diferentes sensores y actuadores, ya que anteriormente en el vehículo, cada dispositivo necesitaba de su propio cable, implicando varios metros. Entonces el *CAN bus* al ser un único bus donde se transmiten datos, implicó un ahorro significativo de cable. Su tolerancia a fallos lo hace ideal para comunicarse con sistemas como el *ABS*, donde no se puede permitir datos erróneos que repercutan en un posible accidente. Además de tener una compatibilidad electromagnética (por sus siglas en inglés *EMC*, teniendo la capacidad de funcionar sin complicaciones en un entorno electromagnético), dándole más robustez y tolerancia a fallas, considerándose una red prioritaria en el vehículo [52].

A continuación, se muestran 5 características que resaltan de este protocolo:

1. Esta conformado por un par trenzado de cables, donde usualmente *CAN-HIGH* tiene color negro con naranja y *CAN-LOW* [52] es marrón con naranja (ver figura 2.4). Sin embargo, dependiendo de las necesidades de cada compañía, es libre de modificar la tonalidad a su conveniencia. Trenzar los cables reduce el efecto de *Crosstalk* [39].

Figura 2.4: Colores típicos de cable *CAN Bus* [5].

2. En la figura 2.5 se muestra la representación de un cero lógico, considerado por el protocolo como un bit dominante y para el caso contrario se considera un bit recesivo [33].

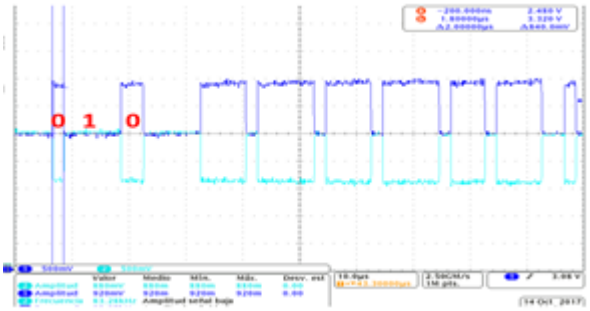


Figura 2.5: Trama *CAN bus* capturada en un osciloscopio donde la línea azul oscuro es *CAN-HIGH* y la de azul claro es *CAN-LOW* .

3. El rango de voltaje va desde 1.5 V a 3.5 V como se muestra en la figura 2.6, donde para *CAN-HIGH* va desde 2.5 V a 3.5 V y *CAN-LOW* desde 1.5 V a 2.5 V [52]. Si una señal parásita llegase afectar los niveles de tensión, como se observa en la figura 2.6, ambos cables varían de forma igual, pero aun así, su mensaje no es modificado. En dado caso, que la señal parásita afecte fuera del rango permitido (por ejemplo una variación que llegue a los 12 volts), el mensaje no será aceptado.

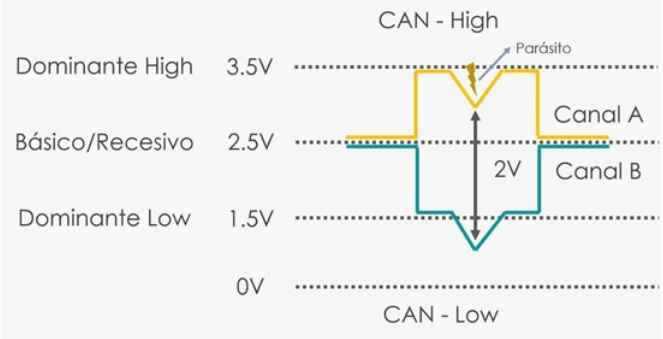


Figura 2.6: Compensación de influencias parásitas [6].

4. Como se observa en la figura 2.7, en paralelo a los cables del *CAN*, se conecta dos resistencias de 120 Ω. Un nodo es un sensor ó actuador conectado a la red [4].

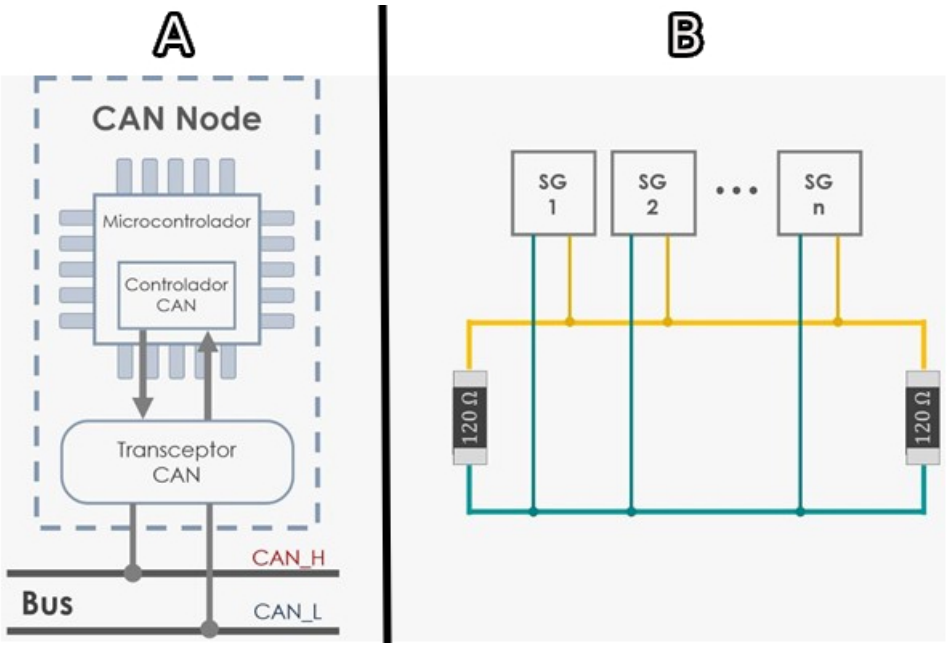


Figura 2.7: La figura A muestra los componentes internos de un nodo y a la figura B la red *CAN* [7].

5. Estrictamente se considera un *nodo* (figura 2.7) a cualquier microcontrolador que internamente tenga un controlador *CAN* [4] (como se mencionó en el capítulo 1, este dispositivo tiene toda la funcionalidad de la trama) ó en caso de no integrarlo, se puede conectar a uno exterior como por ejemplo el *MCP2515*. Después para acoplarse con los niveles de tensión del bus, se utiliza el transceptor *MCP2551* (ver figura 2.8).

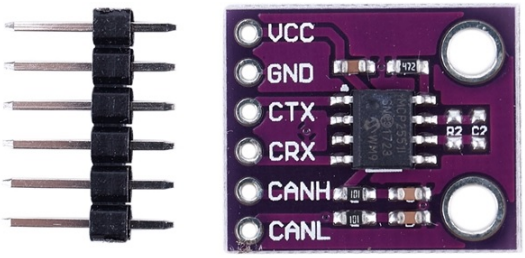


Figura 2.8: Vista frontal del transceptor *CAN MCP2551* de montaje superficial en una tarjeta para conexión a la tablilla de experimentos.

Trama del protocolo *CAN*

Se compone de múltiples bits yuxtapuestos. La figura 2.9 muestra la trama del protocolo *CAN* divido en secciones de bits.

- **El campo inicial** (*Start of Frame*) marca el comienzo del protocolo de datos.
- **En el campo de estado** (*Arbitration Field*) se establece la prioridad del protocolo de datos. Si, p. ej., dos unidades de control quieren enviar al mismo tiempo su protocolo de datos, tendrá preferencia la de mayor prioridad. Además, el contenido del mensaje está caracterizado.
- **El campo de control** (*Control Field*) indica el número de datos a enviar, siendo útil para el receptor que determinará mediante este campo cuantos bytes debe recibir (pude ir desde 1 byte hasta 8).
- **En el campo de datos** (*Data Field*) se contiene de 8 a 64 bits correspondientes al mensaje que se desea enviar hacia las demás unidades de control.
- **El campo de seguridad** (*CRC-Field*) sirve para identificar errores en la transmisión.
- **En el campo de confirmación** (*ACK Field*), los receptores señalizan al emisor que han recibido correctamente el protocolo de datos. Si se identifica una avería, lo comunican inmediatamente al emisor. Seguidamente, el emisor repite su transmisión.
- **El campo terminal** (*End of Frame*) consta de 7 bits recesivos que indica que ha finalizado el protocolo de datos.

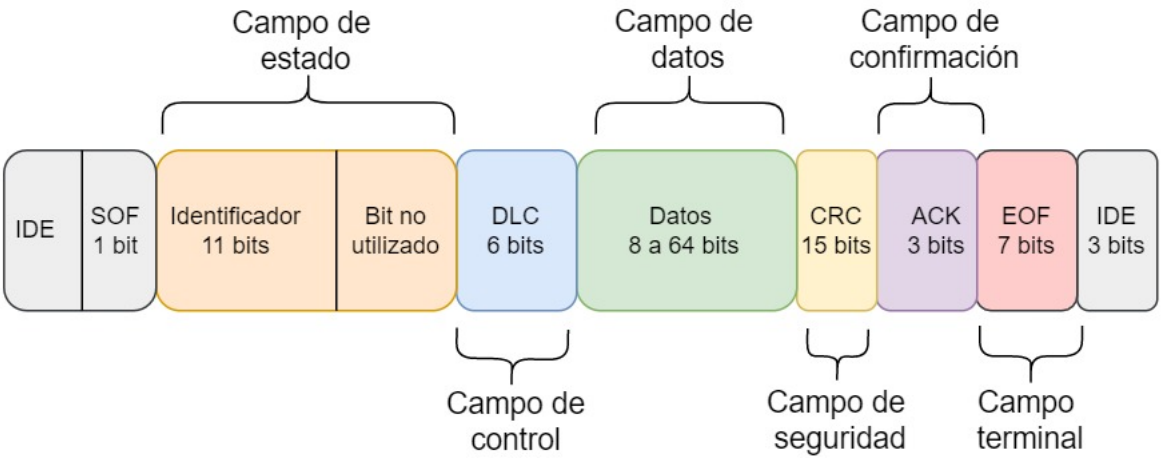


Figura 2.9: Composición de la trama *CAN* bus estándar, indicando el número de bits correspondiente en cada campo [6].

Trama remota

Esta trama similar a la trama de datos, teniendo estas dos diferencias:

- El campo de control compuesto por 6 bits, tiene los 4 bits menos significativos para indicar cuantos bytes enviar, mientras que el primer bit significativo, llamado bit *RTR*, indica si se enviará una trama remota al estar en recesivo.
- No hay ningún Campo de Datos, a menos que se cambie el bit *RTR* a dominante.

Al ser esta trama poco usada para aplicaciones automotrices no se tratará de indagar en ella.

Bit Timming

El *CAN bus* se sincroniza mediante el cambio de un bit dominante a un bit recesivo, es decir en un cambio de 0 a 1 lógico, sin embargo al tener tramas donde pudiera darse el caso donde se trasmitan más de 5 bits con el mismo valor, es necesario para una correcta sincronización de datos introducir el *Bit stuffing*, el cual es un bit de valor contrario a los que se venían transmitiendo. Por ejemplo, al mandar 5 bits recesivos el 6 bit deberá ser dominante con el fin de sincronizar correctamente el mensaje que recibe el receptor. Este bit añadido no será parte de la trama enviada y por ende el controlador *CAN* del receptor será encargado de eliminarlo.
La sincronización de *CAN bus* a nivel de bits se realiza mediante el *Bit Timming*, el cual se representa en la figura 2.10, mostrando el tiempo de un bit dividido en segmentos de tiempos llamados *Time Quanta*. Según lo establecido por la norma *ISO* 11898-1, se puede dividir un bit hasta un máximo de 25 partes, es decir 25 *Time Quanta*. Como se observa en la figura 2.10, se tienen 4 partes, empezando con el segmento de sincronización, que consta de 1 *TQ* solamente, después los siguientes segmentos tienen de 1 a 8 *TQ*. Mediante unas ciertas fórmulas, se calcula cuantos tiempos *TQ* les corresponde a estos tres segmentos, sin embargo es complicado realizar este proceso y para simplificarlo se utiliza un software encargado de darnos los tiempos correspondientes.

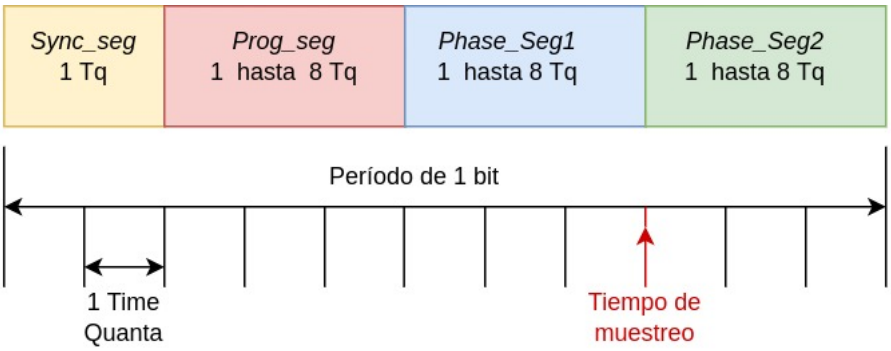


Figura 2.10: Esquema del *Time Quanta* correspondiente a un bit del *CAN bus* [8].

Tipos de red *CAN* dentro del automóvil

En lo que respecta al *CAN bus* dentro del automóvil está separado para cumplir ciertas actividades específicas como se observa en la figura 2.11. Es importante denotar que estas tres divisiones (*CAN-Tracción*, *CAN-Confort* y *CAN-Infotenimiento*) [52], siguen teniendo en común las propiedades antes mencionadas del *CAN* pero trabajan a distintas velocidades y cumplen ciertas tareas en específico. Para acoplarse entre sí, se tiene el *Gateway* [53], como dispositivo encargado de transmitir de manera fluida y segura los mensajes entre los tres tipos de redes *CAN*, así también comunicar entre sí, los protocolos como *LIN*, *FlexRay*, *MOST*, etc.

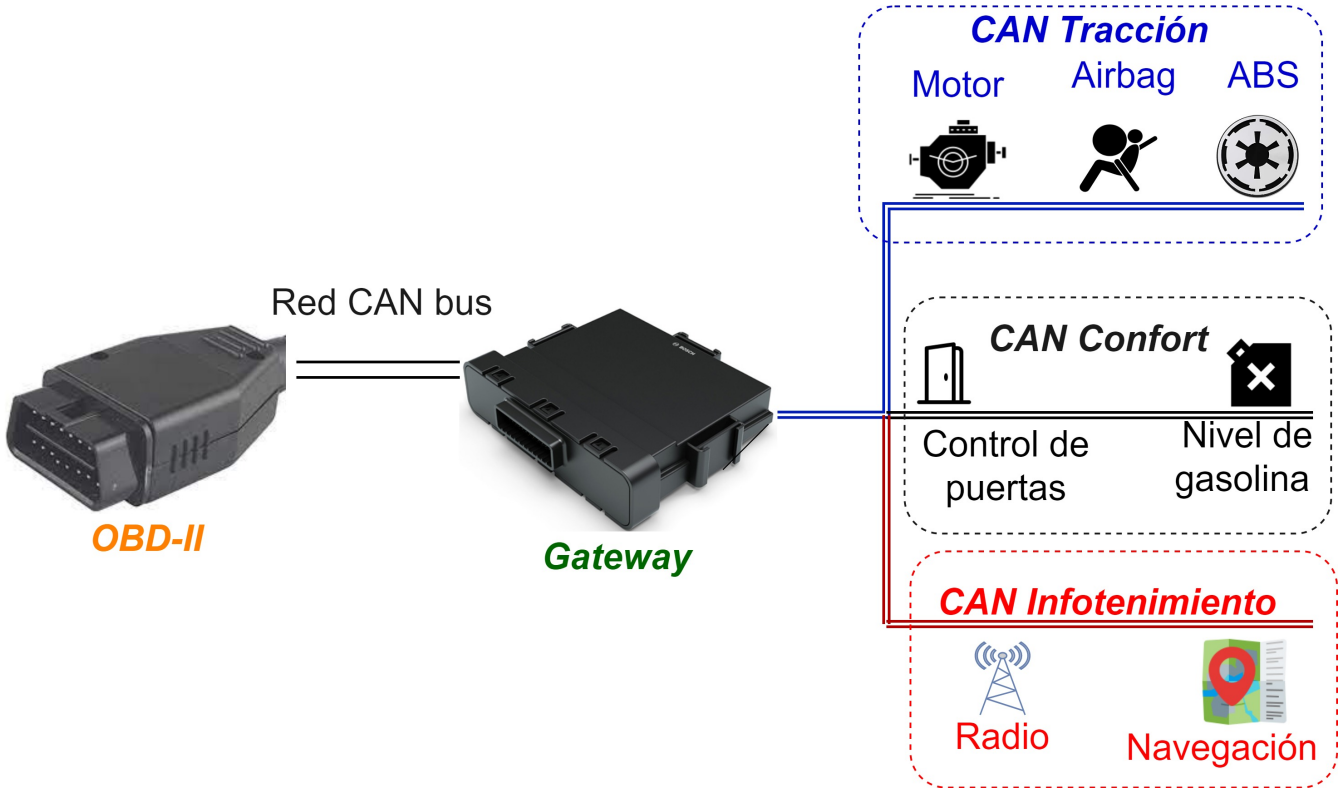


Figura 2.11: Estructura de los tipos de *CAN bus* dentro del automóvil [9].

CAN-Tracción

Considerado el bus más importante de los tres, ya que permite la comunicación con el *ABS* y maneja tareas cruciales para la seguridad tanto del vehículo como del pasajero, operando a 500 *kBit/s*, recibiendo el nombre de *CAN* de alta velocidad o en inglés *high-speed CAN*. El *CAN-Tracción* se comunica con las siguientes unidades de control:

- Unidad de control del motor.
- Unidad de control *ABS*.
- Unidad de control del cambio.
- Unidad de control airbag.
- Cuadro de instrumentos.

CAN Infotenimiento/Confort

Ambos envían datos a 100 *kbps*, considerados en la literatura como *low-speed CAN* (*CAN bus* de baja velocidad) [50], por lo cual tareas de menor prioridad son asignadas a estos dos buses, intercambiando datos en unidad de control como por ejemplo señalización de puertas abiertas/cerradas, unidades de iluminación interior que se encargan del encendido/apagado, posición del vehículo (*GPS*) y similares. Unidades de control del *CAN Confort/Infotenimiento* son, por ejemplo:

- Unidad de control para *Climatronic*/aire acondicionado.
- Unidades de control de puerta-unidad de control del área de confort.
- Unidad de control con unidad indicadora para radio y navegación.

Los códigos *PIDs*

Mediante el modo 1 se puede acceder a los códigos *PIDs* [54] que entregan información de las condiciones u operación del vehículo, muchos de los cuales son en tiempo real. Por decir, dan la información de la velocidad actual, las revoluciones del motor, la temperatura del catalizador, entre otros. Estos se agrupan en conjuntos por funciones, conocidos como servicios o modos *OBD*. En la tabla 2.1 se muestran algunos códigos que se pueden consultar por medio del *OBD-II* y que servirán más adelante para realizar pruebas con el vehículo.

El primer *PID* entrega 4 bytes de respuesta, que sirven para indicar cuáles *PIDs* se pueden consultar, para ejemplificar este proceso, el primer dato de 1 byte se le asigna la letra A, y a sus correspondientes 8 bits con A0, A1, A2, A3 y así sucesivamente (donde A0 es el byte más significativo). Para el segundo byte con la letra B y así hasta llegar al cuarto (donde D7 es el byte menos significativo). Al descomponer estos paquetes en su codificación binaria, los acomodamos de forma vertical como se muestra en la figura 2.12, donde el primer bit A0 corresponde al *PID* 0x01, el segundo A1 al 0x02 y así sucesivamente. Si el valor de los datos recibidos corresponde a 1 lógico, es decir A0 = 1, entonces el valor si se puede consultar, en caso de ser 0, queda descartado. Mediante este proceso podemos determinar que datos se consultan a través del *OBD-II*.

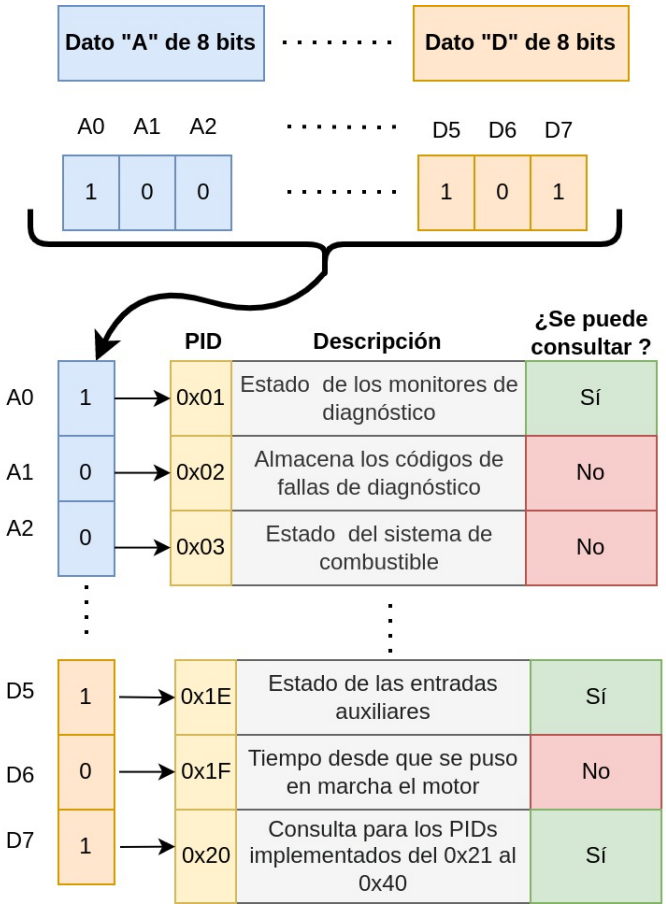


Figura 2.12: Mediantes los bytes A,B,C y D se determinan cuales *PIDS* se pueden consultar [10].

A continuación se explican algunos de los parámetros de la tabla 2.1 y otros que serán de relevancia en el capítulo 5:

Temperatura del líquido de enfriamiento del motor

Este parámetro indica la temperatura del líquido refrigerante que circula por el motor, donde se tiene comúnmente unos 80 °C a 105 °C [55], aunque esto depende del tipo de automóvil, por ejemplo

para vehículos que trasportan cargas pesadas, se tiene un rango de mayor debido al esfuerzo ejercido.

Temperatura del aire del colector de admisión

Para determinar la cantidad de aire que entra a los cilindros, se adquieren datos del MAP (sensor de presión absoluta del colector de admisión), del CKP (sensor de giro del cigüeñal) y del IATS (sensor de temperatura del aire de admisión) [56]. Si este parámetro indica un valor de temperatura negativo como -40 °C o bien en caso contrario 215 °C y el vehículo se encuentra a una temperatura ambiente (20 °C a 25 °C), es posible que este sensor esté descompuesto, ocasionando un mal funcionamiento en el arranque del motor y por ende se tenga que reemplazar.

Velocidad del flujo del aire (MAF)

Este sensor, al igual que el anterior, ayuda a determinar cuanto aire entra al motor, para así determinar la cantidad de combustible que se debe inyectar o cuanto se debe abrir la válvula EGR (esta se encarga de reintroducir el humo de la combustión del motor en los cilindros para así reducir las emisiones de óxidos de nitrógeno) [57]. El caudalímetro (así también se le denomina al MAF) se ubica en la admisión, después del filtro del aire, para así protegerlo de cualquier partícula del exterior [57].

Voltaje del módulo de control

Esta es la medición del voltaje que le llega al módulo de control, el cual puede variar de 11 V (usualmente cuando está apagado el vehículo) hasta 15 V cuando esta en movimiento, pero idealmente debe tener un voltaje similar a los 13 V [58].

Relación equivalente comandada de combustible - aire

Este factor (asignado con la letra griega λ), designa la proporción aire/combustible (en peso) en forma de mezcla que entra al cilindro de un motor de ciclo Otto [59]. Se tienen dos consideraciones, cuando este factor, es mayor a 1 (por ejemplo 1.15), se denomina mezcla pobre, indicando que se tiene un consumo mínimo de combustible y emisiones mínimas de contaminantes, principalmente hidrocarburos y monóxido de carbono (HC y CO), pero no torque o par motor máximo. Caso contrario, donde es menor a 1 (por ejemplo 0.88), se considera mezcla rica, donde se tiene mayor contaminación, pero un mayor torque. Idealmente los vehículos deben tener un un factor cercano a 1 y este no puede ser mayor a 2 [59].

PID **(hex)**	**Bytes de respuesta**	**Descripción**	**Valor mínimo**	**Valor máximo**	**Unidad**	**Fórmula**
0	4	*PIDs* implementados [01 - 20]				Cada bit indica si los siguientes 32 *PID* están implementados (1) o no (0): [A7..D0] == [*PID* 01..20]
1	4	Estado de los monitores de diagnóstico desde que se borraron los códigos de fallas *DTC*.				Codificación en bits, no relevante para el tema de tesis.
2	2	Almacena los códigos de fallas de diagnóstico *DTC*.				
3	2	Estado del sistema de combustible.				Codificación en bits, no relevante para el tema de tesis.
4	1	Carga calculada del motor	0	100	%	A/2.55
5	1	Temperatura del líquido de enfriamiento del motor.	-40	215	°C	A-40
. . .						
0A	1	Presión del combustible	0	765	kPa	3A
0B	1	Presión absoluta del colector de admisión.	0	255	kPa	A
0C	2	*RPM* del motor	0	16,383.75	rpm	(256A+B)/4
0D	1	Velocidad del vehículo	0	255	km/h	A
0E	1	Avance del tiempo	-64	63.5	° antes TDC	A/2-64
0F	1	Temperatura del aire del colector de admisión.	-40	215	°C	A-40
10	2	Velocidad del flujo del aire *MAF*	0	655.35	gr/sec	(256A+B)/100
11	1	Posición del acelerador	0	100	%	A/2.55

Tabla 2.1: *PIDs* del 0x00 al 0x11 de acuerdo a la *SAEJ1939*[14]

2.3. El uso de la Nube para un diagnóstico en línea

El mantenimiento predictivo [25], tiene como objetivo identificar los problemas que se puedan originar en el vehículo antes de que estos ocurran analizando los datos de los sensores del automóvil. El análisis predictivo puede extraer datos de prácticamente todos los vehículos de un año y comparar esa información para buscar algún tipo de tendencia, permitiendo anticiparse a una avería puntual o incluso subsanar el problema de los vehículos antes de su producción, si se tratara de un error de fabricación o diseño [25]. Para esto es necesario el uso de un servicio de nube [20] que permita ejecutar en tiempo real un procesamiento de los datos almacenados en su base de datos y notificar oportunamente al conductor de los posibles desgastes del vehículo.

2.3.1. El Internet de los Vehículos

Compartiendo en su núcleo conceptos similares al Internet de las cosas (IoT) [24], Internet de vehículos (IoV) [25] se refiere a la comunicación inteligente entre vehículos, automotores o dispositivos electrónicos a través del intercambio de información a través de la nube. Dichas aplicaciones automatizadas pueden ser muy beneficiosas en términos de control de vehículos, seguridad vial, gestión del tráfico, vigilancia urbana o eficiencia del transporte, pero aún deben ser capaces de funcionar debajo condiciones rigurosas, así como temperaturas extremas, humedad, altitud, golpes, vibraciones, apagones repentinos, o largas horas de operación [20].

2.3.2. Servicios de Nube actuales y sus características para el procesamientos de datos

El término de nube [20] [38] es empleado para describir una red de servidores remotos ubicados en todo el globo terráqueo, conectados entre sí para funcionar como un ecosistema. Los servidores deben tener la capacidad de almacenar y administrar datos masivamente, recibiendo solicitudes diariamente para acceder a vídeos, correos, música, etc. Un ejemplo de nube, sería las redes sociales *Facebook*, *Instagram* o bien *Youtube*, que diariamente los usuarios acceden desde todo el mundo, además de visualizar y subir constantemente contenido a sus servidores. Si bien, se han presentado fallas en estas redes sociales, en cuestión de horas, se ha restablecido el acceso a ellas, esto gracias a la interconexión de servidores, que permite que reparar o encontrar los fallos en un servidor en específico, mientras los demás siguen funcionado.

Existen 4 tipos de nubes, la primera es la pública [60], donde se ofrecen servicios a los usuarios a través de internet, sin ninguna restricción; una nube privada [60] por su parte, no comparte información de manera libre, solo está dispuesta para un número específico de usuarios, que pueden acceder a sus recursos. La tercera es la híbrida [60] que entabla comunicación a través de nubes tanto públicas como privadas, dependiendo de las necesidades de la empresa o las personas que soliciten el servicio. La nube comunitaria, están diseñadas para organizaciones, como puede ser instituciones educativas, gubernamentales, etc.

Las ventajas que tiene los usuarios al emplear la nube, es que pueden escalar servicios para satisfacer sus necesidades, personalizar aplicaciones y acceder a servicios desde cualquier lugar con una conexión a internet [61] . Los usuarios empresariales pueden llevar las aplicaciones al mercado rápidamente, sin preocuparse por los costos de infraestructura subyacentes o el mantenimiento. Los usuarios pueden elegir ofertas de almacenamiento público, privado o híbrido, según las necesidades de seguridad y otras consideraciones [61].

Entre las desventajas principales es necesitar acceso a internet, tener cierta dependencia de los proveedores de este tipo de servicio confiando en su tecnología y funcionamiento. La modifican continuamente las interfaces de las aplicaciones y una posible sobrecarga en los servidores si el número de usuarios es muy alto o no se sigue una política de uso adecuada [62].

Cloud computing

El *Cloud computing* también conocida como computación en la nube (en algunas literaturas se denomina como servicios en la nube), son los servicios que ofrece la nube, ya sean desde una aplicación (*software*), almacenamiento de datos, infraestructura, etc. El usuario elige que tipo de servicio desea y no se preocupa por la logística usada, el consumo de energía requerido o el lugar en el globo terráqueo donde se almacenan los datos, él solo debe considerar que tipo de proyecto debe realizar y cuanto sería el costo o puede ser gratuito [63] [62].

Hoy en día los servicios en la nube ofrecen una amplia gama de funcionalidades y herramientas para suplir las necesidades de empresas, usuarios o inclusive entidades gubernamentales, entre ellas se mencionarán 3 modelos con el fin de ejemplificar el *Cloud Computing*:

Software como servicio (***SaaS*** [64])

El usuario puede acceder a *software* en línea ya sea gratuitamente (usualmente con recursos limitados) o bien adquiriendo un plan de licencia [64], que va desde pagar por horas de uso, días, semanas o hasta anualidades, dependiendo del tipo de recursos solicitados. Un ejemplo es las redes sociales como *Facebook*, *Whatsapps* y *Tik Tok*, donde se accede a un software para comunicarse, ver vídeos o jugar *online* con diferentes personas; pero a pesar de ser gratuitos, el usuario inminentemente tendrá que ver anuncios, a menos que pague una versión para bloquearlos (como *Youtube premium*).

Plataforma como servicio(***PaaS*** [65])

Empresas de desarrollo de aplicaciones web, móviles, entre otras, usan este tipo de servicios para crear, optimizar o modificar sus desarrollos que van desde una página web, hasta juegos en línea que utilizan un considerable cantidad de recursos y datos en el momento. Entonces aquellas compañías que adquieren el servicio solo deben enfocarse en el desarrollo del software, mientras que por su parte, el proveedor, ofrece la manutención, administración y la infraestructura requerida [65]. Por ejemplo para *Google Cloud* se tiene a *Firebase* (ver tabla 2.2) como servicio *PaaS*, que ofrece un almacenamiento tanto gratuito como de paga, donde el usuario no debe preocuparse sobre la energía consumida, ni los mantenimientos que lleve guardar sus datos, él solo debe enfocarse en el desarrollo de su proyecto.

(Infraestructura como servicio(***IaaS*** [66])

Para ciertos proyectos de *IoT*, se emplea el uso de *IaaS*, ya que ofrece infraestructura, servidores, *racks*, *routers*, etcétera, para la transmisión de datos, así como su almacenamiento. Un caso donde se utiliza este tipo de servicio, es en la agricultura, donde los sembradíos están alejados de las antenas *5G*, *4G* y demás tecnologías anteriores, entonces es coherentes contratar hardware que permita transmitir una reducida cantidad de datos en regiones alejadas [66]. Añadido al servicio *PaaS* que se esta utilizando, se investigó los diferentes servicios *IaaS* existentes, comparándolos en la tabla 2.3, para en un futuro seleccionar uno que se adecue a las necesidades de la tesis.

Características	***Google Cloud*** [67]	***AWS*** [68]	***Hive MQTT*** [69]
Almacenamiento gratis o de paga	*Firebase Realtime Database* 1 GB (gratis)	*Amazon S3* 0.023 *USD* por GB al mes	10 GB (gratis)
Lenguajes	*Java, Python, Node js, PHP, C++, Go, Ruby*	*Java, Python Ruby, PHP, Node js*	*Java, C++, Python, Javascript C#*
Documentación	Amplia documentación intuitiva, con guía paso a paso. Adicionalmente tiene una comunidad en crecimiento.	Amplia documentación sin embargo es confusa y su comunidad es pequeña.	Documentación precisa sin embargo es básica y no profundiza en diversos temas como el procesamiento de datos. Comunidad en crecimiento.
Soporte para *MQTT* y *HTTP*	Si	Si	Si
Duración de plan de prueba	3 Meses	3 Meses	3 Meses
Costo por el servicio en dólares al mes	$67.31 500 dispositivos Incluye un almacenamiento de 15.947089838 *GB* según su calculadora	$125 500 dispositivos No incluye almacenamiento	$53.13 500 dispositivos 15 *GB* de almacenamiento

Tabla 2.2: Comparación de las tres principales nubes actuales.

Como se observa en la tabla 2.2, se compara *Google Cloud* [67], *Amazon Web Services* (*AWS*) [68] y *Hive MQTT* [69], los cuales tiene características similares entre sí, aunque se destaca *Google* debido a su documentación intuitiva, ya que para usuarios novatos o con poca experiencia es bastante útil esta información, ahorrando tiempo en desarrollo de aplicaciones.

Comparativa de compañías de comunicación inalámbrica enfocadas en el *IoT*

Actualmente existen diversas compañías que ofrecen tecnología para una comunicación inalámbrica (*IaaS*), en su mayoría venden una tarjeta *SIM*, similar a la de una compañía telefónica y para el envió de datos a internet, utilizan la infraestructura propia o de terceros, como por ejemplo, aquí en México se apoyan de *Telcel* o *Movistar*. A diferencia de las dos empresas mencionadas anteriormente (las cuales van enfocadas con planes para 1 usuario o familiar), estas van destinadas para muchos dispositivos y disponen de planes tanto para un número reducido de *Mega Bytes* al mes hasta llegar a una cantidad de *Giga Bytes* al año, esto dependerá de los requerimientos solicitados para un proyecto. Por ende para tener un mejor contraste sobre estas compañías, en la tabla 2.3 se realizó una investigación al respecto sobre 4 empresas con sus características y sus costos.

Compañías de *IoT*				
Características (Todos los costos son en pesos mexicanos)	*Things mobile*	*Hologram Io*	*Emnify*	*1NCE*
4G/LTE	Si	Si	Si	Si
5G	No	No	No	No
Seguridad	Desconocido	-*APN* privado -*SSH* -*TCP/UDP*	-*VPC* (Red vitual privada en la nube)	-*APN* privado -*OpenVPN*
Plataforma para visualizar datos respecto al tiempo, datos usados, etc.	Si	Si	Si	No
Costo de *SIM*	$0	$100	$0	$437.41 Incluye -500 MB -1 año de subscripción.
Costo por activación	$53.89 por la activación	$0		$0
Costo por MB	$0.4	$3.79	$4.84	$1.14
Subscripción	$40	Gratis	Uso constante de la tarjeta *SIM*, sino se cancela.	-1 año de subscripción al pagar el costo de la *SIM*.
Plan básico	$598.24 Incluye -50 dispositivos -500 MB -30 días	$316.70 Incluye -100 dispositivos -250 MB -30 días	$1713.79 Incluye -500 dispositivos -2000 MB -30 días	$437.41 Incluye -500 MB -1 año de subscripción
Plan más costoso	$398,826.00 Incluye -10 k dispositivos -500 GB -1 año	$1219.38 Incluye 1 k dispositivos -2G GB -1 año .	$8485.43 Incluye -10 k -2000 MB dispositivos -30 días	Desconocido.

Tabla 2.3: Comparativa de 4 compañías *IoT* enfocadas en la comunicación inalámbrica.

2.3.3. Comparativa de los protocolos *IoT* para la comunicación con la nube

Existen dos protocolos de comunicación para el *IoT*, que usualmente se emplean en casi todo el mundo, esto son a *HTTP* y *MQTT*. Si bien existen otros, estos dos son los favoritos para los usuarios, ya que en la web se puede encontrar una basta bibliografía referente a las bibliotecas de ambos protocolos.

HTTP

Protocolo de transferencia de hipertexto o por sus siglas en inglés *HTTP* [70], desarrollado en 1999, donde el usuario hace solicitudes al servidor y este responde bajo el formato establecido por este protocolo. A continuación se muestran los siguientes métodos de envío/recepción de datos:
GET [70] para obtener datos, *PUT* [70] para enviar datos (objeto, archivo o bloque), *POST* para crear un recurso y *DELETE* [70] para eliminarlo. Como ventaja a destacar es su capacidad de enviar

y recibir vídeos, archivos, imágenes, audio, etcétera, de manera eficiente.

MQTT

MQTT [69] desarrollado en 1999 para la empresa *IBM* con uso privado y puesto libre en el 2010, su modo de operación se basa en la publicación/suscripción a un tópico (este sirve para filtrar mensajes que son recibidos desde los publicadores). El máximo tamaño de datos a enviar con este protocolo es de 256 MB [69], siendo poco en comparación a *HTTP* donde inclusive se pueden transferir archivos de varios *Giga Bytes*. Sin embargo tiene como ventaja que al ser compacto en sus mensajes, consume menos energía y como múltiples dispositivos de *IoT* usan baterías, es recomendable el uso de este protocolo.

Comparación de ambos protocolos

Dependiendo de los factores a considerar en el proyecto, se seleccionará uno en específico, dando como prioridad el tamaño de datos que se transmitirán, ya que si es una cantidad considerable, es recomendable *HTTP*, por otro lado, si es pequeño y añadido se está empleando una batería, entonces es mejor *MQTT*. En este caso al ser datos del vehículo y del *GPS* a transmitir, se sugiere usar *MQTT*, pero al ser una protocolo de comunicación reciente, es posible presentar fallos de comunicación, utilizando entonces *HTTP*. En la tabla 2.4, se comparan ambos protocolos:

Características	***MQTT***	HTTP
Tipo de datos	Binario	*ASCII*/Texto
Patrón	Publicar/Suscribirse	Solicitud/Respuesta
Ancho de banda	Bajo	Alto
Distribución	Uno a varios	Uno a uno

Tabla 2.4: Comparación de los protocolos *MQTT* y *HTTP* [15] .

2.4. Tecnologías inalámbricas para la transmisión de datos del *OBD-II* hacia la nube

Para el proyecto son necesarios módulos inalámbricos que puedan transmitir datos en dispositivos portátiles, como son los vehículos, entonces se revisará las tecnologías existentes y se compararán entre sí, con el fin de elegir la que cubre mejor los requerimientos del trabajo de tesis.

SigFox

Empresa de origen francés, fundada en el año 2009, ofrece un servicio de tecnología de comunicación inalámbrica de bajo consumo de energía, comúnmente conocido como *Low-Power Wide-Area Network* (*LPWAN*) [71], además de tener tasas de transferencia de 12 bytes, ideal para aplicaciones con pocos sensores y que usan baterías convencionales. Características:

- Hasta 12 bytes por mensaje.
- Las antenas tiene una cobertura hasta 250,000 dispositivos.
- La distancia que alcanza las antenas para transmitir un mensaje puede variar dependiendo del medio pero se define entre 1-5 kilómetros en áreas urbanas densas y hasta 15-30 kilómetros en áreas rurales.
- Los mensajes se pueden cifrar con el fin de dar seguridad al usuario.
- Suscripción del tipo anual.

LoRa

En sus inicios esta tecnología tenía fines de uso militar y espacial, aunque actualmente es usada para el *IoT* [72]. Sus ventajas:

- Alta tolerancia a las interferencias.
- Alta sensibilidad para recibir datos (-168 dB).
- Bajo Consumo (hasta 10 años con una batería).
- Largo alcance 10 a 20 km.
- Baja transferencia de datos (hasta 255 bytes).
- Frecuencias de trabajo: 868 MHz en Europa, 915 MHz en América, y 433 MHz en Asia.

Entonces por sus características presentadas, es ideal para aplicaciones donde se necesite bajo consumo de energía y grandes distancias como en la agricultura, ganadería y *Smart Cities* [17] (ciudades inteligentes). Aunque también *LoRa* ofrece servicios de redes privadas, con el fin de proteger mensajes de distintas índoles.

LTE

Sus siglas en inglés viene del término *Long Term Evolution* [28] [36], ofrece una banda ancha inalámbrica con una velocidad 10 veces más rápida que su antecesor *3G*. Integrada en la mayoría de *smartphones* y dispositivos portátiles. Para dimensionar la rapidez de esta tecnología, usuarios han reportado en foros como descargaron en 90 segundos, un vídeo de 700 MB [36], con lo cual es ideal para redes sociales como *YouTube*, donde se visualizan vídeos en resolución 4K. Por último, esta tecnología integra los protocolos *MQTT* y *HTTP*.

A continuación en la tabla 2.5, se muestra una comparativa de las tres tecnologías presentadas anteriormente, donde se puede observar como *LTE* es superior a las demás, sin embargo como desventaja tiene un consumo energético mayor al de *Sigfox* o *LoRa*.

Características	***Sigfox***	***LoRa***	***4G/LTE***
Rango de alcance	<13 km	<11 km	<11 km
Espectro	868-915 MHz	433-915 MHz	700-900 MHz
Ancho de Banda	100 kHz	<500 kHz	1.4 MHz
Velocidad de transmisión/recepción	<100 kbps	<10 kpbs	<1 Mbps
Cobertura	En 2019 planean llegar al 85 % del territorio en México	En algunos estados (principalmente en Aguascalientes)	En todo el país, donde la compañía *Telcel* es la más amplia alcanzando 210,000 poblaciones

Tabla 2.5: Comparación de los principales protocolos de comunicación inalámbricos [16].

2.5. Microcontroladores *ARM-Cortex* y sus ventajas para la comunicación con el *OBD-II*

La mayoría de los fabricantes de semiconductores ofrecen microcontroladores basados en tecnología *ARM* [73] [41]. Estos microcontroladores ofrecen una amplia gama de periféricos y una inmejorable relación precio-prestaciones. La arquitectura *ARM* tiene un conjunto de instrucciones simple pero eficiente que permite un tamaño de silicio compacto y ofrece alta velocidad de ejecución a bajo consumo

de energía [73]. Para tener una idea más clara de las ventajas de esta tecnología, se contrasta en la tabla 2.6, un *PIC32* de arquitectura de 32 bits y un *ESP8266* que tiene una arquitectura semejante a la de un *ARM*.

Características	*PIC32 PIC32MZ1025DAK*	*ARM-CORTEX M4 STM32F446RE*	*ESP8266*
RAM	256 *KB*	512 *KB*	128 *KB*
Máxima frecuencia	200 *MHz*	180 *MHz*	80 *MHz*
Periféricos	*CAN, I2C, SPI, UART*	*CAN, I2C, I2S, SDIO USART, USB*	*GPIO, I2C, I2S SDIO, SPI, UART WIFI*
Soporta sistema operativo	Si, *FreeRTOS*, 236 *Bytes RAM*, 10 *KB ROM* [74]	Si, *FreeRTOS*, 236 *Bytes RAM*, 10 *KB ROM* [74]	Si, *micropython* que es un sistema operativo interprete (*SOIM*) 16 *KB RAM* [75]
Precio en dólares	$20.87 [76]	$13.66 [77]	$9.04 [78]

Tabla 2.6: Comparación de los principales microcontroladores dominantes en el mercado.

Como se puede observar en la tabla anterior, el microcontrolador *ARM*, tiene un precio competitivo y brinda diversos periféricos, entre ellos el *CAN bus*, el cual no está disponible en el *ESP8266*, además de trabajar a altas frecuencias. Debido a las características mencionadas, es interesante profundizar en los microcontroladores *ARM-Cortex*, además se tiene al igual que *Google Cloud*, una documentación e comunidad que favorecen el desarrollo de aplicaciones para el *IoV*. A continuación se describirán brevemente algunas familias de microcontroladores *ARM*.

ARM-Cotex R

ARM Cortex-R es una familia de núcleos *ARM* que implementan el perfil R de la arquitectura *ARM*; ese perfil está diseñado para aplicaciones críticas de seguridad y en tiempo real de alto rendimiento [73].

ARM-Cotex A

La serie de procesadores de aplicaciones *ARM Cortex-A* proporciona una gama de soluciones para dispositivos que realizan tareas informáticas complejas, como el alojamiento de una plataforma de sistemas operativos (*SO*) sofisticados y el soporte de múltiples aplicaciones de software. Los procesadores de la serie *Cortex-A* se escalan eficientemente a través de una gama de dispositivos de consumo, integrados y empresariales de alto rendimiento. Estos incluyen un espectro de teléfonos inteligentes, plataformas informáticas móviles, televisores digitales, decodificadores y redes empresariales [73]. En un entorno empresarial cada vez más consciente del consumo de energía, la eficiencia energética de los procesadores *Cortex-A* puede proporcionar ventajas significativas.

ARM-Cotex M

La familia *Cortex-M* [73] es una serie de microprocesadores diseñados para bajo consumo de energía. Las principales ventajas de esta arquitectura estandarizada es la facilidad de migrar el código entre las distintas variantes de la familia de microcontroladores. Esto permite encontrar el equilibrio perfecto entre desempeño, consumo de energía, seguridad y cantidad de periféricos disponibles [41]. Permite, también, elegir entre un gran ecosistema de fabricantes que eligen la arquitectura *ARM*, que se traduce un gran abanico de herramientas de hardware y software de desarrollo.

2.5.1. Arquitectura general de los microcontroladores *ARM Cortex-M*

Los microcontroladores de la arquitectura *ARM Cortex-M* tienen una arquitectura relativamente compleja en comparación con los microcontroladores clásicos de 8 bits (8051, *PIC16*, *AVR*, etc.), pues incorporan mecanismos avanzados más habituales en procesadores de aplicación general [41].

Sistema de reloj

El esquema de la figura 2.13, muestra el sistema de reloj en los microcontroladores *ARM-Cortex M4*, siendo de manera similar para las demás familias. Para explicar este esquema se describen los siguientes términos:

- *HSE*: Por sus siglas en inglés *High Speed External Clock*, es un oscilador externo.
- *HSI*: Reloj interno de bajo costo y opera a 16 MHz.
- *AHB*: Interfaz esclava que permite a las *CPUs* internas y a otros periféricos bus master acceder a memorias externas.

Un cristal externo se conecta al *OSCIN* y *OSC OUT* para hacer oscilar al *HSE* y dar la frecuencia de trabajo del microcontrolador, esto siempre y cuando el *mux* esté activado para dejar pasar esa señal, en caso de no ser así, se podrá trabajar el microcontrolador mediante el *HSI*, lo cual nos ahorra un oscilador externo. Entonces una vez seleccionado la señal de ese *mux*, su salida deberá pasar forzosamente mediante el *PLL* (circuito de realimentación negativa que acepta una frecuencia en su entrada y que proporciona en la salida una frecuencia o una tensión), este tiene tres señales que se definen como:

- *PLLCLK*: Señal que será la frecuencia del microcontrolador (180 MHz es la máxima frecuencia).
- *PLLQ*: Frecuencia para el *USB OTG* de 48 MHz.
- *PLLR*: Señal para el protocolo *I2S* o también funciona como reloj de alta velocidad.

En el caso de la señal de reloj usado para el periférico *CAN bus*, se debe utilizar el *PLLR* que sale del *PLL* y configurar el *mux* que recibe esta señal, para que la deje pasar a su salida, descartando las otras tres (*HSE*, *HSI* y *PLLCLK*). Por último pasamos por los prescaladores *AHB* y el *APB1* para tener la frecuencia deseada, que hará funcionar el controlador *CAN*.

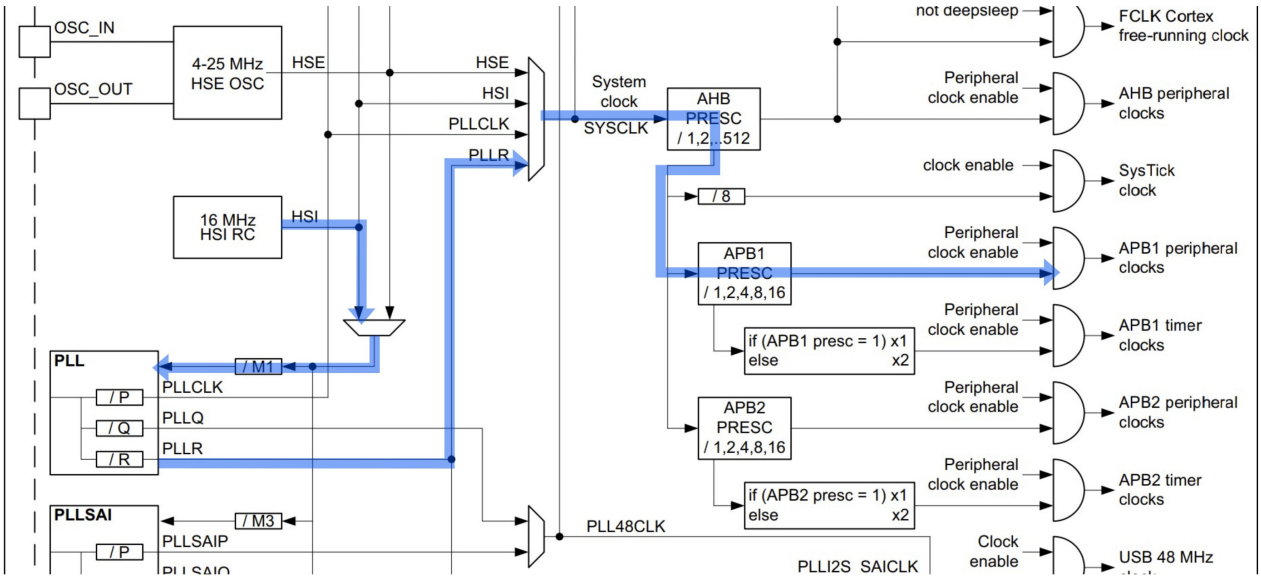


Figura 2.13: Esquema del sistema de reloj *CAN bus*, resaltando en color azul la trayectoria que debe seguir la señal de reloj correspondiente [11].

Módulo *CAN bus*

El periférico *CAN* integra de uno a tres módulos, particularmente en los microcontroladores *ARM* de la compañía *STMicroelectronics*, pero en general para la familia M4 tiene dos módulos, donde el primero es maestro y el segundo esclavo. Si bien el *CAN* es multimaestro, es decir que todos los nodos pueden iniciar la transmisión de mensajes de manera asíncrona siempre y cuando el bus este libre, siendo el nodo de mayor prioridad el que toma el control del bus y los demás reciben la información, pero para el caso del segundo módulo de los *M4*, no pueden iniciar comunicación a menos que se le solicite por medio de otro nodo.

Modos de operación

Se tienen tres modos de operación para el *CAN*, es importante, tener seleccionado un modo para que el periférico funcione correctamente. Los modos son:

- ***Sleep***: El controlador *CAN bus* está en bajo consumo de energía, solo se puede acceder a los *buffers* de mensajes que ya ha recibido anteriormente. Este despertará cuando se encuentre actividad en el bus.
- **Inicialización**: Se configura la velocidad del *CAN* (*bit timming*) y filtros. No se reciben ni transmiten datos.
- ***Normal mode***: Transmisión de datos y recepción.

Primero se determina que pines serán *CAN RX* y *CAN TX*, ya que en general los pines de I2C pueden configurarse como *CAN*, después se pasa a modo de *Inicialización*, determinando la velocidad en el bus, el filtrado de mensajes (explicado más a detalle en la sección 3.2.2), la interrupción y una vez realizado esto, pasamos a *Normal mode*, para comenzar a transmitir datos en el bus. Entre las características que tenemos para este protocolo en los *ARM-Cortex* son:

- Velocidad máxima de transferencia de 1 *Mbit/s*.
- Acceso a 256 bytes de *SRAM*.
- Para cada módulo *CAN* se tiene una interrupción.
- *CAN bus* en versión 2.0 A y B (la versión A tiene un identificador de 11 bits, el cual es utilizado en este tema de tesis, mientras que el B, tiene 29 bits).

Módulo *SPI*

En 1985 la empresa *NPX Semicondutor* desarrolla el protocolo *SPI* [12], este a diferencia del *CAN bus*, tiene una señal de sincronización (*SCK* mostrada en la figura 2.14), la cual genera pulsos que llegan hasta los 50 MHz [12]. Sin embargo, eso no da garantía que los bits transmitidos no sea erróneos, entonces dependerá del microcontrolador, *FPGA*, entre otros, que al implementar este protocolo, se utilice la redundancia cíclica, para verificar errores, ya que al no estar estandarizado, el usuario es libre de determinar cuantos bytes se transmiten por paquete, siendo ideal para enviar largas cadenas de datos.

De igual manera que *CAN* y *I2C*, tiene la posibilidad de transmitir mensajes entre diversos dispositivos conectados en bus, para ello como se observa en la figura 2.14, la línea *MOSI* (Maestro salida, esclavo entrada) del maestro, se conecta al *MOSI* del esclavo, con esto se transmiten datos desde el maestro hacia el esclavo y de manera opuesta, mediante el MISO (maestro entrada, esclavo salida) del maestro conectado al MISO del esclavo. Entonces para controlar a un esclavo en específico se utiliza *CS* (selección del *CHIP*), donde al activarlo, el esclavo recibirá información. Pero si se tiene más de un esclavo, es necesario agregar otras señales de *CS*, lo que tiene como desventaja agregar una línea de *CS* por cada esclavo que controle, es decir por 100 esclavos, se tendrá en el maestro 100 señales de *CS*.

Como almacenamiento local se tendrá una memoria *SD* de *16 G*, que guardará los datos obtenidos del *OBD-II*. Para controlarla, se empleará el periférico *SPI*, que incluye bibliotecas para crear un archivo *FAT32* dentro de la memoria y así almacenar los datos.

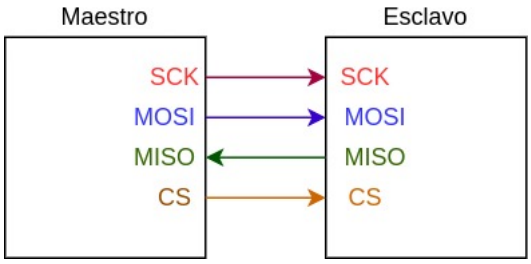


Figura 2.14: Conexión típica del protocolo *SPI*, entre dos dispositivos [12].

Módulo *USART*

La *USART* [13] ó por sus término en inglés *Universal Synchronous and Asynchronous serial Receiver and Transmitter* es un dispositivo de comunicación serial, este periférico utiliza dos cables como se muestra en la figura 2.15, donde de manera intuitiva el *TX* (señal que transmite datos) del dispositivo uno se conecta al *RX* (señal que recibe datos) del segundo dispositivo y viceversa. A diferencia de los demás protocolos mencionados, en este solo dos dispositivos se pueden comunicar a la vez, entonces para comunicar a un tercero, se tendría que utilizar un cuarto dispositivo y su comunicación sería por cables independientes. Por esto los microcontroladores *ARM-Cortex*, tienen al menos dos *USARTs*. Entre las velocidades que maneja son de 4800, 9600, 19.2 k, 57. 6 k y 115.2 k [13], siendo esta última velocidad la utilizada para comunicarse con el módulo *4G/LTE*.

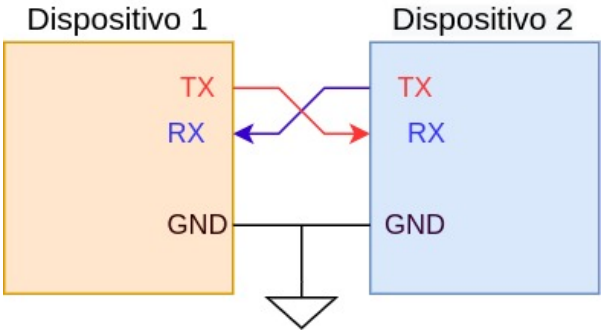


Figura 2.15: Conexión general en la comunicación serial [13].

2.5.2. Conclusión

Se muestra un panorama de los requerimientos necesarios para establecer comunicación con el *OBD-II* de un vehículo, necesitando la red CAN bus, el conocimiento de los *PIDs* y la forma en como representar estos datos en un valor con unidad, que puede ser en °C, Voltaje, *RPM*, distancia, etc. La visualización de estos parámetros se hace mediante una página web, es por esto que se mostraron las nubes actuales y las tecnologías de comunicación inalámbricas para transmitir información a estas. Entonces mediante la información antes mostrada, se da pie a una comparación y selección de la tecnología a emplear para el desarrollo del firmware del sistema de adquisición de datos.

Capítulo 3

Desarrollo del *firmware* del sistema de adquisición de parámetros del vehículo y su visualización.

Con base en lo investigado en el capítulo anterior, se puede determinar que tecnologías son convenientes para el desarrollo del *firmware* del sistema de adquisición de datos del OBD-II (por su acrónimo SADO). Para esto, primero se debe tener en claro cuales son las necesidades del proyecto y que tecnologías se adecuan mejor. Requerimientos:

1. Adquisición de datos mediante el *OBD-II* de un vehículo.
2. Almacenamientos local de datos.
3. Envío de datos a la nube mediante tecnología inalámbrica *4G/LTE*.
4. Adquirir un servicio de nube que almacene lo transmitido por el SADO en una base de datos.
5. Interfaz gráfica que permita visualizar los datos obtenidos de los sensores del vehículo.

El *CAN bus* es un protocolo integrado en el *OBD-II*, debido a esto el sistema empotrado a desarrollar debe tenerlo, por ello en la tabla 2.6 se comparan tres principales microcontroladores dominantes en el mercado, donde el microcontrolador *ARM Cortex-M*, tiene un precio competitivo y brinda diversos periféricos, entre ellos *CAN bus*. La empresa *STMicroelectronics* en particular tiene una documentación y comunidad que favorecen el desarrollo de aplicaciones con microcontroladores *ARM Cortex*, esto debido al *IDE STM32CubeMX* que genera código de inicialización para los periféricos que se empleen, ahorrando tiempo al usuario.

Para el segundo punto, es necesario un elemento que guarde datos, pero que no sea tan costoso en comparación a los discos de estado sólido y que integre un protocolo de comunicación típico, como *SPI*, *I2C*, serial, etc. Actualmente las memorias micro SD, son ideales para almacenar los datos obtenidos del *OBD-II*, ya que tienen un precio módico (entre $800 pesos para una memoria de 256 GB) [79] y se pueden controlar mediante *SPI*. Además *STM32CubeMX* tiene la opción de generar firmware para controlar la memoria SD mediante el sistema *FAT32* [80].

SIMCom Wireless Solutions es una de las empresas que fabrican módulos de comunicación inalámbrica *4G/LTE*, esta ofrece dispositivos compatibles con las bandas de frecuencia en México (B2, B4 y B5), además tiene a disposición del usuario *datasheets* que muestran las conexiones, componentes electrónicos y los requerimientos que se usaron para la fabricación de sus tarjetas, sirviendo de base para este proyecto. En la comparativa de la tabla 2.2, se destaca *Google Cloud* entre *AWS* y *Hive MQTT* por su documentación intuitiva, además de tener *Firebase* como gestor de base de datos que ofrece 1 GB de almacenamiento gratis.

El punto 5, se describe la creación del *Frontend* (parte visual de una página web), donde se debe convertir lo almacenado en la base de datos en una interfaz gráfica que permita una visualización de forma intuitiva y para ello se hace uso de *Vercel* (plataforma para el desarrollo del *Frontend*), por tener compatibilidad con *Git Hub* (base de datos para códigos fuente), guardando en un repositorio el código del *Frontend*, teniendo compatibilidad con *Visual Studio Web* (aplicación web que permite hacer uso de diferentes lenguajes de programación), para que así accediendo a la cuenta de *Vercel* se tendrá sincronizado los archivos del proyecto.

3.1. Etapas del prototipo

En el diagrama mostrado en la figura 3.1, se observan los bloques utilizados para el desarrollo del prototipo del SADO. Primero se realizó la emulación del *OBD-II* mediante el microcontrolador *STM32L432KC*. Por otra parte, el ARM Cortex *STM32F446RE*, servirá como base para las pruebas y desarrollo del firmware. Ambos deben tener un transceptor *CAN bus* que les permita acoplarse en los voltajes correspondientes.

Una vez obtenidos los datos a enviar, se les asigna un formato *JSON*, ya que así lo requiere la base de datos. Entonces mediante el módulo *4G/LTE* se emplea *HTTP* para la transmisión de datos. La configuración y control del módulo se hace a través de los comandos *AT*, para esto se efectúa una comunicación *USART* entre ambos dispositivos como se ve en la figura 3.1.

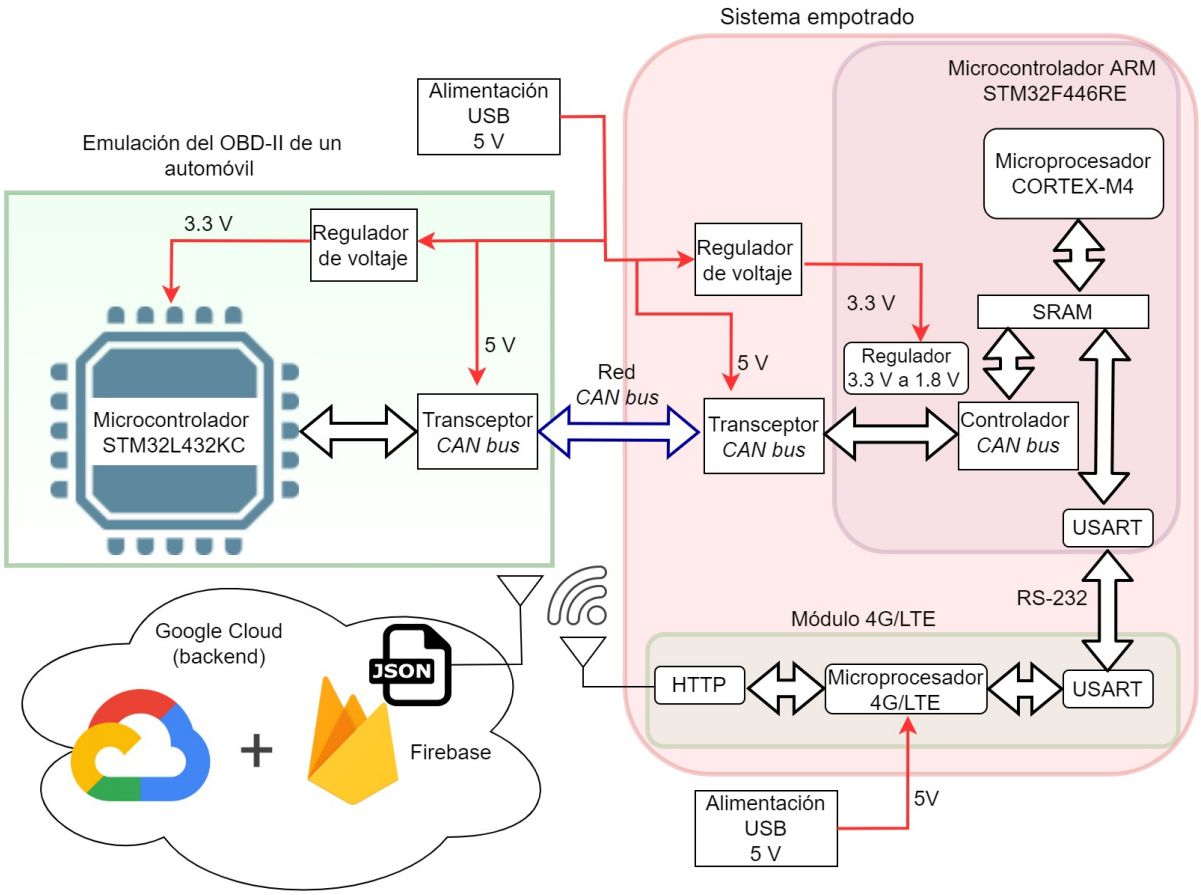


Figura 3.1: Esquema general del sistema desarrollado a lo largo del este capítulo.

3.2. Implementación de una red *CAN bus* con dos microcontroladores *ARM Cortex-M*

La conexión entre los microcontroladores *STM32F446RE* y *STM32L432KC* para armar la red *CAN* se muestra en la figura 3.2, donde se requiere de dos transceptores *MCP2551*, que funciona a 5 V y los microcontroladores a 3.3 V. Se tiene dos resistencias en paralelo de 120 Ω conectadas a los pines *CANL* y *CANH* de ambos transceptores. El pin *Vref* no tiene alguna conexión y el pin *NRST* (el *reset* de los microcontroladores) lleva una resistencia de 10 kΩ y una conexión en *pull up* a un pulsador. Esta primera prueba consiste en enviar datos desde el microcontrolador *STM32L432KC* por medio de CAN bus y mostrarlos en 8 *leds* conectados al puerto A del otro microcontrolador.

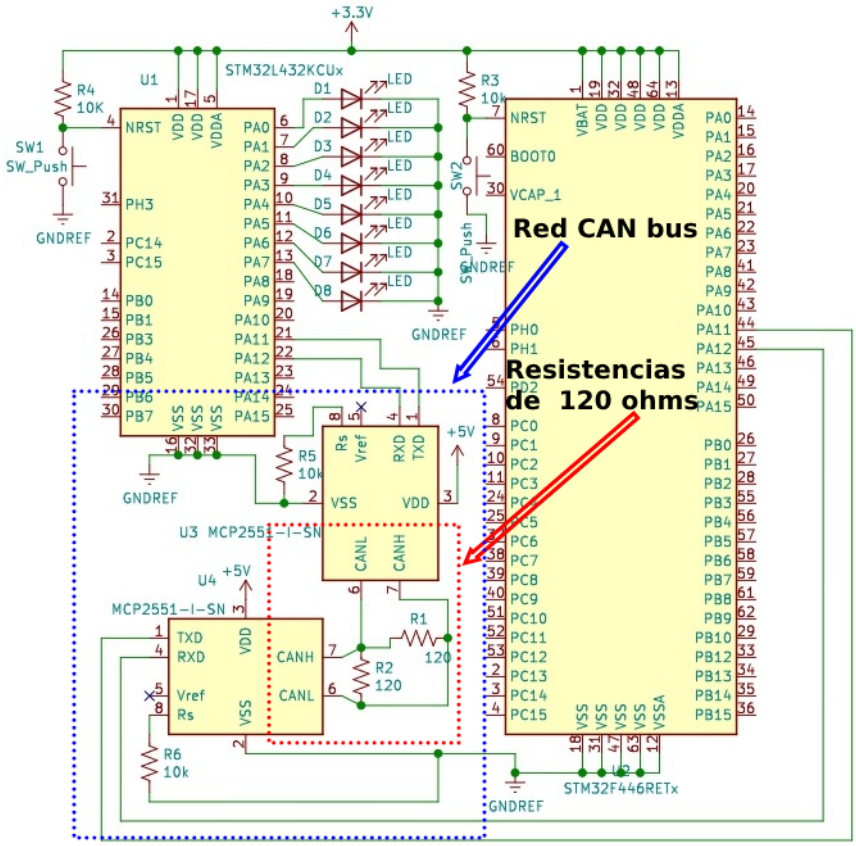


Figura 3.2: Esquemático de la conexión CAN bus entre los microcontroladores *STM32F44RE* y *STM32L432CK*.

3.2.1. Programación de la red *CAN bus*

Para programar la comunicación mediante *CAN bus*, es necesario seleccionar un lenguaje de programación, este dependerá del software que se use. En este caso se seleccionó el entorno de desarrollo *Keil*, ya que en su versión gratuita brinda los recursos necesarios para el proyecto. El algoritmo de la figura 3.3 tiene como propósito comprobar el correcto funcionamiento de la red *CAN bus* entre los microcontroladores, entonces se configura ese periférico y su respectiva interrupción en el *IDE* de *STMCUBEMX*, generando así código en lenguaje C, que posteriormente será editado en *KEIL*.

Como se muestra en la figura 3.3 se declara el *array* de 8 bytes *RXCAN*[ ] para el microcontrolador *STM32L432CK*, en este se van almacenar los datos enviados por parte del microcontrolador *STM32F446RE*, el cual transmite estos datos mediante el *array txData*[ ]. Para ambos, se debe configurar una estructura nombrada *hcan*, donde se específica una tasa de 500 kbps y es necesario configurar los *tiempo quanta* adecuados a cada microcontrolador en particular. Además de un modo normal, que

permite el envío y recepción de mensajes por *CAN bus*.

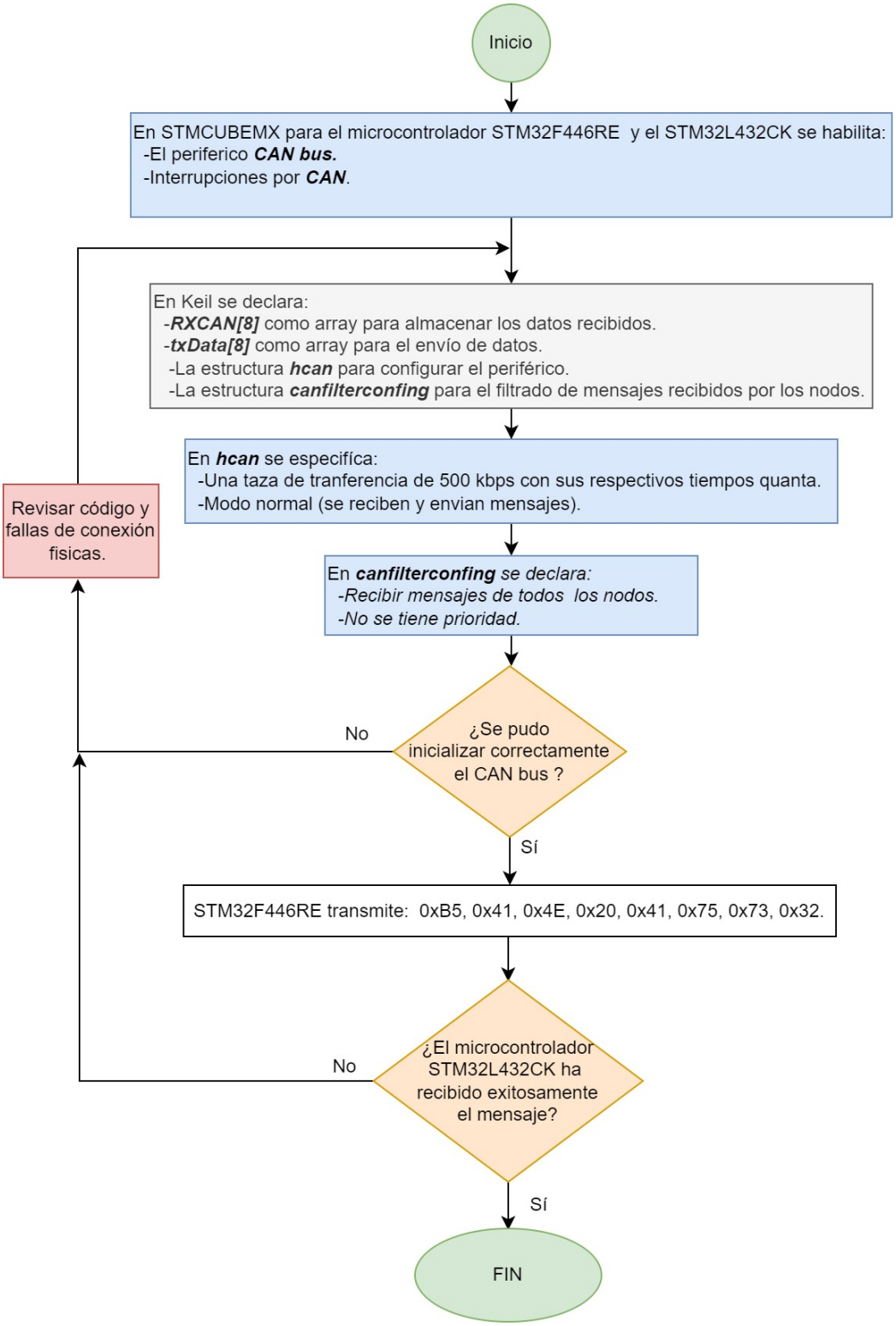


Figura 3.3: Algoritmo de prueba para establecer comunicación *CAN bus* entre dos microcontroladores *ARM-Cortex M*.

Por último se tiene la estructura *canfilterconfing*, que sirve para filtrar mensajes por *CAN bus*. Al solo ser dos dispositivos que interactúan entre sí, se configura el recibir mensajes de todos los nodos y no tener prioridad. Si el par de estructuras fueron declaradas correctamente, para cada microcontrolador, se procede a transmitir por parte del *STM32F445RE* los siguientes datos: 0xB5, 0x41, 0x4E,

0x20, 0x41, 0x75, 0x73, 0x32. Estos datos son solo para probar la comunicación y no tienen concordancia con el *OBD-II*. Si el mensaje se ha recibido exitosamente en el *STM32L432CK*, se procede a mostrar el resultado del primer elemento de *array RXCAN*[ ], en los *leds*. En caso contrario de fallar la comunicación, es posible que no se inicializo correctamente el protocolo, teniendo que revisar código y conexiones. A continuación se explica la configuración del *CAN bus* que se debe seguir y como se utiliza el software *CANCULATOR*, para los *tiempos quanta*.

De acuerdo con la *SAE J1979* (norma que establece la tasa de transferencia, conexión y los códigos requeridos para solicitar datos en el *OBD-II*), se deberá configurar una red *CAN bus* a una velocidad de 500 kbps, este proceso se explicará para el *STM32F445RE*, considerando que se realizó los mismos pasos, pero con diferentes parámetros, en el otro. *STMicroelectronics* dispone del software *STMCubeMX* que permite configurar los periféricos del microcontrolador que se le seleccione, además de generar bibliotecas para interactuar con estos. En la figura 3.4 se observa la interfaz del software, mostrándose el microcontrolador *STM32F44RETx* y del lado izquierdo se configura los parámetros de los periféricos, como son *SPI*, *RTCC* y *USART*. Por el momento, se procede a explicar el *CAN bus*, aunque en las siguientes secciones, se irá detallando cada periférico usado y su propósito.

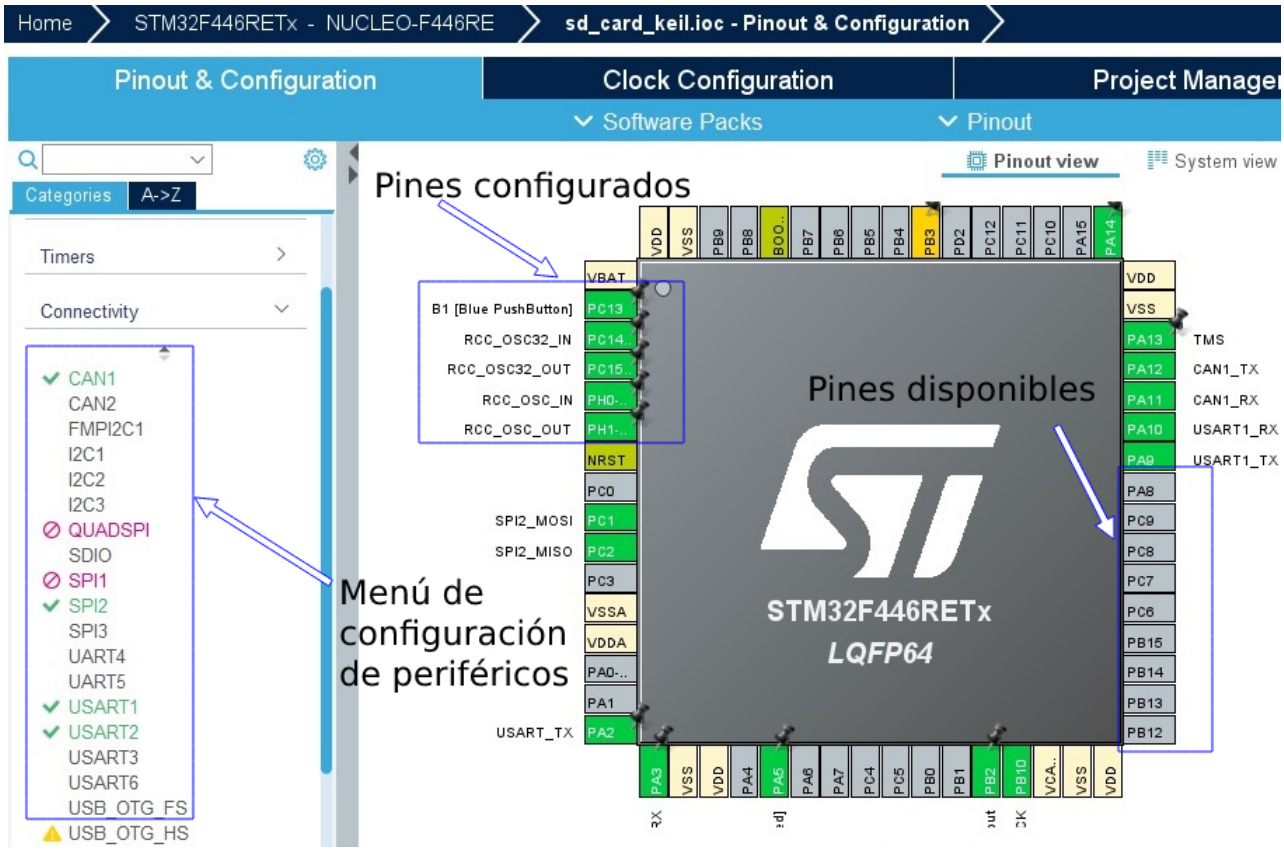


Figura 3.4: Configuración del microcontrolador *STM32F446RE* en el software *STM32CubeMX*.

Entonces como se observa en la figura 3.4, los pines en color verde son los periféricos activados para una cierta función y los que están en gris están disponibles para un uso en general. En el *CAN bus* se muestra asignados los pines *PA12* y *PA11*, esto es importante para después conectar el *transceptor CAN*. Mediante el concepto de *Bit Timming* y como se explicó en el capítulo 2, en la sección 2.24, mediante los *time quanta* necesarios, determinaremos el periodo de 1 bit y a su vez la taza del *CAN bus*, siendo esta de 500 kbps.

Es necesario para la configuración del *Bit Timming*, primero conocer la frecuencia del microcontrolador, observándose en la figura 3.5 el esquema del sistema de reloj cuyos factores de división y

multiplicación M, N, P, R y Q determinarán la frecuencia a la que operará el microcontrolador *ARM*. Por defecto se tienen configurado con los valores de M = 16, N = 336 y P = 4. En la fórmula 3.1, se muestran los cálculos realizados para obtener una frecuencia de 84 MHz que genera el *PLL* mediante la señal *PLLCLK*:

$$PLLCLK = \frac{(\frac{16MHz}{M}) * N)}{P} = \frac{(\frac{16MHz}{16}) * 336)}{4} = 84MHz \tag{3.1}$$

En el software de *STMCubeMX* ya viene configurado los parámetros para obtener esa frecuencia (84 MHz) de salida como se observa en la figura 3.6, por lo que no se tiene que hacer cambios significativos. Las flechas en color amarillo de la figura ya mencionada, muestran el camino que sigue la señal *HSI* (como se mencionó en el capítulo 2, es la señal proveniente del reloj interno del microcontrolador), hasta llegar al *APB1* con valor de 42 MHz, que es la señal de frecuencia que le llega al *CAN bus*.

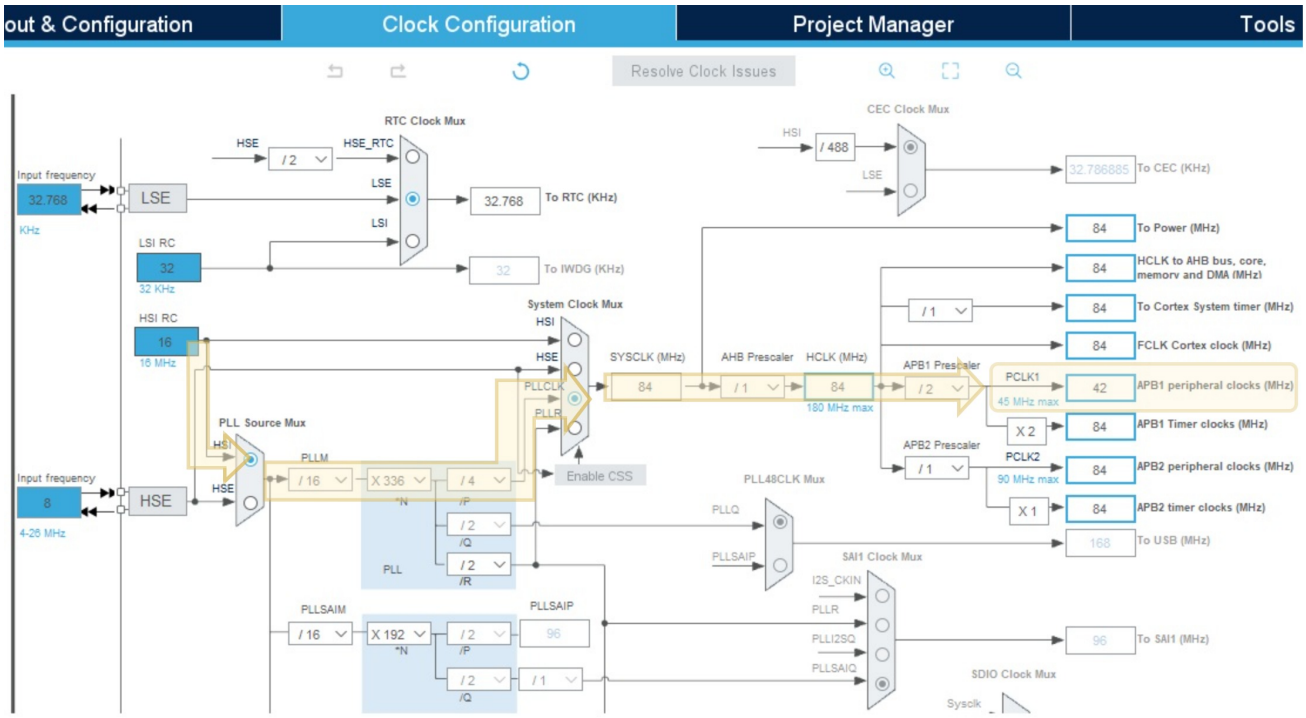


Figura 3.5: Sistema de reloj del microcontrolador *STM32F446RE* en *STM32CubeMX*.

Para la configuración del *CAN* en una velocidad de 500 kbps, es necesario utilizar un software externo llamado *CAN calculator* desarrollador por la empresa *Mikroe*. Dicho software, tiene como propósito ofrecer diversos parámetros del *Bit Timming*, como se observa en la figura 3.6 en la tabla de color verde. El usuario puede seleccionar los valores de la fila que desee para posteriormente introducirlos a *STMCUBEMX* como se muestra del lado A de la figura 3.6, sin embargo, se recomienda que escoja de las primeras filas porque tienen los valores más precisos, según lo específica el mismo software.

Para generar los anteriores parámetros se debe introducir los datos mostrados del lado B de la figura 3.6, que son: la frecuencia del *APB1* (en la figura 3.5 se muestra su valor), el *baud rate* del *CAN* (la frecuencia a la que trabaja que es 500 kbps) y los demás valores se dejan por defecto, ya que es considerando un caso ideal. Una vez que se tienen todos los periféricos configurados para su uso se procede a generar el código base en *KEIL*, para esto desde *STMCUBEMX* se selecciona en que software se va a programar, teniendo a disposición del usuario *Atollic*, *IAR*, *STMCUBEIDE*, *KEIL* y *SW4STM32*.

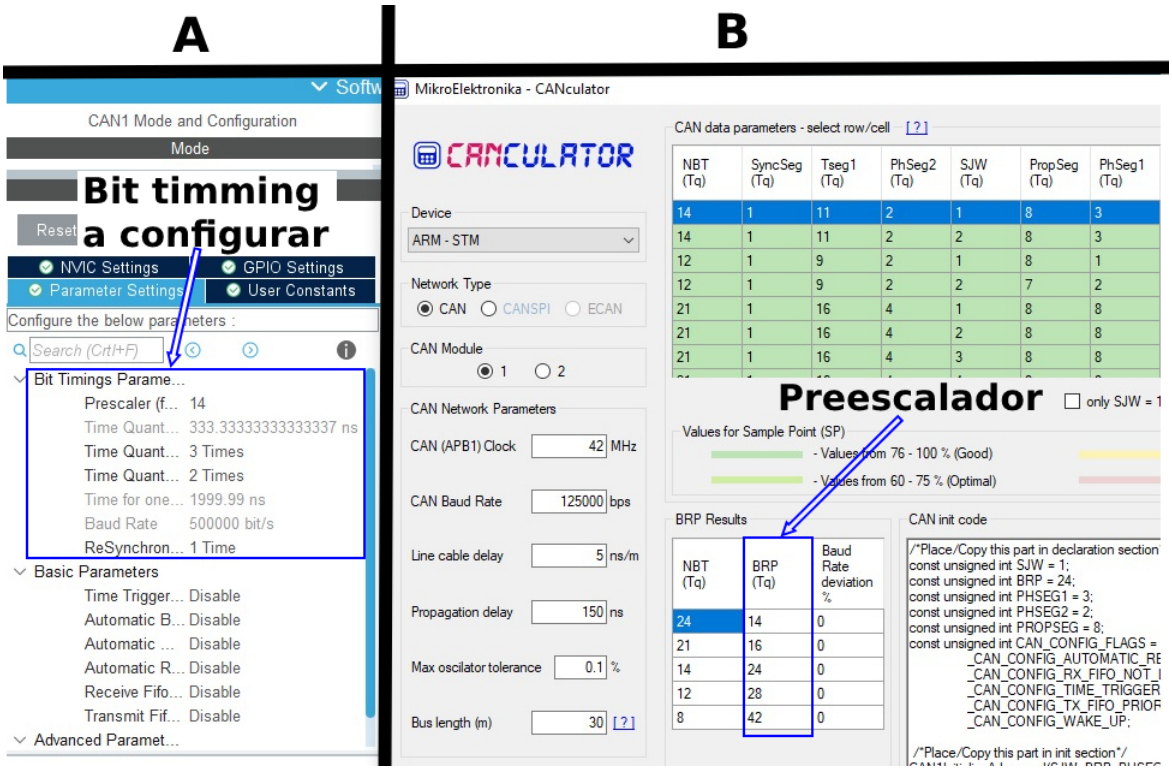


Figura 3.6: Uso del *software CAN calculator* para configurar el *Bit timming* del microcontrolador *STM32F446RE*.

Configuración de la interrupción encargada atender los mensajes recibidos por *CAN*

Al igual que los demás protocolos como *12C*, *SPI* o serial, en este también se puede configurar las interrupciones al recibir mensajes, con el objetivo, de enviar mensajes a la nube mientras se recibe o bien se espera el siguiente dato por *CAN bus*. En *STMCUBEMX* se tiene el *NVIC* que es un apartado para el vector de interrupciones, con el cual se muestran, habilitan y configuran las diferentes interrupciones del microcontrolador (ver figura 3.7).

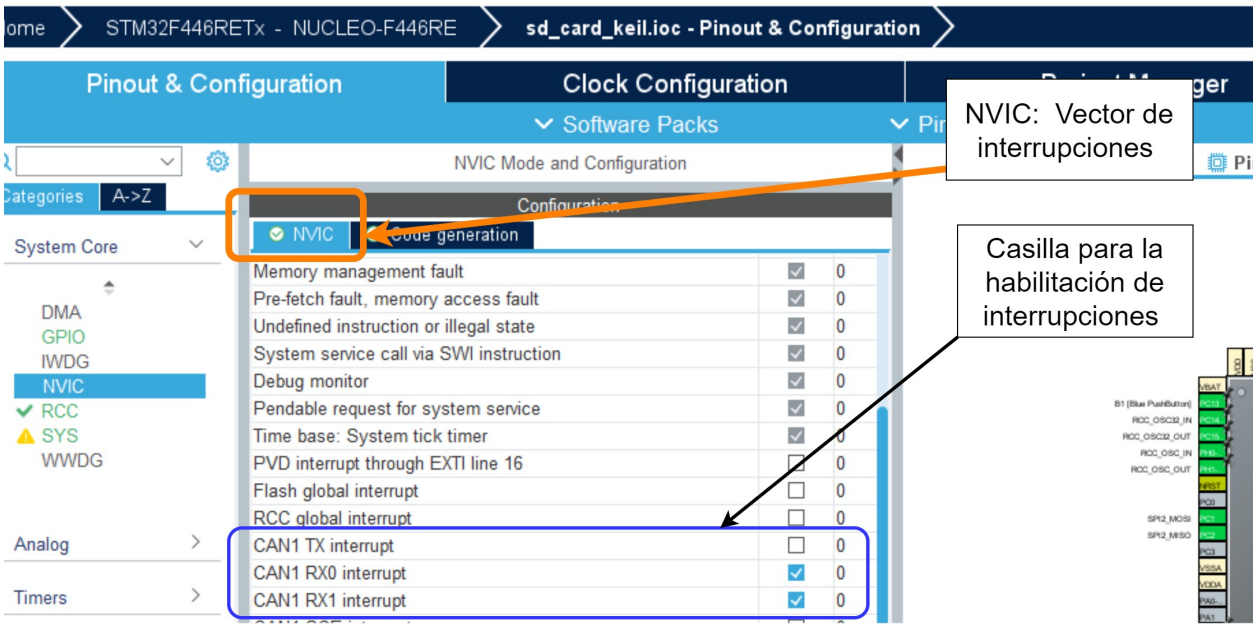


Figura 3.7: Configuración de las interrupciones del microcontrolador *STM32F446RE*.

En el caso del *CAN bus* solo se debe seleccionar la casilla con *RX0* y *RX1*, para disponer de las interrupciones (en algunos microcontroladores con un único periférico *CAN*, solo tienen la opción de *RX0*), después se deberá crear en *KEIL* como se muestra en el código 3.1, una función dedica a la atención de interrupciones. Lo importante es saber donde se almacenan los datos recibidos, que en este caso, es el *array* global *RXCAN[ ]*. En sí este *array* por sí solo no guarda los datos de inmediato, sino que se alojan momentáneamente en una *FIFO* (registro de memoria donde el primer dato en ingresar, es el primero en salir), específicamente en la *FIFO0* [81] del *STM32L432KC* y la *FIFO1* [11] del *STM32F446RE*, para luego ser almacenados en *RXCAN*. Para la prueba se compararon los datos recibidos mediante la estructura condicional múltiple, *switch*, para ir prendiendo un *led* en el puerto A, acorde a lo recibido en el tercer elemento del *array RXCAN[ ]*.

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan1){
  if (HAL_CAN_GetRxMessage(hcan1, CAN_RX_FIFO0, &RxHeader, RXCAN) == HAL_OK)
      {
    if(hcan1-> Instance == CAN1 ){
            if((RxHeader.StdId == 0x7DF)        ||
               (RxHeader.RTR == CAN_RTR_DATA)  ||
               (RxHeader.IDE == CAN_ID_STD)    ||
               (RxHeader.DLC == 8)){
              HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
              switch(RXCAN[2])
              {
               case 0x00:
                HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
                  break;
               case 0x05:
                HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET);
                  break;
                case 0x0C:
                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
                 break;
               case 0x0D:
                    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_SET);
                    break;
               case 0x0F:
              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, GPIO_PIN_SET);
                    break;
               default:
                 HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
                 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6|GPIO_PIN_7,
    GPIO_PIN_RESET);
              }
```

Código 3.1: Declaración de la función de interrupción del microcontrolador STM32L432CK.

3.2.2. Tipos de filtros *CAN bus*

Existen dos maneras de filtrar mensajes de los nodos, la primera es por medio de los registros que integra el controlador *CAN* y la segunda es mediante el condicional *if*, que nos permite especificar el *ID* del cual recibiremos los datos.

Filtrado por máscara

En los registros del controlador CAN de ambos microcontroladores usados para la prueba, tienen un *FilterId* [11] y un *FilterMaskID* [11] (que le llamaremos máscara), el primer registro indica un *ID* inicial con el que comenzará a comparar los *IDs* que reciba de los demás nodos, es decir, antes de este valor de ID no se aceptarán ningún mensaje de otros nodos en el bus. Después el *FilterMaskID* tiene dos funciones:

1. Indica el valor máximo a comparar de *IDs* en el bus, entonces para valores mayores al definido en la máscara, serán descartados.

2. También tiene como función comparar bit a bit, el *FilterId* con el valor del *ID* del bus. Si el bit establecido por la máscara es 1, forzosamente debe coincidir el bit del *FilterId* con el ID del bus para ser aceptado. En el caso de ser 0, no importa si los bits no son iguales, ese bit será considerado.

Para ejemplificar esto, se ponen dos ejemplos, con los siguientes *IDs*: 0x7D3 para el sensor de oxígeno y 0x7FF para el sensor de temperatura. Ejemplo 1: se propone un *FilterId* = 0x7D0 como se ve en la tabla 3.1, que será el valor inicial a comparar, después la máscara (en la tabla esta descrito como *Mask*, ya que así se encuentra en el *datasheet* de ambos microcontroladores) como 0xFF0. Entonces los 4 bits menos significativos del *FilterId* aunque no coincidan los del sensor de oxígeno, serán aceptados, mientras que los restantes si deben ser iguales.

Tabla 3.1: Ejemplo 1 del uso de máscaras donde se acepta el *ID* del bus.

	Hexadecimal	**Binario**	**Comentario**
ID del bus	0x7D3	0111 1101 0011	Oxígeno
FilterId	0x7D0	0111 1101 0000	Se define el valor inicial del ID
Mask	0xFF0	1111 1111 0000	Se define un rango (0x7D0 al 0x7DF)
Resultado	0x7D3	0111 1101 0011	ID aceptado

Para el ejemplo 2, compararemos el sensor de temperatura como se ve en la tabla 3.2, entonces los 4 bits menos significativos, son aceptados sin importar si su valor coincide, pero los siguientes al no coincidir en un bit, todo el *ID* es descartado.

Tabla 3.2: Ejemplo 2 del uso de máscaras, donde se rechaza el *ID* del bus.

	Hexadecimal	**Binario**	**Comentario**
ID del bus	0x7FF	0111 1111 1111	Temperatura
FilterId	0x7D0	0111 1101 0000	Se define el valor inicial del ID
Mask	0xFF0	1111 1111 0000	Se define un rango (0x7D0 al 0x7DF)
Resultado	0x7D0	0111 11X1 1111	ID no aceptado

Se tiene una configuración de *FilterId* = 0x0000, para aceptar el valor mínimo de ID de cualquier nodo y aunque este no coincida con el valor del ID del bus, en la máscara tiene igual el valor 0x0000, dejando pasar cualquier identificador.

Filtrado mediante la función de interrupción haciendo uso de condicionales

Como se deja pasar cualquier *ID* mediante máscaras, se recurre a otro método para filtrar los mensajes del *OBD-II*, recurriendo a la estructura condicional *if*, como se observa en el código 3.2, donde primero se pasan los datos del registro *FIFO0* al *array RXCAN*, si este proceso ha sido exitoso, entonces en la estructura *hcan* por medio del condicional *if*, acepta únicamente los mensajes con el identificador 0x7DF (línea 5 del código 3.1), después en la línea 6, se válida que la trama recibida no sea remota (ver sección 2.2.4, en el apartado ***Trama remota***), añadido a esto, la trama debe ser estándar (11 bits de identificador) y además se esperan 8 bytes de datos (esto al comparar en la línea 8 el *DLC* == 8). Si todo lo anterior se cumple, consecuentemente se aceptará el campo de datos.

```
1
2  if (HAL_CAN_GetRxMessage(hcan1, CAN_RX_FIFO0, &RxHeader, RXCAN) == HAL_OK)
3        {
4      if(hcan-> Instance == CAN1 )
5         {if((RxHeader.StdId == 0x7DF)         ||
6                  (RxHeader.RTR == CAN_RTR_DATA)  ||
7                  (RxHeader.IDE == CAN_ID_STD)    ||
8                  (RxHeader.DLC == 8))
9           {
10             switch(RXCAN[2])
```

Código 3.2: Fragmento de código del microcontrolador *STM32L432CK*, referente al filtrado de *IDs* mediante condicionales.

3.2.3. Prueba realizada para confirmar la comunicación *CAN*

El microcontrolador *STM32F446RE*, que funge como maestro dentro de la prueba realizada, transmitirá lo siguiente: el DLC = 8, una trama estándar, no remota, el identificador igual a 0x7E8 y el código de *CRC* es generado por el controlador *CAN*. Eso está declarado en el código 3.3, donde *TxHeader* es una estructura que permite establecer los parámetros de la trama.

```
1
2   TxHeader.DLC = 8;  // 8 Bytes a enviar
3   TxHeader.IDE = CAN_ID_STD;
4   TxHeader.RTR = CAN_RTR_DATA;
5   TxHeader.StdId = 0x7E8;  // Identificador
```

Código 3.3: Parámetros de la trama *CAN bus* del microcontrolador *STM32F44ER6*.

En el código 3.4 se observa el *array txData* que almacena los siguientes datos: 0xB5, 0x41, 0x4E, 0x20, 0x41, 0x75, 0x73, 0x32. Por último en la línea 20 del código 3.3, obsérvese como los valores de *TxHeader* y *txdata*, son pasados a la estructura *hcan*, por medio de la función *HAL_CAN_AddTxMessage*, estableciendo así la trama completa a transmitir.

```
1  while (1)
2    {if(HAL_GPIO_ReadPin(Botton_GPIO_Port,Botton_Pin) == 0)
3     {  txData[0] = 0xB5;
4      txData[1] = 0x41;
5      txData[2] = 0x43;
6      txData[3] = 0x20;
7      txData[4] = 0x42;
8      txData[5] = 0x75;
9      txData[6] = 0x73;
10     txData[7] = 0x32;
11   HAL_RetVal = HAL_CAN_AddTxMessage(&hcan, &TxHeader, txData, &TxMailBox);
```

Código 3.4: Datos a transmitir por parte del microcontrolador *STM32F44ER6*, para la prueba.

En la figura 3.8, obsérvese una captura de los datos enviados por *CAN bus* en el osciloscopio *MDO4104-6* de la marca *Tektronic*, con el fin de corroborar el correcto funcionamiento del bus. Cabe aclarar que los datos mostrados en el osciloscopio son en formato hexadecimal, además que podemos apreciar el código de *CRC*, cuyo valor es 0x719. El prototipo completo se muestra en la figura 3.9, donde la tarjeta *SD* y el moduló *4G/LTE* son controlados por el nodo 1, que corresponde al microcontrolador *STM32F446RE* que servirá para realizar pruebas destinadas a sistema empotrado y el nodo 2 por su parte corresponderá a la emulación del sistema *OBD-II*.

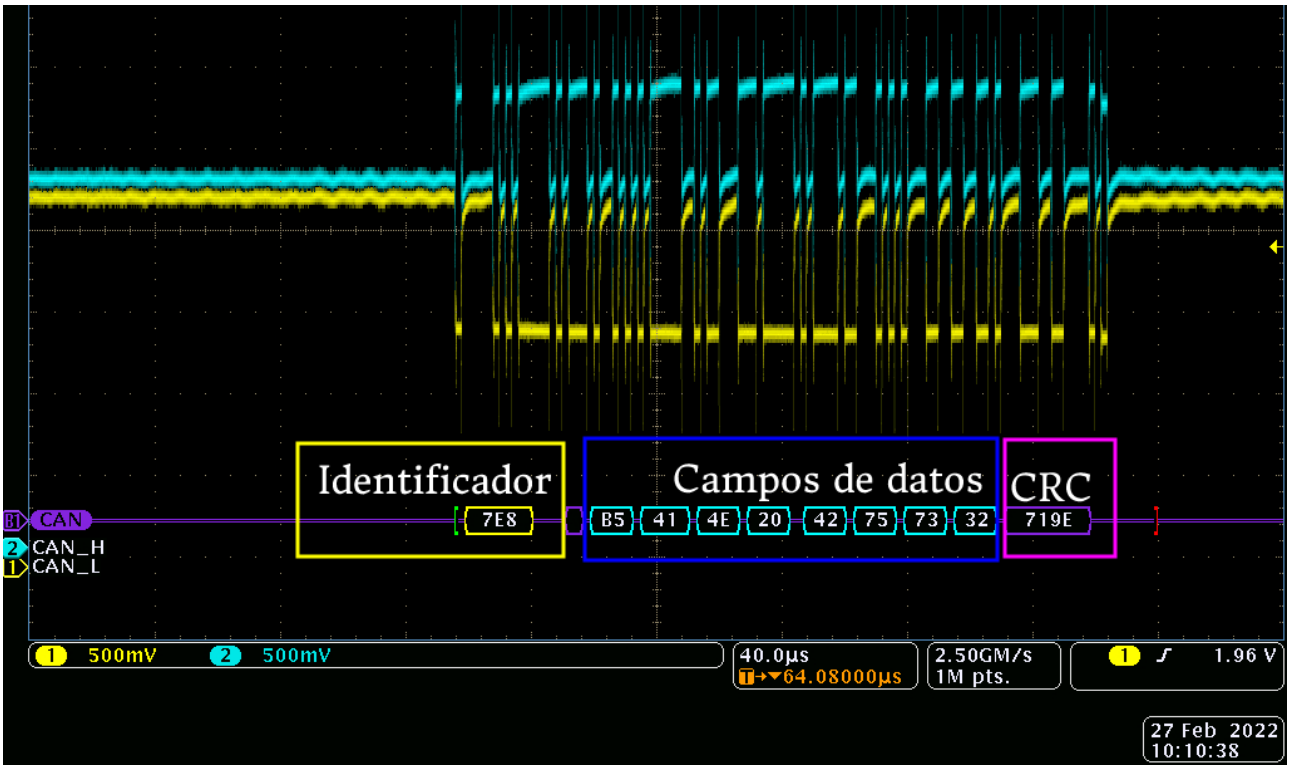


Figura 3.8: Captura de la trama enviada.

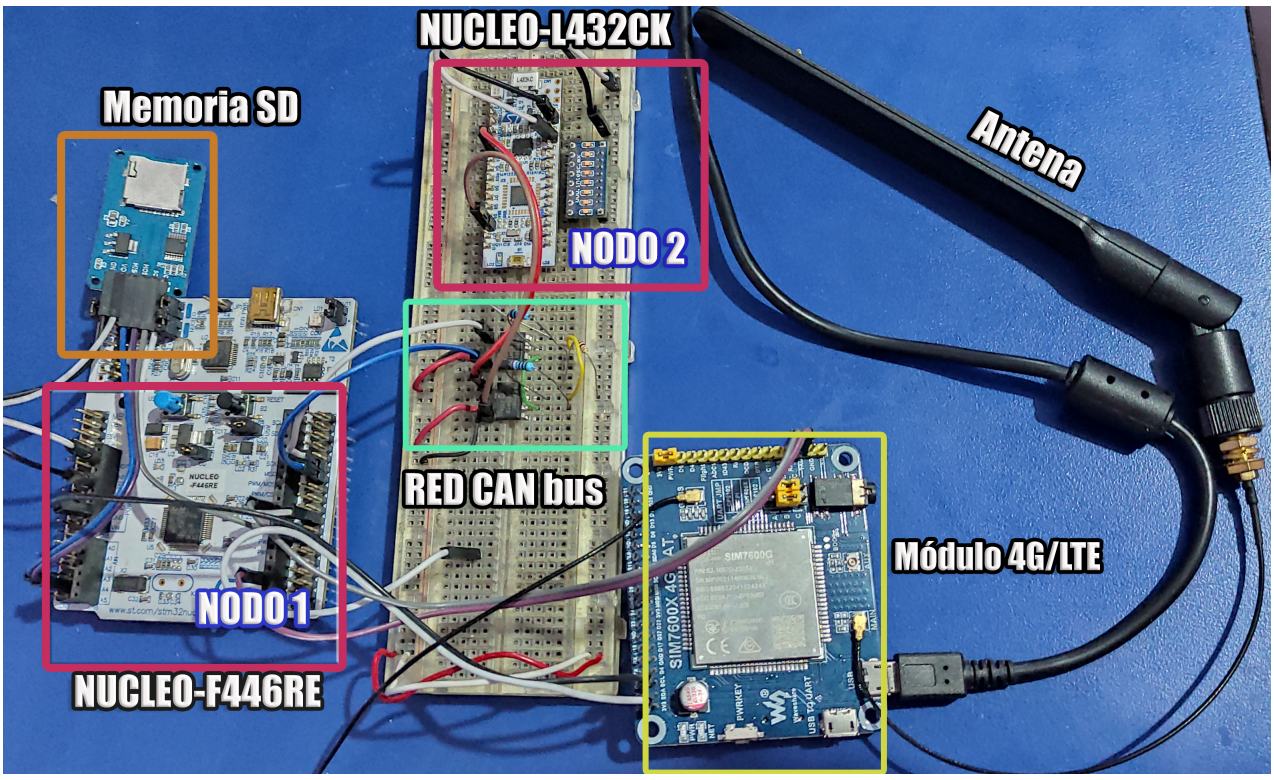


Figura 3.9: Vista frontal del prototipo físico.

3.3. Emulación del *OBD-II*

Para el desarrollo del SADO, resulta factible tener un emulador de *OBD-II*, que no consuma gasolina (permitiendo así realizar pruebas por varias horas) y ocupe un espacio reducido en comparación al de un automóvil. Para realizar este emulador, se utilizó como referencia la *SAEJ1979* que muestra los modos que tiene el *OBD-II*, donde el modo 1 tiene los códigos *PIDs* (mostrados en la tabla 2.1 de la sección 2.2.4), cuyo valor se puede solicitar mediante el formato de trama establecida por la *SAEJ1939* [14]. En las siguientes secciones se muestran la metodología usada para la solicitud y recepción de *PIDs* del *OBD-II*.

3.3.1. Algoritmo y código de solicitud de datos del *OBD-II*

Entonces el SADO debe enviar en su trama un identificador con valor 0x7DF, DLC = 0x08 (para transmitir 8 bytes) y para el campo de datos, el primer byte, debe especificar el número de bytes adicionales, es decir, cuantos bytes después de este serán tomados en cuenta, siendo siempre 2, los cuales son: el modo (que en este caso es 01) y el número de *PID* que se quiere consultar. Los demás datos pueden ser con valor 0x00 o 0x55, como se muestra en la figura 3.10.

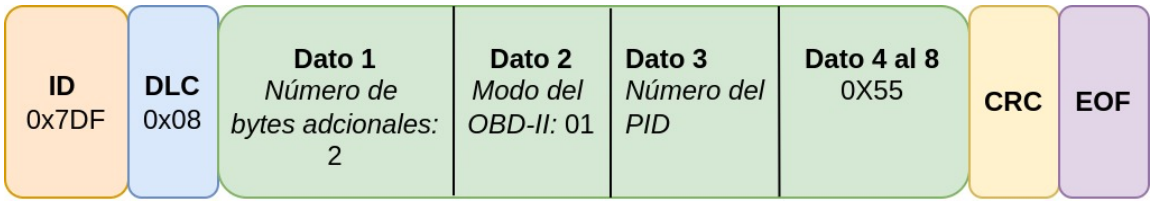


Figura 3.10: Trama de datos enviados desde el prototipo hacia el emulador del *OBD-II* [10].

En el algoritmo de la figura 3.11 se muestra la función de obtención de datos del OBD-II, donde al SADO se le configura un pulsador, declarado como *Botton*, el cual al oprimirse, comenzará por solicitar los datos de *geolocalización* que brinda el *GPS*, después de manera secuencial, empezará a solicitar primero los *PIDs* implementados, para esto se enviará en su campo de datos, en el primer byte, el valor 0x02, después el modo, 0x01 (segundo byte) y por último el *PID* (tercer byte) que es el 0x00, como se observa en la figura 3.10. Los demás bytes restantes tendrán el valor 0x55. Una vez transmitida esta trama, se espera mediante la función de interrupción la respuesta por parte del emulador, el cual contestará con el mismo *PID* solicitado, esto servirá para identificar el siguiente *PID* a pedir, para ello este valor se guarda en la variable *CAN_RXBUS*.

Por ejemplo cuando se solicita la temperatura del motor, si la transmisión fue exitosa, el emulador enviará los datos solicitados y responderá con el mismo *PID* solicitado, siendo 0x05, entonces el prototipo entrará en la función de interrupción guardando este valor en CAN_RXBUS y al compararlo como se muestra en la figura 3.11, se determina que el siguiente dato a enviar será las *RPMs*. Esto continuará así hasta que se reciba el valor del último sensor.

Cabe aclarar que la *función de almacenamiento en memoria SD* (guarda los valores en la memoria SD) y la *función de envío de datos a la nube* (envía los datos obtenidos del emulador del OBD-II a la base de datos en). La codificación de este algoritmo se encuentra en el código 3.5, donde la función $HAL\_CAN\_AddTxMessage(hcan, TxHeader, txData, TxMailBox)$; es la encargada de transmitir los valores de *txData* por *CAN bus*. Mediante la función *myprintf* se envía al *PC* (por medio de la *USART*) los datos y cadenas de texto que ayudan a *monitorear* lo que suceda durante la ejecución del algoritmo de la figura 3.11, visualizándose en una terminal serial, como *TeraTerm*.

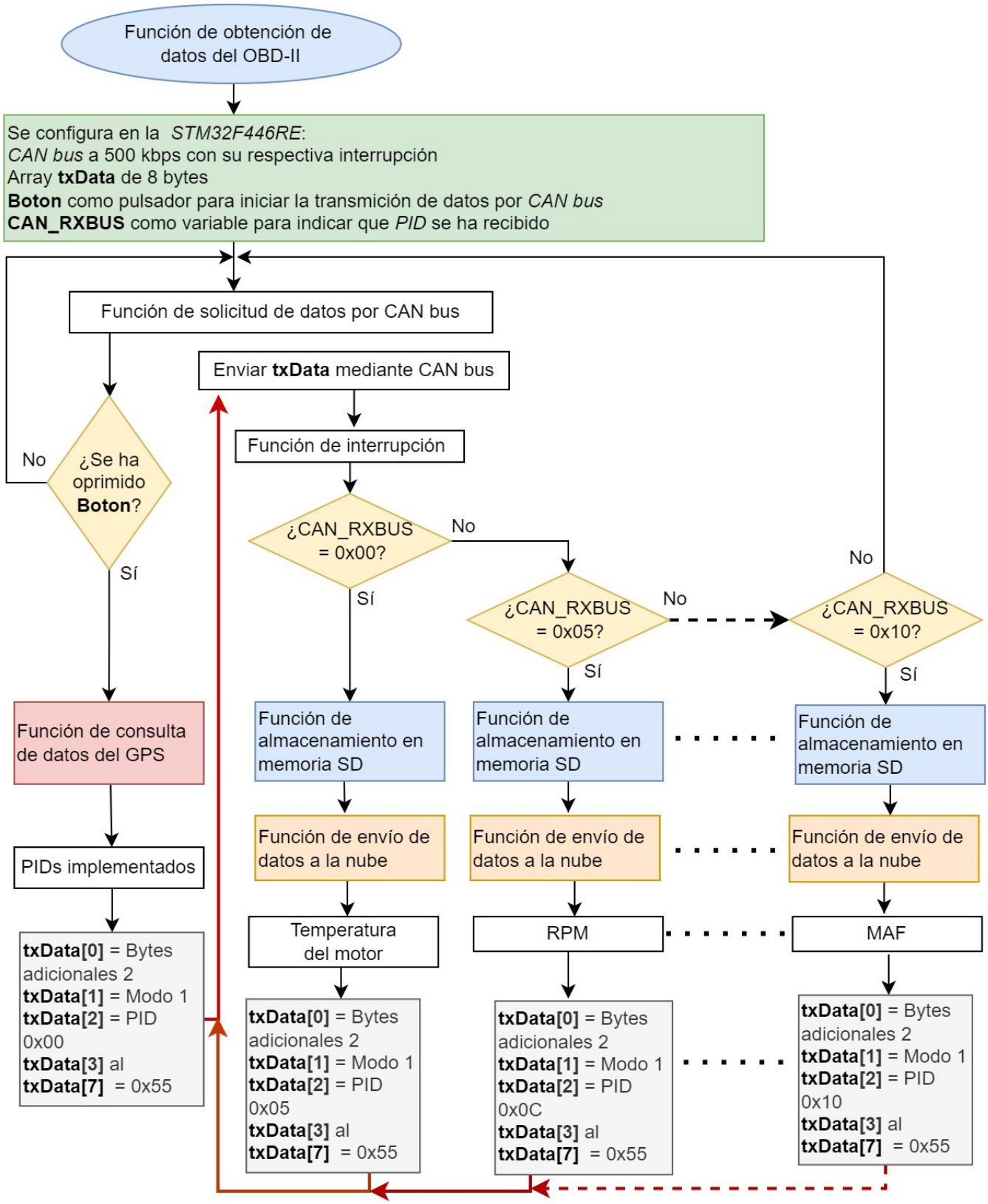


Figura 3.11: Algoritmo de solicitud de parámetros del *OBD-II*.

```
while (1)
  {
    if(HAL_GPIO_ReadPin(Botton_GPIO_Port,Botton_Pin) == 0)
    {
```

```
    myprintf("Consulta de los datos del GPS\n\r");
    SIMTransmit("ATE0\r\n");
    SIMTransmit("AT+CGPSINFO\r\n");
    conta_caracteres = 0;
        .
        .
        .
    myprintf("%s\n\r",cadena);

    /* Consulta de los PIDs por CAN bus*/
    txData[0] = 0x02; //Bytes adiccionales
    txData[1] = 0x01; //Modo 01
    txData[2] = 0x00; // PID 0x00
  //PIDs implementados [01 - 20]
 HAL_RetVal = HAL_CAN_AddTxMessage(&hcan, &TxHeader, txData, &TxMailBox);
  //while(HAL_CAN_IsTxMessagePending(&hcan1, (uint32_t)CAN_TX_MAILBOX0));
    myprintf("Consulta de PIDS\n\r");
      while(datos_CAN_recibidos == 1)
      { httpPost(); }
        Guardar_datos_en_SD();
  //Temp. del liquido de enfriamiento del motor
    txData[2] = 0x05;
    HAL_RetVal = HAL_CAN_AddTxMessage(&hcan, &TxHeader, txData, &TxMailBox);
    myprintf("Temp del liquido de enfriamiento del motor\n\r");
      while(CAN_RXBUS == 0x00){}
        httpPost();
        Guardar_datos_en_SD();
              .
              .
              .
  //MAF
    txData[2] = 0x10;
    HAL_RetVal = HAL_CAN_AddTxMessage(&hcan, &TxHeader, txData, &TxMailBox);
    myprintf("MAF\n\r");
    HAL_Delay(1000);
    //while(CAN_RXBUS == 0x0F){}
        httpPost();
        Guardar_datos_en_SD();
        HAL_Delay(500);
       HAL_GPIO_TogglePin(GPIOA,LED_Pin);
       }
```

Código 3.5: Fragmento de código correspondiente a la consulta del valor actual de los sensores del *OBD-II*.

3.3.2. Algoritmo y código de transmisión de datos del emulador del *OBD-II*

Una vez enviada la trama de la figura 3.10 con los valores del *PID* a consultar, se recibirá la trama de la figura 3.12, donde el identificador es 0x7E8, *DLC* = 0x08 y en el campo de datos, en dato 1, indica el número de bytes adicionales a tomar en cuenta, pudiendo ir de 3 a 6, esto dependerá del valor solicitado, ya que primero se recibe el modo 0x01, después el *PID* solicitado y de aquí pueden ser 1 byte hasta 4 entregados, por ejemplo, si se solicita el *PID* 0x00, el cual indica cuales valor se pueden consultar en el modelo de vehículo, se espera recibir 4 bytes, por lo que el número de bytes adicionales serán 6. Pero si se solicita el *PID* de las *RPMs*, cuyo valor es 0x0C, entonces se espera que sean 4 bytes adicionales, ya que sería el modo, *PID* y añadido a esto los dos datos a los cuales se les tendría que aplicar una fórmula para obtener su valor en las unidades correspondientes de acuerdo a la tabla 2.1. Igual que en la anterior trama de la figura 3.10, en esta el *CRC* y *EOF* son generados por el controlador *CAN* del emulador del *OBD-II*.

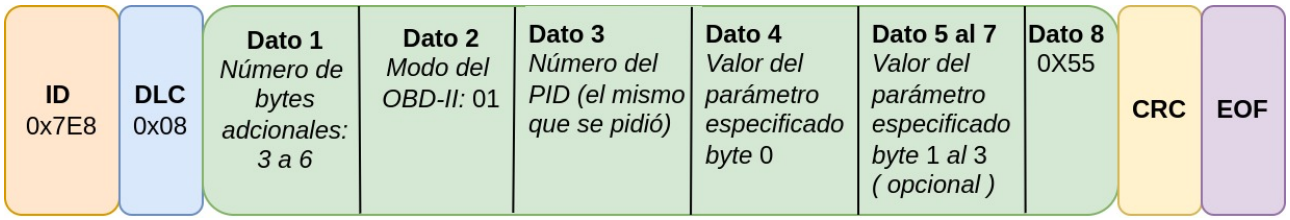


Figura 3.12: Trama de datos enviados desde el emulador hacia el SADO [10].

Entonces el *STM32L432CK* tendrá el algoritmo de la figura 3.13, configurando primero un *CAN bus* a 500 kbps, activando su correspondiente interrupción y un *array* declarado como **Datos** con tamaño de 8 bytes. Al recibir un mensaje por *CAN*, se activa la función de interrupción y como se explicaba en la sección 3.2.2, se tiene un filtrado de mensajes, donde el identificador debe ser 0x7DF, en caso de no ser así, el microcontrolador descartara la trama. En caso de ser el correcto, se va identificando que valor *PID* corresponde, por ejemplo en el caso de la temperatura del motor, en el *array* **Datos**, almacena en su primer byte, el valor 0x03, especificando así, que se transmitirán 3 bytes adicionales, después en su segundo byte, el valor 0x01, indicando el modo 1, para luego en su tercer byte guardar el valor 0x05 que corresponde al valor solicitado del *PID*, según lo establecido por la tabla 2.1 y en el cuarto byte el valor simulado del sensor (este es un valor fijo, solamente con la intención de hacer pruebas). Los demás bytes del 4 al 7, tienen el valor 0x00.

La codificación del algoritmo 3.13, se muestra en el código 3.6, donde los datos recibidos en CAN_RX_FIFO0 (que es el registro *FIFO0*), se pasan al *array RXCAN* y mediante la estructura condicional múltiple *switch*, se va seleccionando que *PID* se solicita y por medio de la función $HAL\_CAN\_AddTxMessage$ se transmite por *CAN bus* el *array* **Datos**, que contiene los parámetros correspondientes. Después se hace conmutar un *LED*, para así indicar que se ha enviado exitosamente los datos solicitados. Se aclara que el emulador del *OBD-II* no tiene ningún código en la función principal *int main*, por lo que estará en todo momento a la espera de recibir un mensaje.

```
void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
{if (HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, RXCAN) == HAL_OK)
      {if(hcan-> Instance == CAN1 )
       {if((RxHeader.StdId == 0x7DF)        ||
           (RxHeader.RTR == CAN_RTR_DATA)  ||
           (RxHeader.IDE == CAN_ID_STD)    ||
           (RxHeader.DLC == 8))
         {switch(RXCAN[2])
           {
           case 0x00: //PIDs implementados [01 - 20]
               Datos[0] = 0x06;
               Datos[2] = 0x00; //PID
               Datos[3] = 0xBE;
               Datos[4] = 0x1F;
               Datos[5] = 0xEC;
               Datos[6] = 0x13;
               HAL_CAN_AddTxMessage(&hcan, &TxHeader, Datos, &TxMailBox);
               HAL_GPIO_TogglePin(LED0_GPIO_Port, LED0_Pin);
               HAL_Delay(100);
               break;
                                    .
                                    .
                                    .
               default:  //MAF
                     Datos[0] = 0x04;
               Datos[2] = 0x10; //PID
               Datos[3] = 0x67;
               Datos[4] = 0xAA;
               HAL_CAN_AddTxMessage(&hcan, &TxHeader, Datos, &TxMailBox);
```

```
30                    HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
31                    HAL_Delay(100);}
32  /* USER CODE END 4 */
```

Código 3.6: Fragmento de código correspondiente a la función de interrupción del emulador del *OBD-II*.

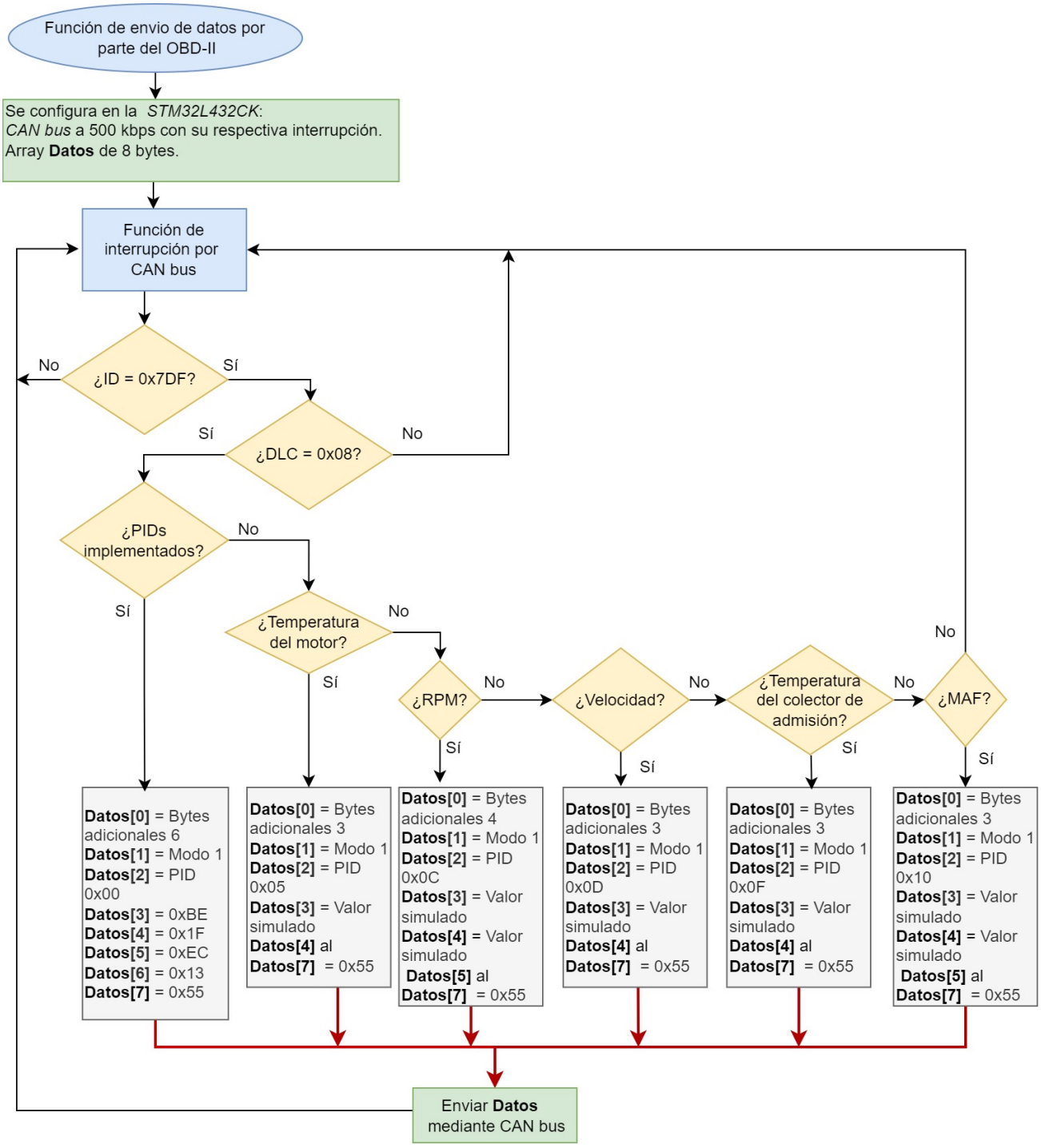


Figura 3.13: Algoritmo del emulador *OBD-II*.

3.3.3. Resultados de la comunicación entre el prototipo y el emulador del *OBD-II*

En la figura 3.14 se observa en el osciloscopio la trama enviada por el SADO, donde se visualiza el identificador con valor 0x7DF, $DLC = 0x08$ y en el campo de datos se tiene los valores correspondien-

tes para hacer la solicitud del *PID* 0, el cual indica que parámetros se pueden consultar del vehículo. Cabe aclarar que estos valores son ficticios y solo son con fines de probar el funcionamiento del firmware.

A los 10 ms de haber enviado la trama por parte del SADO, se tiene como respuesta del emulador (ver figura 3.15) un ID = 0x7E8, DLC = 0x08 y en el campo de datos, en el primer byte, indica los bytes adicionales (siendo 6 en este caso), después el modo que es 0x01, en el tercer byte el *PID* solicitado, que es 0x00 y los consiguientes bytes son los valores 0xBE, 0x1F, 0xEC, 0x13 y el octavo byte tiene valor 0x55. Entonces nuevamente el prototipo enviará otro mensaje para solicitar los 5 valores de sensores a utilizar y por parte del emulador, responderá con los correspondientes datos solicitados.

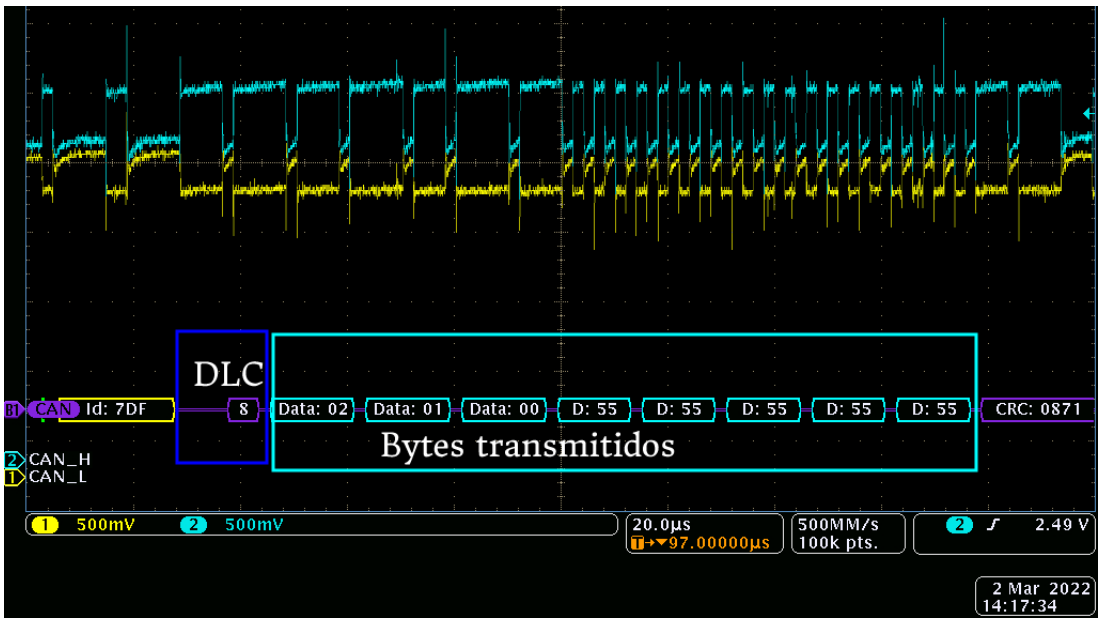


Figura 3.14: Visualización en el osciloscopio de la trama de datos *CAN bus* transmitida por parte del SADO.

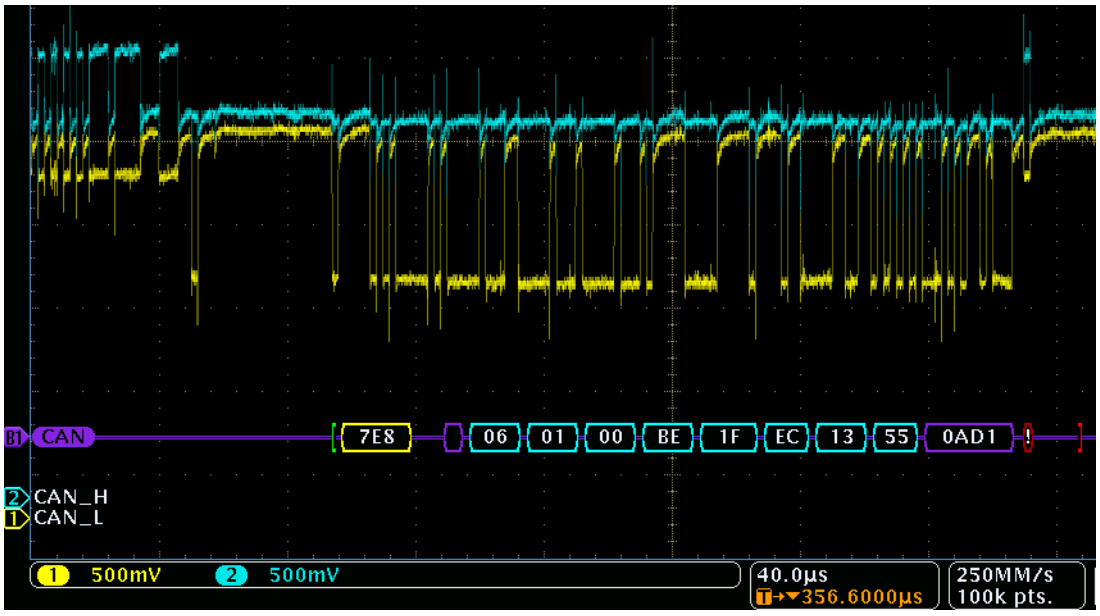


Figura 3.15: Visualización en el osciloscopio de la trama de datos *CAN bus* transmitida por parte del emulador del *OBD-II*.

3.3.4. Justificación de los 5 sensores seleccionados para realizar pruebas en un automóvil real

Previamente en el laboratorio *SLED* (ubicado en la Facultad de Ciencias de la Electrónica, donde se realizan investigaciones del ámbito automotriz), se hicieron pruebas en el año 2019 en un *Jetta Hybrid*, donde se entabló comunicación por medio del *OBD-II* a este vehículo. En la tabla 3.3 se muestran los sensores que se pueden consultar en este modelo de automóvil, sirviendo de referencia para escoger entre ellos 5, aunque esta selección no da garantía que en otro vehículo de la misma marca se obtenga esos mismos sensores.

Tabla 3.3: *PIDs* que se pueden consultar en el *Jetta Hybrid*.

Datos en hexadecimal de respuesta al consultar el PID 0	**Datos en binario de respuesta al consultar el PID 0**	**PID (hex)**	**Descripción**	**¿Se puede consultar?**
		0	PIDs implementados [01 - 20]	**Si**
	A0 = 1	1	Estado de los monitores de diagnóstico desde que se borraron los códigos de fallas DTC	**Si**
	A1 = 0	2	Almacena los códigos de fallas de diagnóstico DTC de un evento	**No**
	A2 = 1	3	Estado del sistema de combustible	**Si**
	A3 = 1	4	Carga calculada del motor	**Si**
BE	A4 = 1	5	Temperatura del líquido de enfriamiento del motor	**Si**
		. . .		
	B0 = 0	9	Ajuste de combustible a largo plazo—Banco 2	**No**
	B1 = 0	0A	Presión del combustible	**No**
	B2 = 0	0B	Presión absoluta del colector de admisión	**No**
	B3 = 1	0C	RPM del motor	**Si**
	B4 = 1	0D	Velocidad del vehículo	**Si**
	B5 = 1	0E	Avance del tiempo	**Si**
	B6 = 1	0F	Temperatura del aire del colector de admisión	**Si**
1F	B7 = 1	10	Velocidad del flujo del aire MAF	**Si**
EC	C0 = 1	11	Posición del acelerador	**Si**
		. . .		
		. . .		
13	D7 = 1	20	PID implementados [21 - 40]	**Si**

En la tabla 3.3 se aprecia alguno de los sensores que se pueden acceder en el *Jetta Hybrid*, teniendo

bien identificados que *PIDs* se solicitarán en pruebas posteriores. Si bien se pueden hacer pruebas en otros modelos de vehículos como *FORD*, *BMW*, etcétera, se siguiere realizarlas en el *Jetta Hybrid*, ya que es un automóvil destinado a pruebas en laboratorio, además considérese que la *SAE J1939*, indica que no todas las empresas automotrices están obligadas a implementar los códigos *PIDs* y modos e inclusive pueden añadir los suyos, siendo privados con uso solamente de la compañía. Añadido a esto, en el tema de tesis *IDENTIFICACIÓN Y CONTROL DE PARÁMETROS DE CLÚSTER DE INSTRUMENTOS AUTOMOTRIZ MEDIANTE RED CAN* [82] de la *Lic. en Ingeniería en Sistemas Automotrices Lady Guadalupe Feliciano Fuentes*, describe las pruebas realizadas para obtener los parámetros de un cuadro de instrumentación del modelo de automóvil *Seat Ibiza*, mostrando algunos de esos valores en la tabla 3.4:

	Identificador		**Byte específico donde esta el valor del parámetro**	
Parámetro	Cuadro de instrumentos 6J0 920801X de un vehículo Seat Ibiza	OBD-II	Cuadro de instrumentos 6J0 920801X de un vehículo Seat Ibiza	OBD-II
RPM	0x280	0x7E8	4	4 y 5
Velocidad	0x5A0	0x7E8	3	4
Luces (MIL)	0x288	0x7E8	1	4 y 5

Tabla 3.4: Comparativa del formato de datos obtenidos en el cuadro de instrumentos de un *Seat Ibiza* en contraste con los del *OBD-II*.

Nótese como el identificador no es el mismo que en el *OBD-II*, además que el byte donde específica el valor del parámetro no concuerda, entonces hacer pruebas en otros vehículos, no resulta factible y conlleva un tiempo más largo para identificar que *PIDs* se pueden solicitar. En conclusión, abajo se muestran los 5 valores a consultar con sus respectivas unidades y fórmulas:

1. Velocidad del flujo del aire *MAF*:

$$MAF[\frac{gr}{sec}] = \frac{(256 * A) + B}{100} \tag{3.2}$$

2. Temperatura del líquido de enfriamiento del motor:

$$Temp\_Motor[^{\circ}C] = A - 40 \tag{3.3}$$

3. *RPM* del motor:

$$RPM[rpm] = \frac{(256 * A) + B}{4} \tag{3.4}$$

4. Velocidad del vehículo:

$$Vel[\frac{km}{h}] = A \tag{3.5}$$

5. Temperatura del aire del colector de admisión:

$$Temp\_Colector\_Ad[^{\circ}C] = A - 40 \tag{3.6}$$

3.4. Almacenamiento local de los datos mediante una memoria *SD*

Se puede dar el caso donde no se disponga de conexión a internet, llegando a requerir guardar los datos obtenidos por el *OBD-II* en una memoria *SD*, para posteriormente enviarlos a la base de datos cuando se restablezca conexión de nuevo. Entonces el SADO tiene integrado una comunicación con una memoria *SD* de 16 GB, sin embargo solo se está utilizando 4 GB, esto ya que la memoria

tiene un formato *FAT32* [80] que solo permite controlar esta cantidad de *GigaBytes*. Entonces como se observa en la figura 3.16, se tiene esta opción habilitada *FATFS* en la sección de *Middleware*, brindando múltiples opciones para controlar la memoria. La codificación es en *UTF-8* (aunque se pueden seleccionar otras opciones como *UTF-16BE*, *UTF-16LE* ó *ANSI*), ya que es el más común y es compatible con *ASCII* (es un código de caracteres basado en el alfabeto latino).

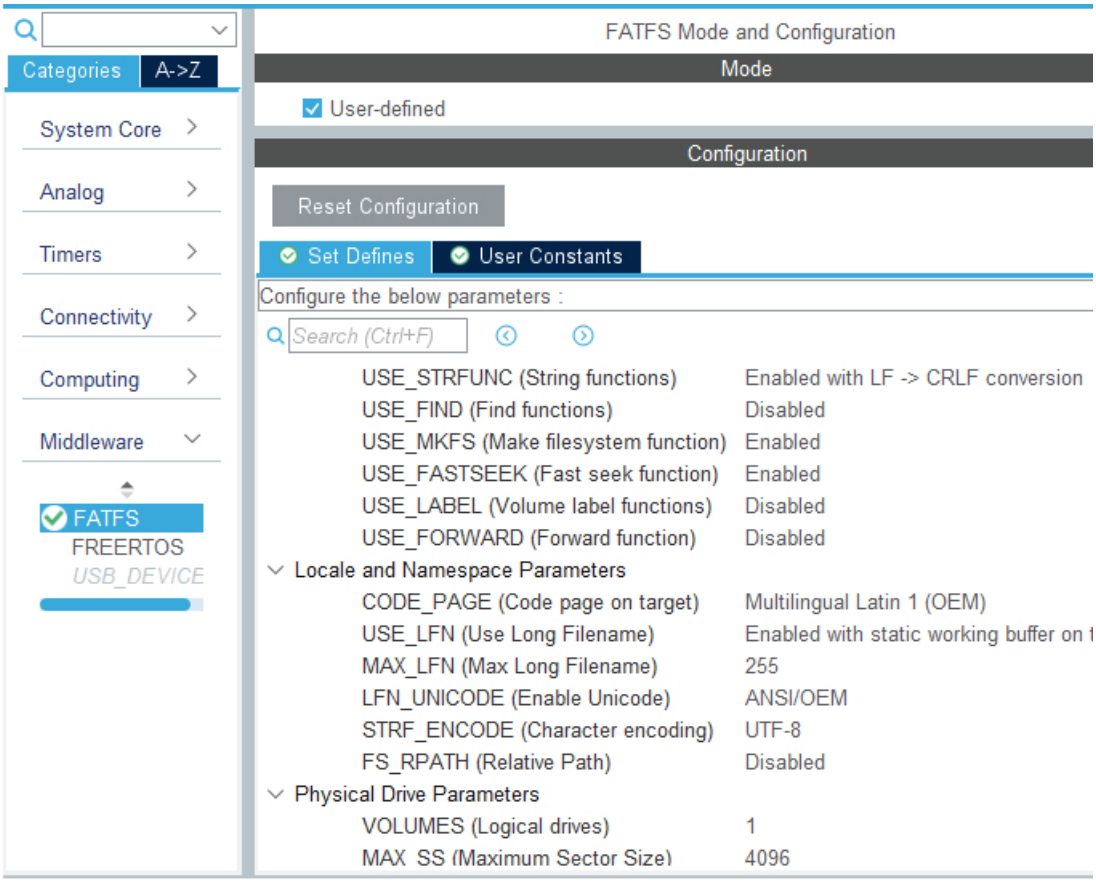


Figura 3.16: Generación de la biblioteca *FATFS* para controlar memorias con formato *FAT32*.

Para que la anterior biblioteca funcione, se necesita un protocolo de comunicación entre el prototipo y la memoria *SD*, el cual es *SPI*. La configuración de este periférico se tiene en la figura 3.17, donde este opera a una velocidad de transmisión de 18 *MBps*. Además, se muestra que pines son asignados para comunicarse con la memoria *SD*, teniendo a consideración que el pin PA4 fungirá como *CS* (selección del *CHIP* explicado en el capítulo 2 en la sección 2.5.1). En la figura 3.9, se tiene un módulo para la memoria *SD*, que integra todos los circuitos y conectores necesarios, sin embargo en el capítulo 4 se mostrará el circuito realizado para integrar este módulo en una única placa de circuito impreso.

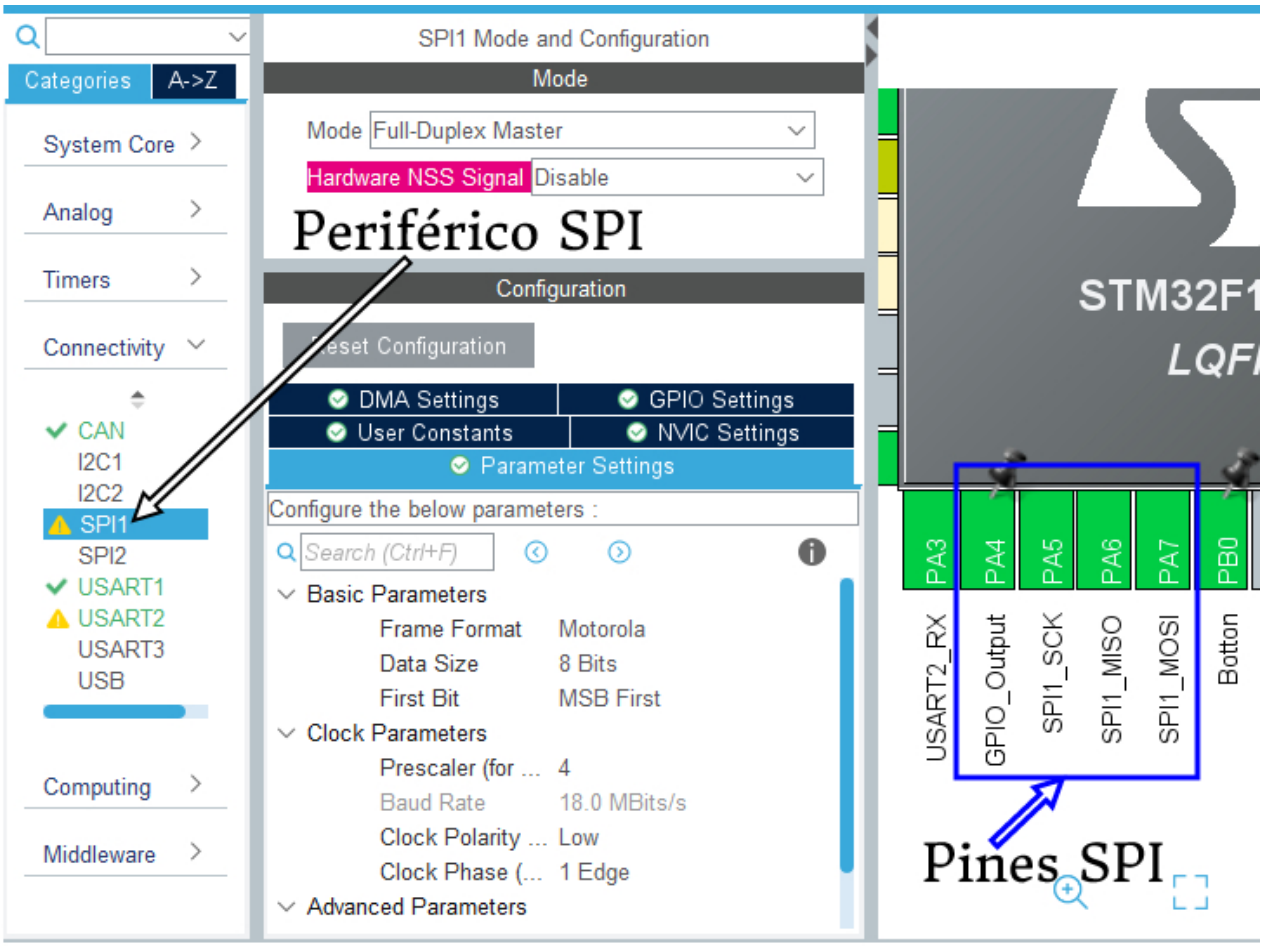


Figura 3.17: Configuración del periférico *SPI* del microcontrolador *STM32F446RE*.

Entonces enfocándose en la programación de este módulo, lo primero a realizar, es el algoritmo de la figura 3.18, que muestra la *función de inicialización de la memoria SD*, donde se declara las bibliotecas $fatfs.h$ y $fatfs\_sd.h$, las cuales integran todas las funcionalidades necesarias para comunicarse con la memoria *SD*. Seguido de esto, se configura el protocolo *SPI* a 18 MHz y la *USART* a 115200 *bps* (para monitorear el proceso del algoritmo), de ahí se espera un segundo, en lo que se termina de configurar los demás periféricos como el *CAN bus* y el módulo *4G/LTE*.

Después se entra en un bucle para preguntar si se ha montado la memoria *SD* (previamente se debió conectar la memoria), en caso de estar conectada, se mandará por la *USART*, *"Memoria SD montada exitosamente"*, en caso de no estar montada correctamente el usuario debe verificar las conexiones realizadas y por la *USART* se enviará *"La Micro SD card no montada"*. Entonces se procederá abrir el archivo "Datos.txt" y mediante la función $f\_getfree$ se obtiene el tamaño de espacio libre en la memoria *SD*. A través de la función f_puts se guarda en la memoria la cadena *"Aquisición de datos vehiculares"*, esto nos ayudara a separar los datos cada vez que se haga una prueba. Por último se cierra el archivo *"Datos.txt"*.

Puede ocurrir dos casos al momento de leer el archivo ya mencionado, la primera que no se encuentre, pero que si se pueda crear uno nuevo. La segunda que no se pueda leer la memoria, por lo tanto, no se podrá saber si está el archivo, entonces se mandará un mensaje por medio de la *USART*, con la siguiente cadena: *"El archivo no pudo ser abierto"* y se tendrá que revisar si no se tiene algún virus malicioso. La codificación del algoritmo de la figura 3.18 se muestra en el código 3.7.

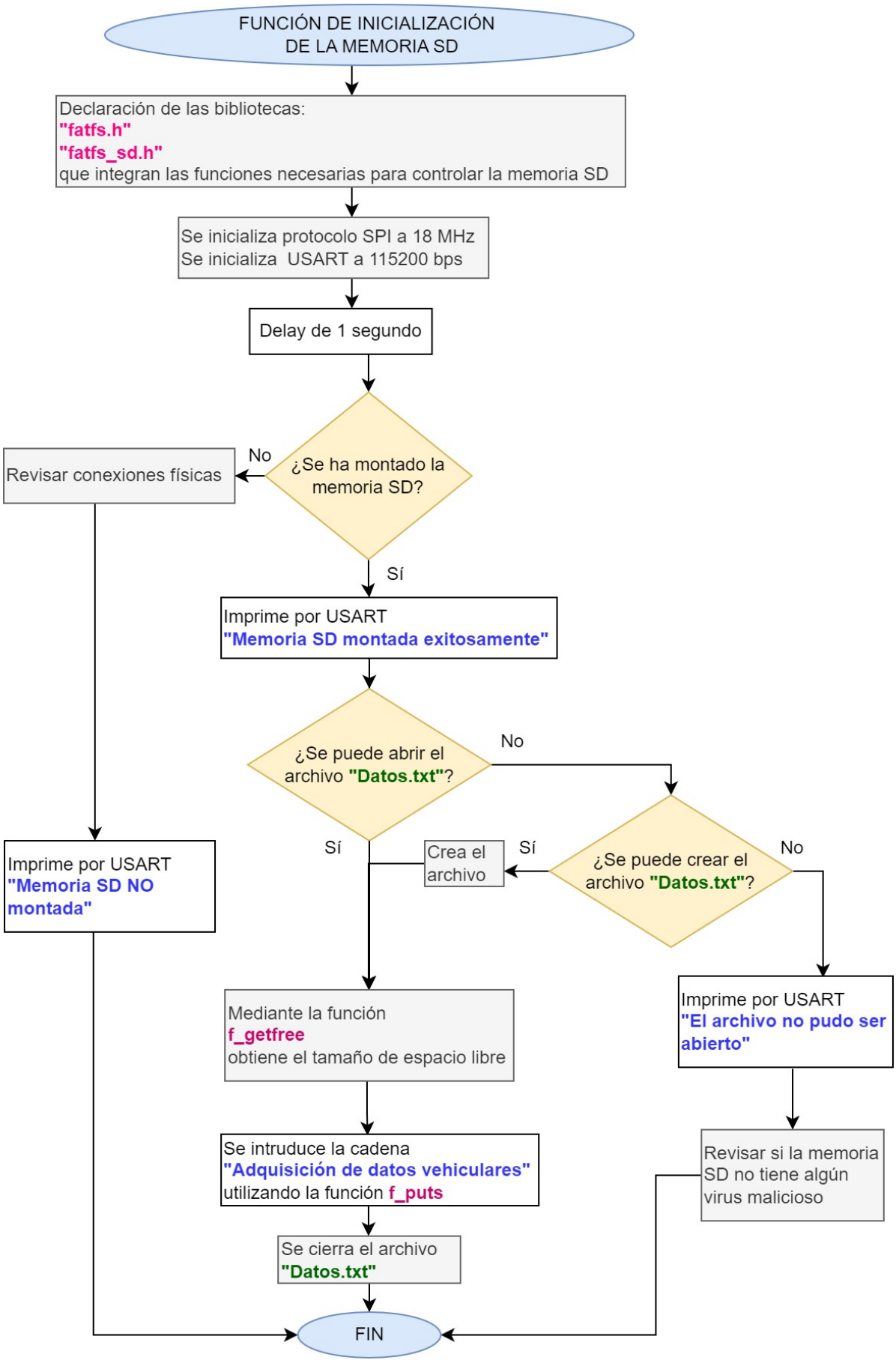


Figura 3.18: Algoritmo de inicialización de la memoria SD.

```
/* Inicialización de la mememoria SD */
  HAL_Delay(1000); //a short delay is important to let the SD card settle
  myprintf("\r\n~ Lectura y escritura en la memoria SD ~\r\n\r\n");

 //Open the file system
  fres = f_mount(&fs, "", 0);
  if (fres == FR_OK) {
    myprintf("La Micro SD card ha sido montada exitosamente\r\n");}
  else if (fres != FR_OK) {
      myprintf("La Micro SD card no se ha podido montar\r\n");}

// FA_OPEN_APPEND abre el archivo si existe y si no lo crea,
  // el puntero se establece al final del archivo para agregar
  fres = f_open(&fil, "Datos.txt", FA_OPEN_ALWAYS | FA_WRITE | FA_READ);

  if (fres == FR_OK) {
    myprintf("Archivo abierto para revisar el espacio libre de la memoria.\r\n");}
   else if (fres != FR_OK) {
    myprintf("¡Archivo NO abierto para revisar el espacio libre de la memoria!\r\n");}

  fres = f_getfree("", &fre_clust, &pfs);
  totalSpace = (uint32_t) ((pfs->n_fatent - 2) * pfs->csize * 0.5);
  freeSpace = (uint32_t) (fre_clust * pfs->csize * 0.5);
  char mSz[12];
  sprintf(mSz, "%u", freeSpace);

  if (fres == FR_OK) {
    myprintf("El espacio libre en la memoria es: ");
    myprintf(mSz);
    myprintf(" KiB \n");}
  else if (fres != FR_OK) {
    myprintf("El espacio libre no ha podido ser determinado\r\n");}

  f_puts("      Adquisición de datos vehiculares\n", &fil);

  fres = f_close(&fil);

  if (fres == FR_OK) {
    myprintf("Archivo cerrado.\r\n");}
   else if (fres != FR_OK) {
    myprintf("Archivo no pudo ser cerrado.\r\n");}
```

Código 3.7: Función de inicialización de la memoria SD.

Para la *Función de almacenamiento de la memoria SD* (ver figura 3.19), la cual guardará los parámetros solicitados por el *OBD-II* en formato *JSON*, se declara un array de tamaño indefinido nombrado *content* el cual almacenará los dichos parámetros. Primero se ejecuta la función de inicialización de la memoria SD descrita anteriormente, para luego invocar a la función *Crear_JSON* para convertir los datos del *OBD-II* en dicho formato.

Para guardar cualquier cadena de caracteres en la memoria SD, se necesita primero abrir el archivo, después guardar la cadena (en este caso es *content*), luego escribir mediante la función *f_puts* un salto de línea, esto para que la siguiente cadena a almacenar se escriba debajo de esta y no sobre la misma línea. Por último se debe cerrar el archivo, para garantizar que la cadena sea almacenada exitosamente. El código 3.7 muestra la codificación de este algoritmo, donde cabe aclarar que la función *Crear_JSON*() no se muestra su contenido para que al lector le sea factible observar el código de la función *Guardar_datos_en_SD*(), la cual representa la función antes mencionada. En la figura 3.20 se muestra los *PIDs* almacenado en el archivo "*Datos.txt*".

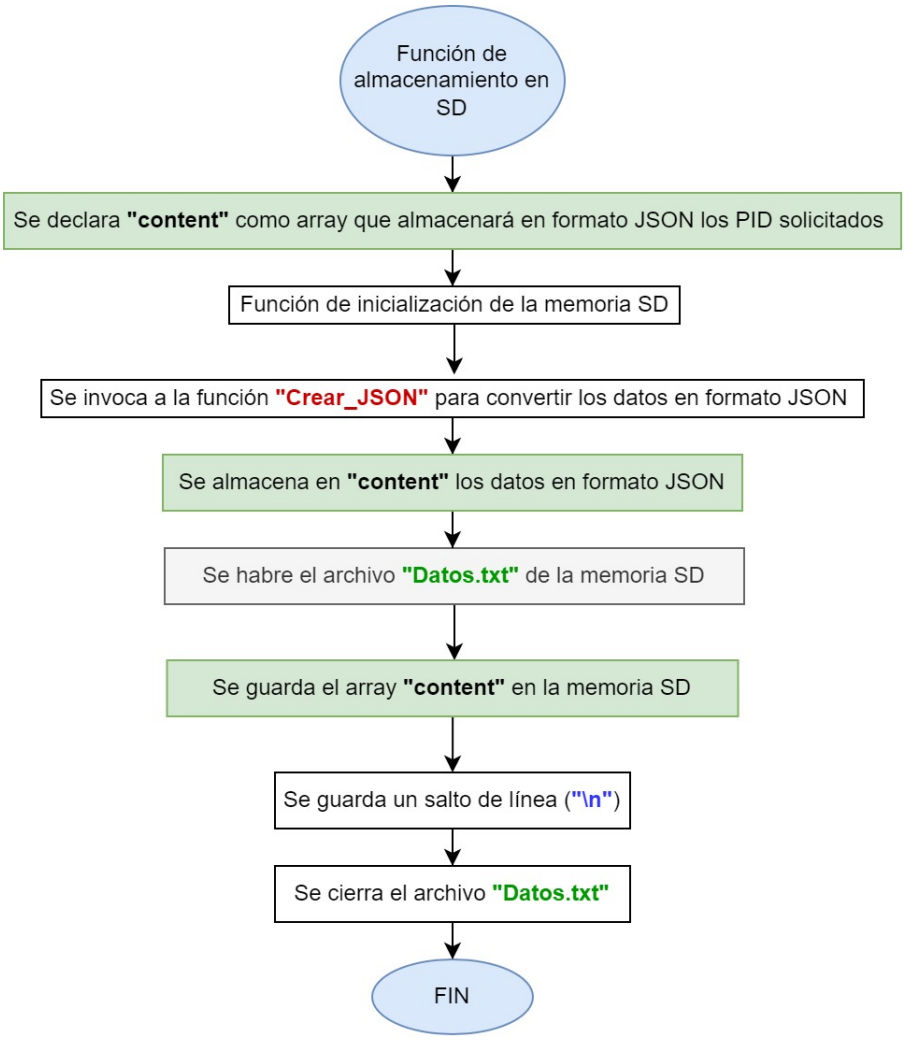


Figura 3.19: Algoritmo para el almacenamiento de los datos obtenidos del *OBD-II* en la memoria SD.

```
void Crear_JSON(){
.
.
.
}
void Guardar_datos_en_SD()
{
    f_open(&fil, "Datos.txt", FA_OPEN_ALWAYS| FA_WRITE | FA_READ);
    f_write(&fil,content,sizeof(content),&bw);
    f_puts("\n", &fil);
    fres = f_close(&fil);
}
```

Código 3.8: Codificación de la función *Guardar_datos_en_SD*.

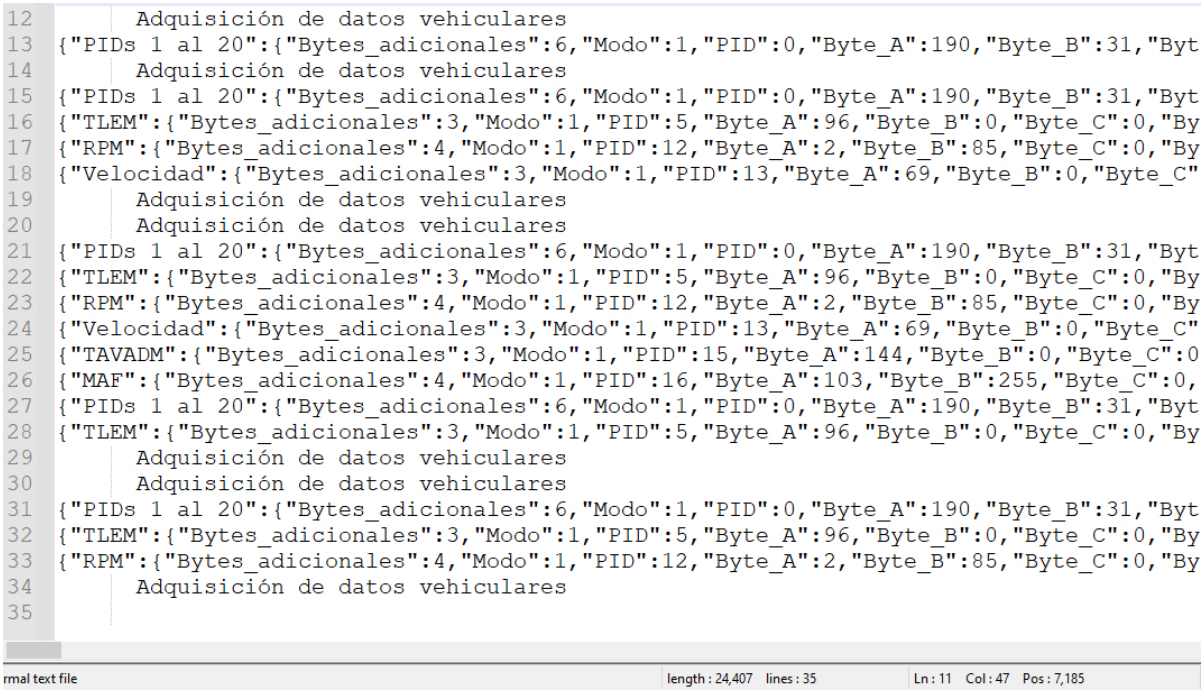


Figura 3.20: Parámetros del *OBD-II* con formato *JSON* almacenados en el archivo *"Datos.txt"*.

3.5. Comunicación con el módulo *4G/LTE*

La tarjeta de desarrollo seleccionada para hacer pruebas de comunicación inalámbrica es la *SIM7600X 46 HAT*; integra un módulo *SIM7600G* de la empresa *SIMCOM*, permitiendo enviar/recibir mensajes por medio de la tecnología *4G/LTE*, además de integrar un *GPS*. El módulo tiene los protocolos de comunicación *SPI*, *I2C* y serial para ser controlado. En este caso se optó por usar la *USART*, ya que por defecto viene seleccionado este protocolo. En la figura 3.21 se observa la configuración de la *USART1* que tiene el microcontrolador del prototipo, donde se establece una tasa de transferencia de 115200 *baudios*, no paridad y 1 bit de *stop*. El pin PA10 es el transmisor y el pin PA9 es receptor.

Antes de inicializar este módulo es necesario conocer los *códigos AT*, que son básicamente una cadena de caracteres, que inicia con *AT*, de ahí proviene su nombre, seguido de esto, según lo especifique el manual de comandos [83] del *SIM7600G*, pueden venir acompañados de un *+* y otros caracteres, por ejemplo para solicitar la información del *GPS*, se escribe *AT+CGPSINFO\r\n*, donde el retorno de carro (\r) y el salto de línea (\n) son necesarios al final de cada comando para dar la instrucción al módulo *4G/LTE* que debe ejecutarlo, por ello, retomando el ejemplo anterior, si solo se manda: *AT+CGPSINFO*, el módulo no ejecuta ninguna instrucción, regresando una cadena de caracteres con la leyenda *error*. Entonces, el módulo puede responder de diferentes maneras a cada comando enviado, sin embargo el más común es: *OK*, que indica que se ha ejecutado correctamente.

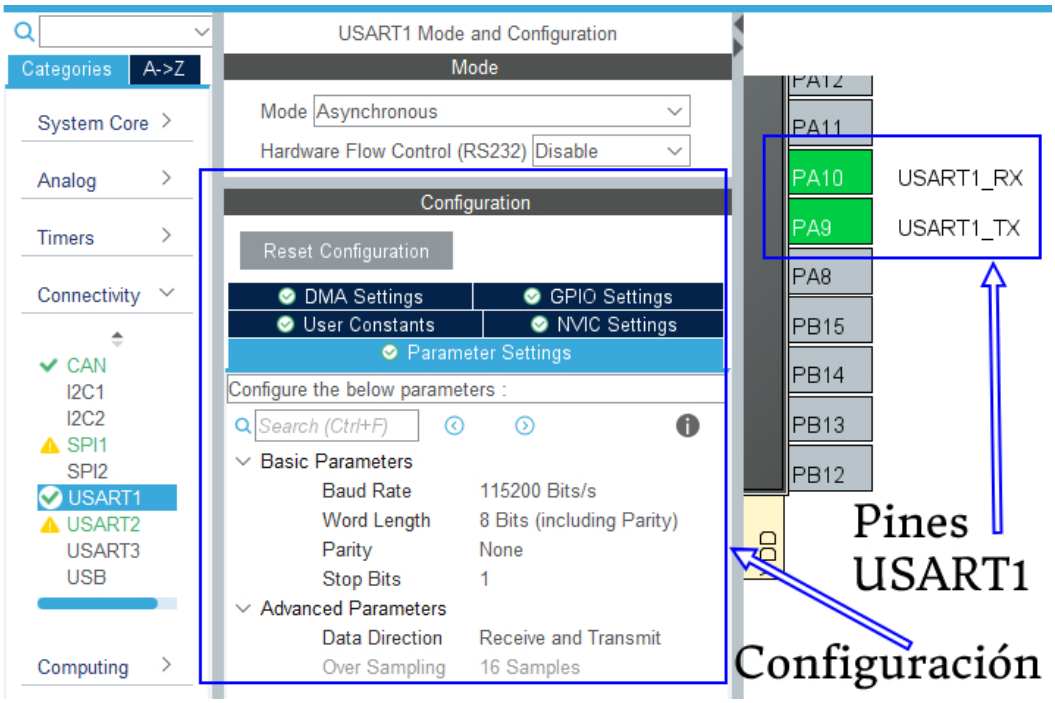


Figura 3.21: Configuración de la *USART1* del microcontrolador *STM32F446RE*.

La función de inicialización de este módulo se observa en la figura 3.22, configurando tanto la *USART1* como la *USART2* a una velocidad de transmisión de *115200 bits/s*, 8 bits (incluido el bit de *stop*) y sin paridad. Por el momento no se explicará a detalle como transmite los datos hacia la *PC* la *USART2*, eso se deja para la sección 3.7, pero para la *USART1*, se tiene que incluir la biblioteca *String.h*, que tiene la función *sprinf*, la cual en su primer parámetro se declara el *array* que almacena la cadena de caracteres y en su segundo parámetro el comando *AT* en formato de cadena.

Entonces primero se guarda el comando $AT\backslash r$ en el array *codigos_AT*, para ser enviado tanto por la *USART1* como por la *USART2*. Este comando en particular es para probar si la conexión serial no tiene fallas, ya que el módulo *4G/LTE* debe responder con un *OK*, transmitiendo esta respuesta a la *PC* mediante la *USART2*. En caso de presentarse una complicación, el módulo responderá con *error* y el usuario deberá revisar las conexiones físicas del módulo. El segundo comando es $AT+CREG?\backslash r$, que indica si se tiene conexión a la red inalámbrica y de igual manera debe responder con *OK* para confirmar que todo es correcto, sino se deberá revisar si la tarjeta *SIM* tiene saldo o en su defecto, verificar si se encuentra bien conectada. El tercer comando es $AT+CGPS=0\backslash r$ que desactivará el módulo *GPS*, esto se hace porque al cargar un programa al microcontrolador *STM32F446RE*, se debe presionar el pulsador de *reset* y si se deja activado el *GPS*, al activarse por segunda vez, genera un falló, así que por eso es necesario desactivarlo al comenzar la función. Por último con el comando $AT+CGPS=1\backslash r$ activa el *GPS*.

La codificación de esta función se visualiza en el código 3.8, donde la función *myprintf* es la encargada de transmitir por la *USART2* hacia la *PC*, los mensajes que se reciben del módulo *4G/LTE* y con la función $HAL\_UART\_Transmit$, se envían los comandos *AT* al módulo *4G/LTE* por medio de la *USART1*, pero sin el último carácter, ya que este es el nulo y al enviarlo en conjunto con los demás caracteres, ocasiona que el módulo no reconozca el comando. La función $HAL\_UART\_Receive$ espera la respuesta del módulo $4G/LTE$ y los datos recibidos son almacenados en el array $datos\_recibidos\_uart1$, para después ser mostrados en la PC mediante la función $myprintf$. Esto se realiza con los 4 comandos mencionados anteriormente.

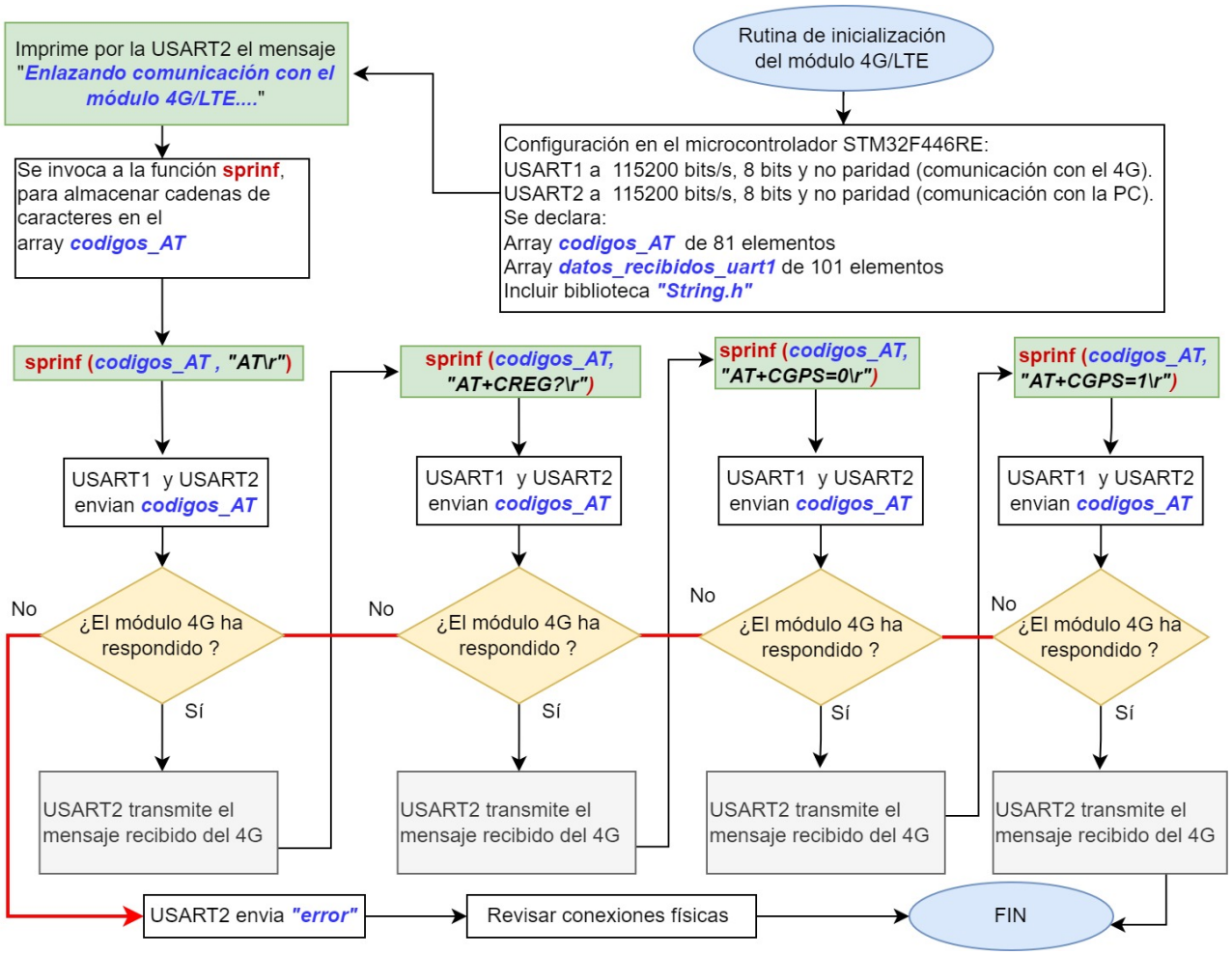


Figura 3.22: función de inicialización del módulo *4G/LTE*.

```
1  char codigos_AT[80];
2  myprintf("Enlazando comunicación con el módulo 4G/LTE....\r\n");
3
4   sprintf(codigos_AT,"AT\r");
5   myprintf(codigos_AT);
6   HAL_UART_Transmit(&huart1, (uint8_t*)codigos_AT, sizeof(codigos_AT) - 1, -1);
7 // Se le resta -1 para no enviar el nulo
8   HAL_UART_Receive(&huart1,(uint8_t*)datos_recibidos_uart1,sizeof(datos_recibidos_uart1
      ),100);
9   myprintf((char*)datos_recibidos_uart1);
10
11  sprintf(codigos_AT,"AT+CREG?\r");
12  myprintf(codigos_AT);
13  HAL_UART_Transmit(&huart1, (uint8_t*)codigos_AT, sizeof(codigos_AT) - 1, -1);
14  HAL_UART_Receive(&huart1,(uint8_t*)datos_recibidos_uart1,sizeof(datos_recibidos_uart1
      ),100);
15  myprintf((char*)datos_recibidos_uart1);
16
17  sprintf(codigos_AT,"AT+CGPS=0\r");
18  myprintf(codigos_AT);
19  HAL_UART_Transmit(&huart1, (uint8_t*)codigos_AT, sizeof(codigos_AT) - 1, -1);
20  HAL_UART_Receive(&huart1,(uint8_t*)datos_recibidos_uart1,sizeof(datos_recibidos_uart1
      ),100);
21  myprintf((char*)datos_recibidos_uart1);
22
23  sprintf(codigos_AT,"AT+CGPS=1\r");
24  myprintf(codigos_AT);
```

```
25  HAL_UART_Transmit(&huart1, (uint8_t*)codigos_AT, sizeof(codigos_AT) - 1, -1);
26  HAL_UART_Receive(&huart1,(uint8_t*)datos_recibidos_uart1,sizeof(datos_recibidos_uart1
      ),100);
27  myprintf((char*)datos_recibidos_uart1);
```

Código 3.9: Codificación de la función de inicialización del módulo $4G/LTE$.

3.5.1. Obtención de geolocalización del *GPS*

En el código 3.9 se activó el *GPS*, sin embargo cabe aclarar que este módulo tarda de 2 a 5 minutos para recibir la geolocalización, antes de este tiempo, se obtiene una cadena vacía. La función de la figura 3.23 muestra la forma de obtener la geolocalización, para esto se declaran dos variables (*conta_car* y *conta_caracteres*) que servirán para incrementar la posición de los *arrays bufferH*[ ] y *cadena*[ ]. Se tiene una función dedicada al envío de comandos AT al *SIM7600G*, la cual se explica en la sección 3.6, nombrada *SIMTransmit*, que tiene como parámetro recibir una cadena de caracteres con el comando a enviar, en este caso, es ATE0, que desactiva el eco, con el fin de que el SIM no envíe dos veces el mensaje y AT+CGPSINFO que solicita la geolocalización con formato de grados y minutos decimales (*DMM*). La respuesta del *GPS* sera guardada momentáneamente en el *array bufferH*[ ] y se muestra a continuación un ejemplo de su estructura:
+CGPINFO: *1900.497470,N,09812.365136,W,190722,224401.0,2135.0,0.0,*

Obsérvese como se debe quitar los caracteres +CGPINFO: para así obtener únicamente la geolocalización, para esto, se hace uso de un bucle, donde la posición del *bufferH* se irá incrementando, partiendo de 0 hasta llegar al espacio en blanco " ", el incremento se hace mediante la variable *conta_caracteres* y una vez que se ha llegado a dicho carácter, se incrementara un vez más y ahora se compara con el retorno de carro, si aún no se llega a este carácter, se procederá a copiar los valores de *bufferH*[ ] a *cadena*[ ], y cada posición se incrementara con la variable *conta_car* como se muestra en la figura 3.23. Una vez que *bufferH*[*conta_caracteres*] = '\r' las variables antes mencionadas se restablecerán con valor 0, para que así, la siguiente vez que se solicite estos datos, comiencen a incrementar su valor desde 0.

El código 3.10 corresponde al algoritmo de la figura 3.23, donde inicialmente se manda a la PC la cadena *"Consulta de los datos del GPS*\n\r*"*, seguido se envía mediante la función *SIMTransmit*, el comando de desactivar el eco y la solicitud de la geolocalización. Después los dos bucles mencionados en el párrafo anterior se codifican mediante un ciclo *while* y las variables se incrementan de uno en uno mediante el operador de incremento ++. Por último al final cada variable se le asigna el valor 0, para reiniciar el conteo.

```
1   myprintf("Consulta de los datos del GPS\n\r");
2   SIMTransmit("ATE0\r\n");
3   SIMTransmit("AT+CGPSINFO\r\n");
4     while(bufferH[conta_caracteres] != ' ')
5     { conta_caracteres ++; }
6         conta_caracteres ++;
7       while(bufferH[conta_caracteres] != '\r')
8       {   cadena[conta_car] = bufferH[conta_caracteres];
9           conta_car++;
10          conta_caracteres ++; }
11
12          conta_car = 0;
13          conta_caracteres = 0;
```

Código 3.10: Codificación de la función de obtención de datos de geolocalización.

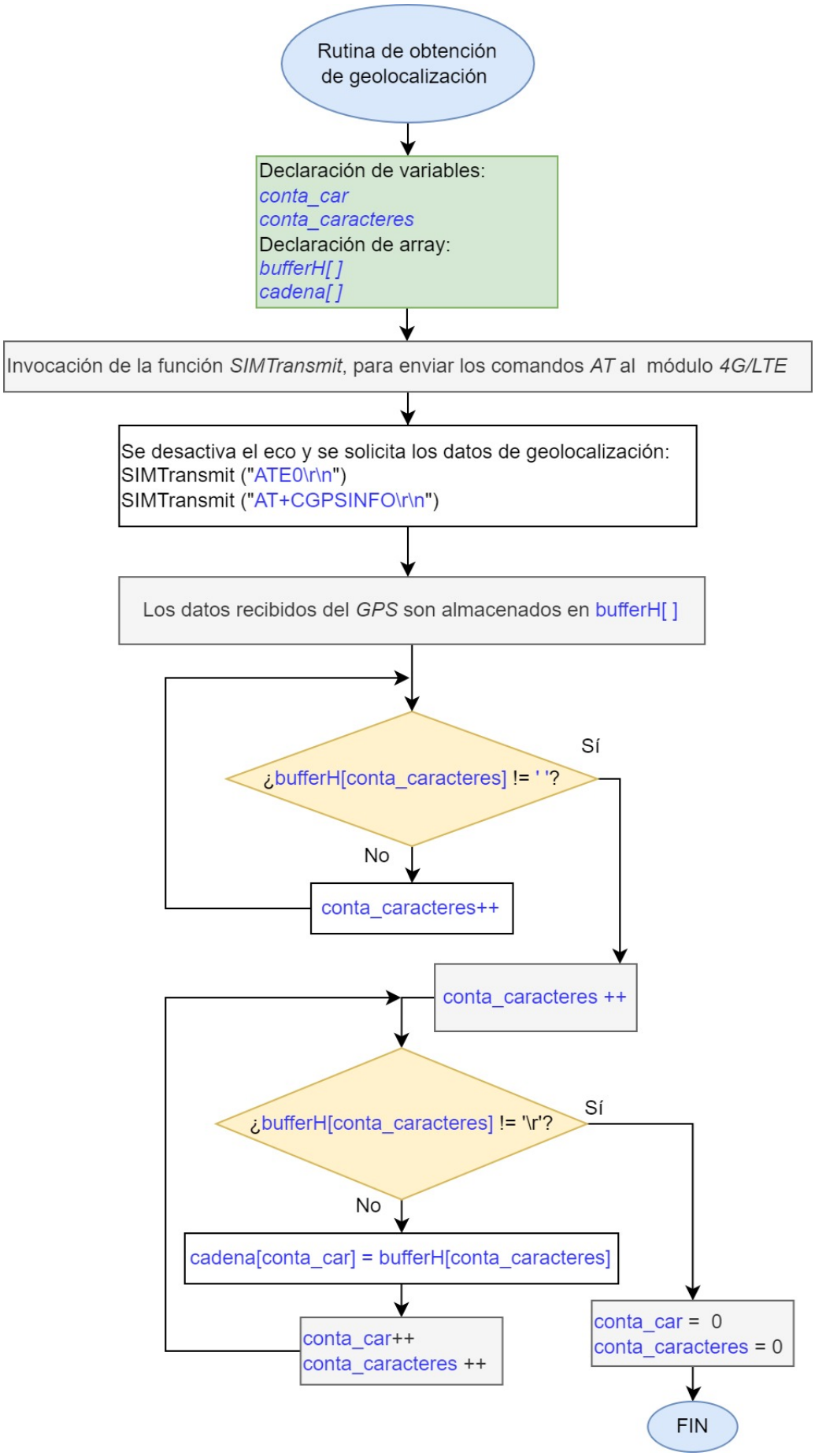


Figura 3.23: Algoritmo para obtener la cadena de caracteres correspondiente a la geolocalización.

3.5.2. Conversión de los datos adquiridos del *OBD-II* al formato *JSON*

Al seleccionar *Google Cloud* como nube, este tiene por defecto *Firebase* como gestor para la base de datos, el cual recibe lo transmitido en formato *JSON*.

Formato *JSON*

Este formato es utilizado principalmente para el intercambio de mensajes entre dispositivos, como por ejemplo, en este caso, transmitir/recibir datos entre el SADO y la nube. Existen dos elementos en este formato, según lo especificado por sus sintaxis:

- Matrices: Son una lista de valores (pueden ser tanto números decimales como caracteres con acento) separados por comas y dentro de corchetes. Ejemplo: [78, "*JUAN*", 17.509, "*Conexión*"]
- Objetos: Es una lista de valores semejante a las matrices, solo que estos van identificados por un nombre y después por dos puntos *":"* sigue su valor, separando de la siguiente pareja por una coma, todo esto dentro de llaves. Ejemplo: {"*PID*" : 5, "*ID*" : "0001", "*Voltaje*" : 3.3}

Para el envío de los datos hacia la nube se usara objetos, ya que permite poner un identificador a cada valor enviado. Entonces se definieron los siguientes identificadores para el objeto *datos_auto*:

- *ID_Auto*: Este es un identificador particular de cada vehículo. En este momento tiene el valor constante "0001", ya que se realizaron pruebas en el emulador del *OBD-II*, sin embargo a futuro se siguiere modificarlo dependiendo la flotilla de vehículos a utilizar.
- *Identificador*: Este es el *ID* que envía el *OBD-II*, que es 0x7E8.
- *Bytes_adicionales*: Acorde a la figura 3.18, donde se especificó la trama enviada desde el emulador hacia el prototipo, este es el primer byte recibido en el campo de datos, indicando cuantos bytes adicionales serán tomados en cuenta.
- *Modo*: Modo del *OBD-II* (se espera el valor 1).
- *PID*: Número de *PID* solicitado.
- *Byte_A*: 4° byte del campo de datos solicitado.
- *Byte_B*: 5° byte del campo de datos (puede tener valor 0x00).
- *Byte_C*: 6° byte del campo de datos solicitado (puede tener valor 0x00).
- *Byte_D*: 7° byte del campo de datos solicitado (puede tener valor 0x00).
- *Dato8*: 8° byte del campo de datos solicitado (con valor 0x55 ó 0x00)
- *GPS*: Datos de geolocalización en formato de grados y minutos decimales (*DMM*).

A continuación se muestra un ejemplo del objeto *datos_auto* donde se envían las *RPM*:
{ "*Bytes_adicionales*" : "4", "*Modo*" : "1", "*PID*" : "12", "*Byte_A*" : "2", "*Byte_B*" : "85", "*Byte_C*" : "0", "*Byte_D*" : "0", "*Dato8*" : "85", "*GPS*" : *"1900.497470,N,09812.365136,W,190722,224401.0,2135.0,0.0,* ", "*ID_Auto*" : "0001", "*Identificador*" : "7E8" }

Función de interrupción *CAN bus* del SADO

El envío de los datos hacia la base de datos, se realiza mediante la función de interrupción *CAN bus* del SADO, para esto como se observa en la figura 3.11, al oprimir el *Boton*, se solicita el primer *PID* del *OBD-II*, entonces el emulador responde con el parámetro solicitado y es ahí cuando esta función toma lugar, la cual se observa en la figura 3.24, donde se hace un filtrado dejando pasar solo los mensajes con ID = 0x7E8, después se va comprando el tercer byte del *array RXCAN*[ ], si este es 0x00, indica cuales parámetros del *OBD-II* se pueden consultar, luego se invoca a la función *Crear_Josn* (código

3.11), donde convierte los datos obtenidos del *OBD-II* en el formato *JSON*, para después ser enviados a base de datos utilizando el protocolo *HTTP*.

Si el *PID* solicitado es diferente al 0x00, entonces se tiene la variable CAN_RXBUS y como se observa en la figura 3.30, sirve para identificar que *PID* se ha solicitado y cual es el siguiente a pedir, por ejemplo si se ha solicitado la temperatura del motor (PID = 0x05), el siguiente parámetro a pedir son las *RPMs* (PID = 0x0C), entonces la variable CAN_RXBUS se le debe asignar el valor 0x05, para indicar al microcontrolador del prototipo que el emulador del *OBD-II*, ha respondido a este *PID* solicitado.

La codificación de la figura 3.30, se observa en el código 3.10. Notesé como la estructura condicional múltiple *switch*, sirve para identificar que *PID* se ha recibido y posteriormente enviar el *array* $RXCAN$ a la base de datos en *Firebase*, mediante la función $Crear\_JSON$. Al final se imprimen los datos del *array content*, que tiene en formato *JSON*, los datos recabados tanto del *OBD-II* como del *GPS*.

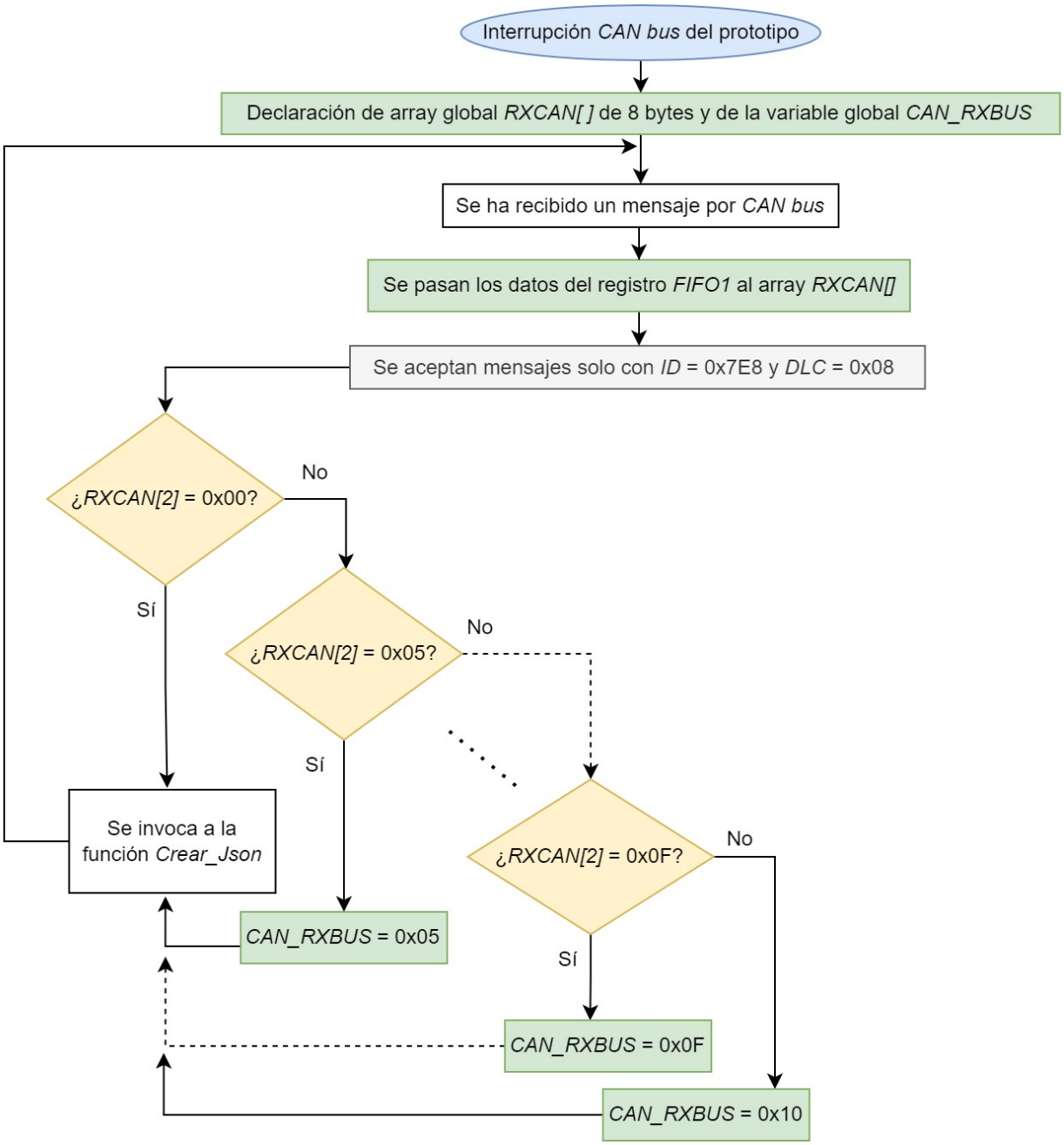


Figura 3.24: Algoritmo de la función de interrupción *CAN bus* del microcontrolador *STM32F446RE*.

```
void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
  if (HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO1, &RxHeader, RXCAN) == HAL_OK)
      {
    if(hcan-> Instance == CAN1 )
       {if((RxHeader.StdId == 0x7E8)        ||
                (RxHeader.RTR == CAN_RTR_DATA) ||
                (RxHeader.IDE == CAN_ID_STD)   ||
                (RxHeader.DLC == 8)){
           switch(RXCAN[2])
              {
                case 0x00: //PIDs implementados [01 - 20]
                  Crear_JSON();
                  break;

                case 0x05: //Temp. del liquido de enfriamiento del motor
                  CAN_RXBUS = 0x05;
                  Crear_JSON();
                  break;

                case 0x0C: //RPMS del motor
                  CAN_RXBUS = 0x0C;
                  Crear_JSON();
                  break;

                case 0x0D: //Velocidad del vehiculo
                  CAN_RXBUS = 0x0D;
                  Crear_JSON();
                  break;

                 case 0x0F: //Temp. del aire del colector de admisión
                  CAN_RXBUS = 0x0F;
                  Crear_JSON();
                    break;

                default:  //MAF
                          CAN_RXBUS = 0x10;
                  Crear_JSON();

              }
          myprintf(content);
          datos_CAN_recibidos = 1;
```

Código 3.11: Función de interrupción *CAN bus* del SADO.

Conversión los parámetros solicitados del *OBD-II* en formatos *JSON* mediante la función *Crear_Json*

La conversión al formato *JSON*, se realiza con base en la función *sprintf*, por lo que se procederá a explicar que parámetros recibe y que valor devuelve, mediante el código 3.12 de ejemplo. Su estructura es la siguiente:
int sprintf (*char * str*, *const char* * formato);
Donde:

- Primer parámetro (*char * str*): *Array* donde se almacena la cadena de caracteres.
- Segundo parámetro (*const char* * formato): Cadena de caracteres a guardar.
- Retorno: Se devuelve el número actual de caracteres escritos, sin contar el nulo que se escribe al final.

Ejemplo:

```c
#include <stdio.h>

int main ()
{
  char buffer[50];
  int n, a=5, b=3;
  n=sprintf (buffer, "%d mas %d es %d", a, b, a+b);
  printf ("[%s] esta es una cadena de %d caracteres de largo\n",buffer,n);
  sprintf(buffer+n, " + cadena sumada");
  printf(buffer);
  return 0;
}
```

Código 3.12: Ejemplo del uso de la función *sprintf*.

Salida:

[5 mas 3 es 8] esta es una cadena de 13 caracteres de largo

5 mas 3 es 8 + cadena sumada

Explicación: La variable *"n"* recibe el total de caracteres almacenados en *buffer*, que en este caso son 13, al invocar por primera la función *sprintf*. Después cuando se invoca por segunda vez esta función, se agrega los caracteres: *"+ cadena sumada"* y para que no se sobre escriba, se pone en el primer parámetro de *sprintf*, *"buffer+n"*, indicando así la posición donde debe escribir la siguiente cadena de caracteres.

El algoritmo de la función *Crear_Json*, se muestra en la figura 3.25, donde se invoca la función *sprintf* y la variable *j* lleva el conteo del número actual de caracteres escritos. Entonces al guardar en el *array content* la siguiente cadena: "*Bytes_adicionales*" : "4", en *j* se tendrá la posición final del último carácter guardado, entonces para la siguiente cadena: "*Modo*" : "1", se debe indicar por medio del parámetro de la función *sprinft*, donde debe comenzar a escribir la siguiente cadena, para ello se declara así: *j += sprinft(content+j,* "*Modo*" : "1", teniendo entonces el siguiente conjunto de caracteres: { "*Bytes_adicionales*" : "4", "*Modo*" : "1". Nótese como se usará *j+=*, para ir sumando el número anterior de posición con el actual y así no sobrescribir los datos con los anteriores.

El código 3.13, se muestra la función crear *JSON*, donde los datos de *RXCAN[ ]*, son convertidos de hexadecimal a decimal por medio del especificador de formato "*%d*" y en el apartado de *GPS*, se añade los datos de geolocalización como una cadena de caracteres mediante el especificador "*%s*", al final ambos serán almacenados en *content* como una cadena de caracteres. Por último se guarda el *ID Auto*, donde se escogió el valor 0001, ya que solamente se están haciendo pruebas en el emulador del *OBD-II*. Es importante aclara que la variable *j*, se debe reiniciar una vez que se ha finalizado la conversión de datos al formato *JSON*, para así volver a iniciar el conteo.

```c

void Crear_JSON(){
  j  = sprintf(content, "{\"Bytes_adicionales\":%d,", RXCAN[0]);
    j += sprintf(content+j, "\"Modo\":%d,",   RXCAN[1]);
  j += sprintf(content+j, "\"PID\":%d,",    RXCAN[2]);
  j += sprintf(content+j, "\"Byte_A\":%d,", RXCAN[3]);
  j += sprintf(content+j, "\"Byte_B\":%d,", RXCAN[4]);
  j += sprintf(content+j, "\"Byte_C\":%d,", RXCAN[5]);
  j += sprintf(content+j, "\"Byte_D\":%d,", RXCAN[6]);
  j += sprintf(content+j, "\"Dato8\":%d,", RXCAN[7]);
  j += sprintf(content+j, "\"GPS\":\"%s\",",  cadena);
  j += sprintf(content+j, "\"ID_Auto\":\"0001\",\"Identificador\":\"7E8\"}");

}
```

Código 3.13: Uso de la función *sprintf* para convertir los parámetros adquiridos del *OBD-II* en formato *JSON*.

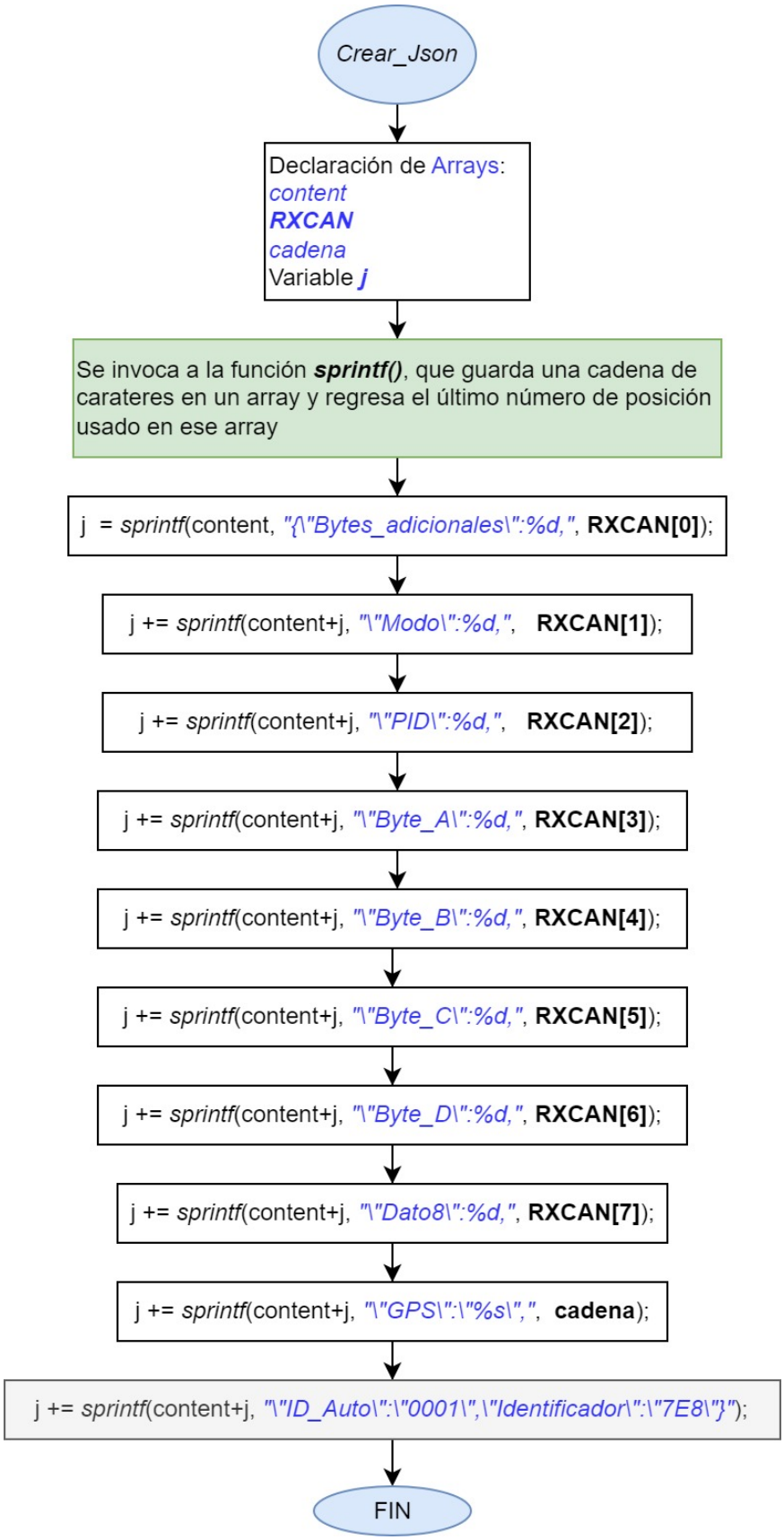


Figura 3.25: Algoritmo para convertir los datos obtenidos del *OBD-II* en formato *JSON*.

3.6. Comunicación entre el módulo *4G/LTE* y la base de datos

Una vez que los datos obtenidos tienen el formato *JSON*, el siguiente paso, es enviarlos a la base de datos. La plataforma de Firebase se observa en la figura 3.26, donde se tiene el objeto *datos_auto* y sus correspondientes elementos, ya mencionados anteriormente. Es importante conocer la dirección *HTTP* de la base de datos para saber a donde envían y en que objeto se almacenan.

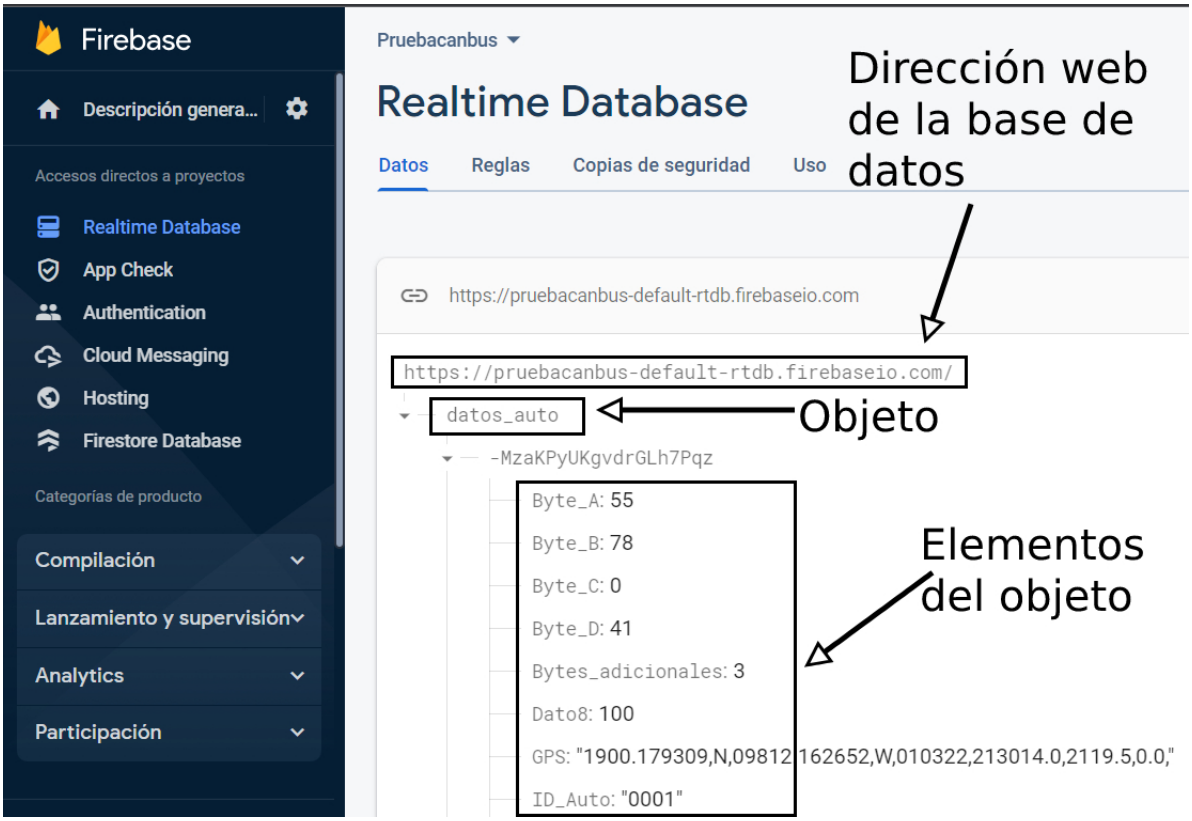


Figura 3.26: Visualización de objeto datos_auto en *Firebase*.

Después se debe obtener la clave de autorización, que servirá para acceder a esta base de datos en tiempo real. Para ello como se ve en la figura 3.27, en la *configuración del proyecto*, en la opción *Cuentas del servicio*, en el apartado de *secretos de la base de datos*, se muestra la clave: *vrae1gf9Ji3B3eF03iwaD3V7LdHMEUVKhGYXvTdL*. Es importante destacar que, aun teniendo este clave, la base de datos es vulnerable ataques cibernéticos, pero se pueden modificar las reglas de seguridad, mediante código para hacerla robusta. Se aclara que no se modificó esta parte, ya que por el momento solo se están haciendo pruebas y no es necesario que la seguridad sea estricta, sin embargo si el proyecto se amplía a un cliente en concreto, entonces se debería considerar las medidas necesarias para garantizar la seguridad de los datos.

Antes de presentar los algoritmos y códigos para la transmisión de datos desde el *4G/LTE* hacia la base de datos, se justifica porque no se usó *MQTT* como protocolo, ya que se había mencionado en el capítulo 2, este tiene mejores ventajas sobre *HTTP*, cuando no se envían fotos, vídeos o archivos del orden de *MegaBytes*.

Figura 3.27: Apartado en *Firebase* para obtener la clave de la base de datos.

3.6.1. Justificación del uso de *HTTP* como protocolo para enviar los datos a la nube

El formato *JSON* definido no sobrepasa los *MegaBytes*, siendo entonces mejor usar *MQTT*, sin embargo la mayoría de los servicios en *Firebase* se comunican principalmente mediante *HTTP* [84]. Y aunque el módulo *4G/LTE* integra ambos protocolos, actualmente no se tiene documentación detallada que sirva como marco de referencia para realizar algún ejemplo de comunicación con *MQTT* [84], de manera simple o intuitiva, siendo entonces necesario ser experto en el área de computación o comunicaciones inalámbricas para realizar de manera satisfactoria el envío-recepción de datos por *MQTT*, contrastando con *HTTP*, como se mencionaba en el capítulo 2, al ser popular entre la comunidad de desarrolladores de base de datos, se tienen códigos y documentación que permite a los usuarios un desarrollo sin la necesidad de tener un amplio conocimiento. Pero no se descarta en un futuro el uso de *MQTT* por sus ventajas en el *IoT*.

3.6.2. Algoritmo y código de envío de datos mediante *HTTP* hacia la base de datos

El envío de mensajes hacia la base de datos gestionada en *Firebase* está compuesto principalmente por dos funciones, la primera es *SIMTransmit* que se observa en la figura 3.28, esta función tiene como propósito mandar cadenas de caracteres al módulo *4G/LTE* y mediante la función *myprintf* monitorear lo transmitido/recibido. Entonces primero se inicializa la *USART1* y *USART2*, a 115200 *baudios*, sin paridad y con 1 bit de *stop*. Después se declara el *array bufferH* de 10000 elementos, este servirá para almacenar los datos de respuesta que envíe el módulo *4G/LTE*. Como a esta función se le va a pasar una cadena de caracteres, se tiene como parámetro el apuntador **cmd* del tipo *char*, que permite apuntar a la cadena recibida. Primero se inicializará el *array bufferH*, con el valor 0, para esto se hace uso de la función *memset*, la cual está compuesta de la siguiente manera:

*void *memset(void *str, int c, size_t n)*
Parámetros:

- str: Puntero del *array* a donde se va a guardar los datos.
- c: Carácter del tipo entero que se quiere guardar en el *array* antes especificado.
- n: Indica cuantas veces se quiere guardar el carácter anterior.

En ejemplo del uso de esta función se muestra en el código 3.14

```
#include <stdio.h>
#include <string.h>

int main () {
   char str[50];

   strcpy(str,"Esta es la biblioteca string.h");
   puts(str);

   memset(str,'0',7);
   puts(str);

   return(0);
}
```

Código 3.14: Ejemplo del uso de la función *memset*.

En consola debe salir el siguiente resultado:
Esta es la biblioteca *string.h*
0000000 la biblioteca *string.h*

Nótese como el carácter 0, se sobrescribe de los anteriores, modificando la cadena original. Entonces como se verá a continuación en el algoritmo de la figura 3.28, se invoca múltiples veces a la función *SIMTransmit*, siendo necesario inicializar todo el *array bufferH* con el valor 0, con la finalidad de que el siguiente mensaje no se sobrescriba con el anterior. Para esto se hace uso de la función *sizeof* que obtiene el tamaño del *bufferH*, para así determinar el número necesario para llenar todo el *array* con 0. Seguido se manda el comando AT en formato de cadena a través del apuntador *cmd* y la función $HAL\_UART\_Transmit$, donde se le específica que transmitirá dicha cadena mediante la *USART1*. El módulo *4G/LTE* debe responder a lo enviado, para ello se hace uso de la función $HAL\_UART\_Receive$ y los datos se guardan en *bufferH*. Tanto lo transmitido como lo recibido se manda a la PC mediante la función *myprintf*, a través de la *USART2*.

La codificación de la función *SIMtransmit* se muestra en el código 3.15, donde esta función no devuelve ningún valor (es del tipo *void*) y tiene como parámetros el apuntador **cmd*. Se da el siguiente ejemplo de como funciona:

$SIMTransmit(\text{“}AT + HTTPINIT\text{”})$;
Donde $AT + HTTPINIT$ es la cadena a la que apunta *cmd* y se espera como respuesta:
OK

Entonces *bufferH* almacena la cadena OK y esto se muestran en consola así:
$SIMTransmit(\text{“}AT + HTTPINIT\text{”})$;
OK

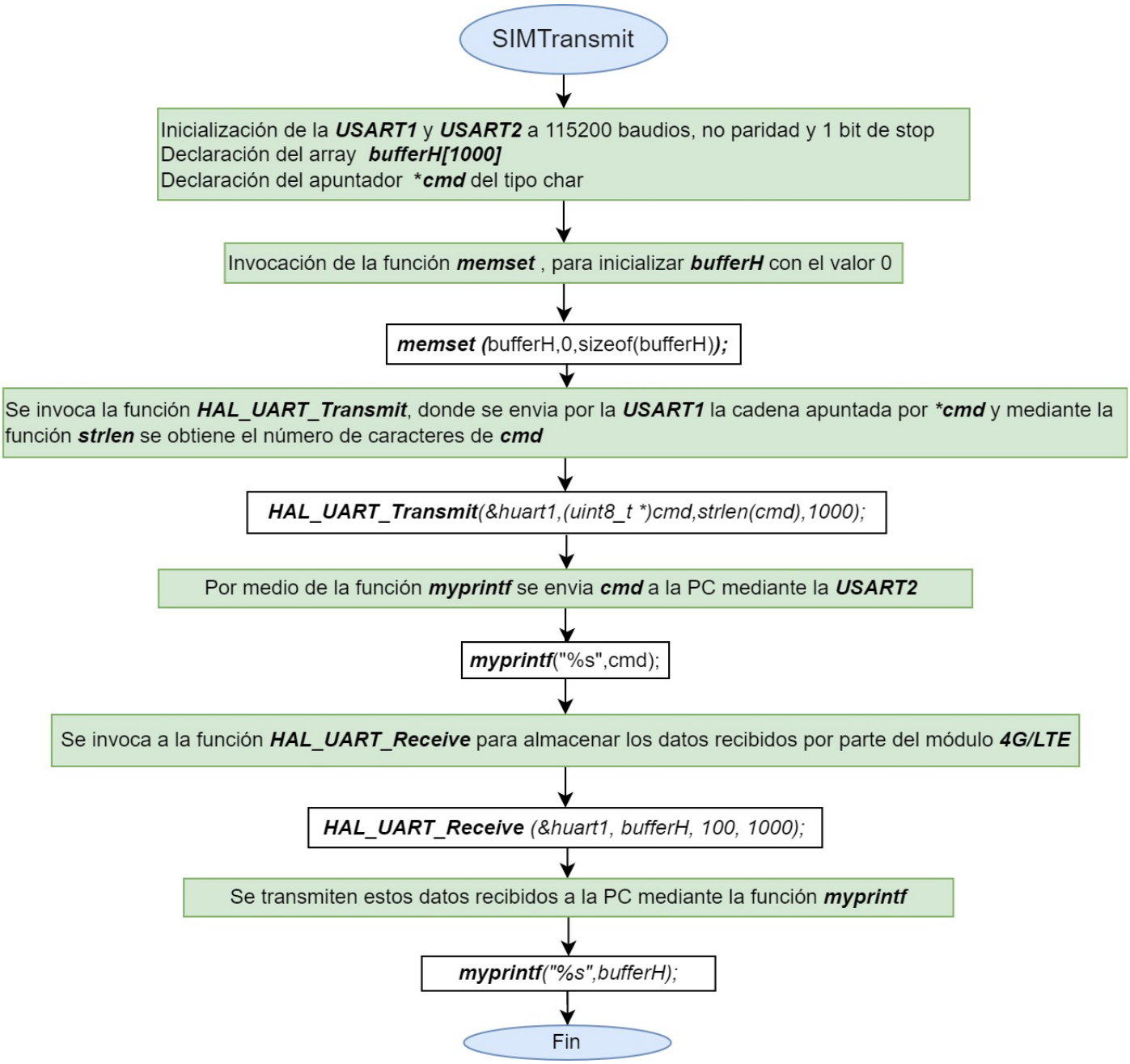


Figura 3.28: Algoritmo de la función *SIMTransmit*, que permite enviar cadenas de caracteres al módulo *4G/LTE* y guardar su respuesta en el *bufferH*.

Es importante aclarar que la función $HAL\_UART\_Transmit$ y $HAL\_UART\_Receive$ deben convertir la cadena *cmd* al tipo de dato $uint8\_t$ (número entero con signo de 1 byte), para esto se hace un *cast* como se observa en el código 3.15, en la línea 4:

```
1 void SIMTransmit(char *cmd)
2 {
3   memset(bufferH,0,sizeof(bufferH));
4   HAL_UART_Transmit(&huart1,(uint8_t *)cmd,strlen(cmd),1000);
5   myprintf("%s",cmd);
6   HAL_UART_Receive (&huart1, bufferH, 100, 1000);
7   myprintf("%s",bufferH);
8 }
```

Código 3.15: Declaración de la función *SIMTransmit*.

Para transmitir los parámetros del vehículo hacia *Firebase*, se tiene la *Función de envío de datos a la nube* de la figura 3.29, donde primero se declara el *array content*[1000], que guarda los datos obtenidos del *OBD-II* en formato *JSON*, después se crea el *array ATcommand*[1000], que tiene como propósito guardar los comandos *AT* y pasarlos a la función *SIMTransmit*. Se debe activar el protocolo *HTTP* del módulo *4G/LTE*, para esto, se envía el comando *AT+HTTPINIT* y se espera como respuesta *OK*, es importante espera 100 ms antes de enviar el siguiente comando *AT*, para esto se implementa un *delay*.

Antes de seguir explicando los demás comandos involucrados en el envío de datos, se muestra el formato que se debe seguir para enviar un mensaje por *HTTP*. Existen distintos métodos para transmitir/recibir datos en este protocolo, uno de ellos es *POST*, utilizado comúnmente para enviar datos en formato *JSON*. Este tiene la siguiente estructura, donde en color azul se resaltan las palabras reservadas:
POST /Seguido se específica el documento donde se almacena los datos, después va el signo de interrogación ? y la palabra reservada authorization= para entonces escribir la clave de la base de datos, terminado con HTTP/1.1
Host: Aquí se escribe la dirección *URL* pero sin *https://*
Content-Type: application/json (para indicar que los datos están en formato *JSON*)
Content-Length: Longitud del los datos
Datos a enviar.

Aquí se muestra un ejemplo:
POST /datos_auto.json?authorization=vrae1gf9Ji3B3eF03iwaD3V7LdHMEUVKhGYXvTd HTTP/1.1
Host: pruebacanbus-default-rtdb.firebaseio.com
Content-Type: application/json
Content-Length: 200
{ “*Bytes_adicionales*” : “4”, “*Modo*” : “1”, “*PID*” : “12”, “*Byte_A*” : “2”, “*Byte_B*” : “85”, “*Byte_C*” : “0”, “*Byte_D*” : “0”, “*Dato8*” : “85”, “*GPS*” : “*1900.497470,N,09812.365136,W,190722,224401.0,2135.0,0.0,*”, “*ID_Auto*” : “0001”, “*Identificador*” : “7E8” }

La anterior estructura se va realizando de la siguiente manera: para indicar la dirección *URL*, se usa el comando $AT + HTTPPARA = \text{“}URL\text{”}, \text{“}$ y seguido la cadena “$https://pruebacanbus-default-rtdb.firebaseio.com/datos\_auto.json$, esto se pasa a la función *SIMTransmit* para enviarlo al módulo *4G/LTE* y después se tiene el respectivo retardo de 100 ms antes de enviar el siguiente comando AT. Luego se específica que los datos obtenidos del *OBD-II* están en formato *JSON*, para esto se usa el comando $AT + HTTPPARA = \text{“}CONTENT\text{”}, \text{“}application/json$. Seguido se indica la clave de la base de datos, con el comando $AT + HTTPPARA = \text{“}USERDATA\text{”}, \text{“}authorization : key =$ la cual es: *vrae1gf9Ji3B3eF03iwaD3V7LdHMEUVKhGYXvTd*.

Antes de enviar lo guardado en *content*, se debe obtener su tamaño, para esto se hace uso de la función *strlen*, después este valor se concatena con el comando $AT + HTTPDATA =$ y se guarda en el *array ATcommand*, a través de la función *sprintf*. La respuesta por parte del *4G/LTE*, se guarda en *bufferH* y si corresponde a la cadena *Download* como se observa en la figura 3.29, entonces se procede a enviar *content* seguidamente, en caso contrario, ocurrió un problema al convertir los parámetros obtenidos del *OBD-II* al formato *JSON* y por ende, no se enviará el mensaje.

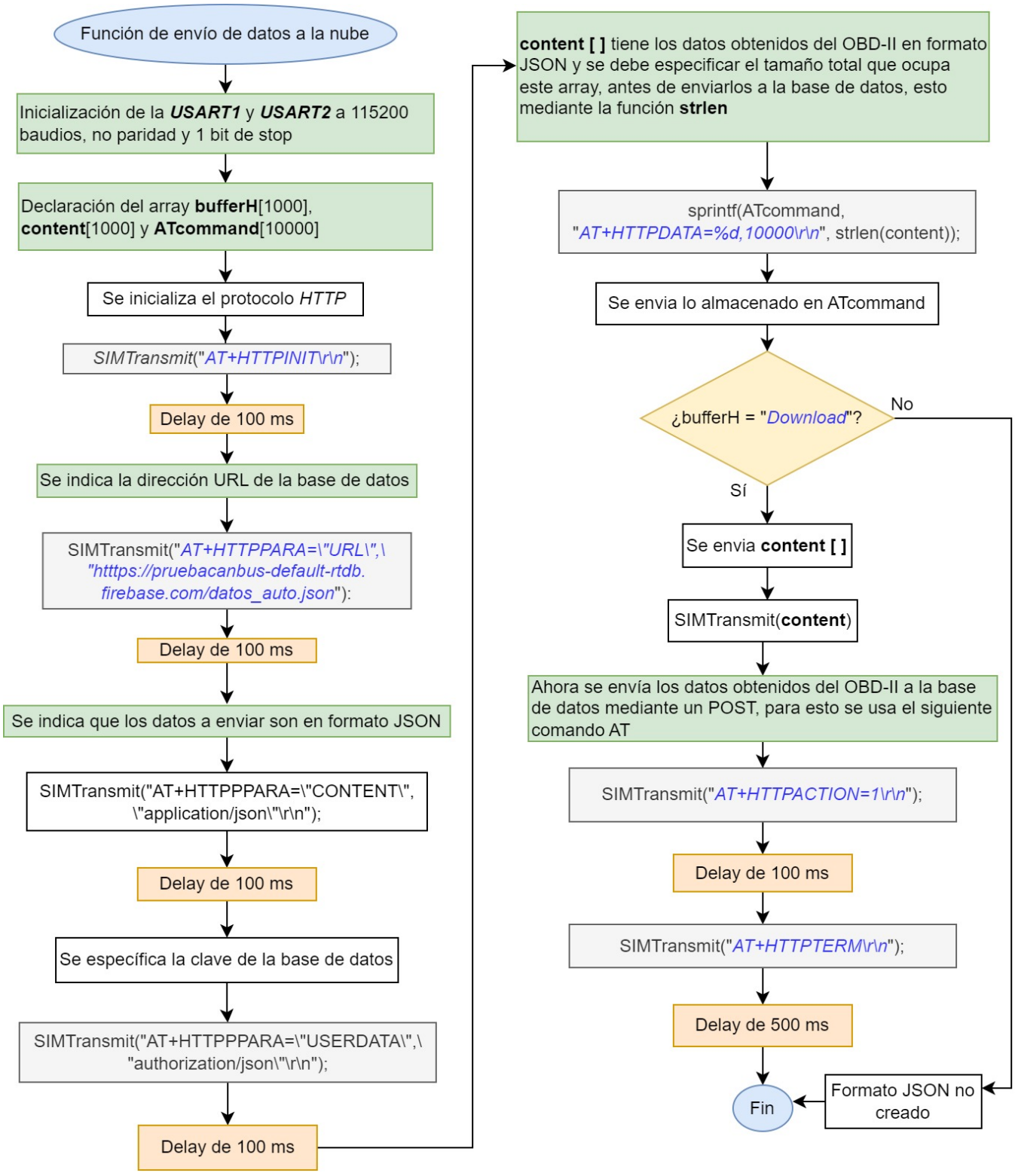


Figura 3.29: Algoritmo de la función de envió de los valores de los sensores hacia la base de datos.

Por último, se específica que se utiliza el método *POST*, con el comando $AT + HTTPACTION = 1$, que a su vez también manda el mensaje a la base de datos. Como se observa en la figura 3.29, se requiere esta vez un retardo de 500 ms antes de enviar el último comando AT. Se debe finalizar el protocolo *HTTP* con el comando $AT + HTTPTERM$ y otro *delay* de 500 ms.

El código 3.16 representa el algoritmo de la figura 3.37, donde la función *httpPOST* es la encarga

de ejecutar el envió de *content* a *Firebase*. Nótese como para realizar los retardos se recurre a la función *HAL_Delay()* y en la línea 23 de este código, se implementa el condicional *if* para determinar si el módulo *4G/LTE* ha respondido con la cadena *DOWNLOAD*, de ser correcto, entonces se procese a enviar el comando $AT + HTTPACTION = 1$, pero en caso contrario, el módulo responde con ERROR y no se enviará *content*.

```
1  //Función para enviar los datos del OBD-II hacia Firebase
2  void httpPost(void)
3  {
4  //Inicializa el protocolo HTTP
5  SIMTransmit("AT+HTTPINIT\r\n");
6    HAL_Delay(100);
7  //URL de la base de datos
8  SIMTransmit("AT+HTTPPARA=\"URL\",\"https://pruebacanbus-default-rtdb.firebaseio.com/
       datos_auto.json\"\r\n");
9       HAL_Delay(100);
10 //Se específica el formato de envió, que es en este caso JSON
11 SIMTransmit("AT+HTTPPARA=\"CONTENT\",\"application/json\"\r\n");
12      HAL_Delay(100);
13 //La contraseña para acceder a Firebase
14 SIMTransmit("AT+HTTPPARA=\"USERDATA\",\"authorization: key=
       vrae1gf9Ji3B3eF03iwaD3V7LdHMEUVKhGYXvTd\"\r\n");
15      HAL_Delay(100);
16      sprintf(ATcommand,"AT+HTTPDATA=%d,10000\r\n",strlen(content));
17 //Invocamos a la función SIMTransmit
18 SIMTransmit(ATcommand);
19 //Si se recibe DOWLOAD entonces ha concluido sastifactoriamente el envío de datos
20 if(strstr((char *)bufferH,"DOWNLOAD"))
21  {
22   SIMTransmit(content);
23   }
24     SIMTransmit("AT+HTTPACTION=1\r\n"); //POST
25     HAL_Delay(500);
26   // Cierre de conexiones
27 SIMTransmit("AT+HTTPTERM\r\n");
28       HAL_Delay(500);
29 }
```

Código 3.16: Codificación de la función *httpPost*, que sirve para enviar los datos del *OBD-II* hacia la base de datos.

3.7. Monitoreo de la obtención/transmisión de datos mediante la *USART2* y su visualización en la terminal *Teraterm*

Todo el proceso de envío de datos a la nube, se va a monitorear a través del puerto serie, para esto es necesario un convertidor de serial a *USB*, afortunadamente la *NUCLEO-F446RE* que tiene integrado el microcontrolador STM32F446RE tiene un *ST-LINK V2*, que permite programar microcontroladores *ARM Cortex-M* externos y realizar la conversión antes mencionada, como se observa en la figura 3.30. Se aclara que internamente ya viene conectados por defecto los pines de la *USART2* al convertidor de *USB*.

La función *myprintf* (ver código 3.15) puede funcionar de dos maneras, primero es escribir en el argumento de la función, la cadena a enviar por la *USART2*, por ejemplo:
myprintf("Enlazando comunicación con el módulo 4G/LTE....");
La segunda manera es escribir en el argumento, la marca de formato, seguido del array donde están los caracteres, como se observa en la línea 7 del código 3.15:
myprintf("%s",bufferH);

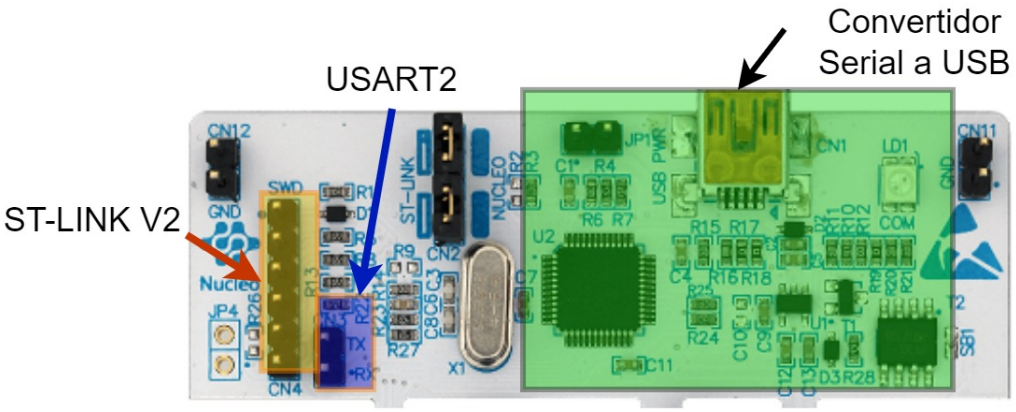


Figura 3.30: Posición de los pines del *ST-LINK V2*, *USART2* y ubicación del convertidor Serial a *USB* en la tarjeta NUCLEO-F446RE.

La figura 3.40, muestra el monitoreo de la obtención y transmisión de datos del *OBD-II*, visualizándose en la terminal *Tera Term*, esto es de gran utilidad para ver algún error en caso de que suceda.

Figura 3.31: Captura de pantalla del monitor serial *TeraTerm*, donde se visualiza los comandos AT y sus respectivas respuestas.

Además se añade el comando *AT+CCLK?* (ver figura 3.31) que obtiene la fecha y hora del *RTC* del módulo *4G/LTE*. La cual no está actualizada, por lo que debe ser sincronizada manualmente, sin embargo esta no se configura, puesto que la cadena de geolocalización proporciona el año, mes y día en curso. Entonces la justificación del porque se manda este comando, es debido a la hora que brinda el *RTC*, la cual empieza a ir incrementando desde cero cuando el módulo *4G/LTE* se energiza. Esto es importante para tener un registro de los intervalos de segundos que le toma obtener y enviar un dato desde el OBD-II.

3.8. Uso de *Vercel* para el desarrollo de la interfaz gráfica web (*Frontend*) que nos permite ver los datos obtenidos del *OBD-II*

Vercel es una empresa que ofrece un *SaaS* (software como servicio, ya explicado en el capítulo 2), este servicio permite al usuario acceder a una plataforma para vincular su código fuente almacenado en *Git Hub* y cuando este sea modificado mediante algún *IDE* como *Visual Studio*, observar los cambios realizados en un dominio (dirección *URL*) generada por *Vercel*, a esto se le conoce como *Deploy*, que significa ver el sitio web con las modificaciones realizadas en el código fuente, con la condición que esta página, este en internet y no es un servidor local, como típicamente se hace para pruebas. Como

se observa en la figura 3.32 se tiene creada una cuenta del tipo *Hobby*, siendo gratuita para el usuario, sin embargo se tiene como desventaja la *URL* que le pertenece a la empresa, aunque si se desea una propia, se deberá pagar por ello.

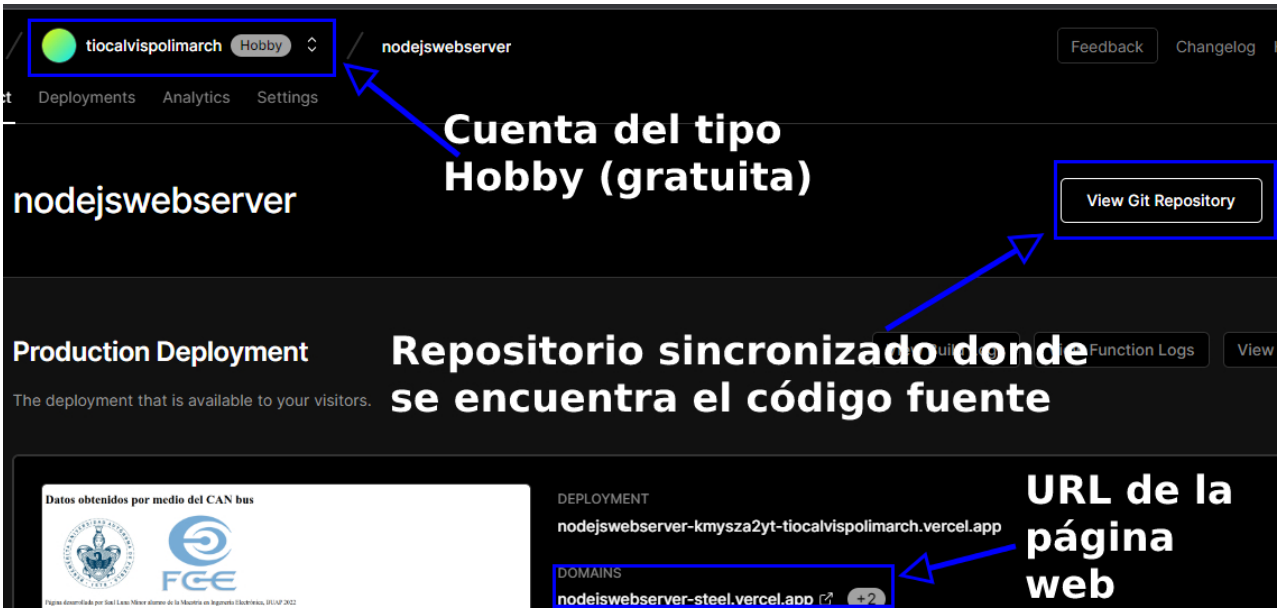


Figura 3.32: Vista del proyecto creado en *Vercel* para el desarrollo de la página web.

Entonces es primordial tener un repositorio en *Git Hub*, donde se guarda el archivo principal que es *index.html*, que tiene la codificación del *Frontend* (parte visual de una página web), allí se agrega la parte los títulos y subtítulos que lleva el sitio web, además de mostrar los datos gestionados en *Firebase* por medio de una tabla.

Visual Studio tiene el icono señalado de la figura 3.34, que sirve para subir (guardar en el repositorio los cambios realizados) el código fuente, estas modificaciones se reflejan en la página web, una vez que *Vercel* lo indique. Entonces para hacer uso de esta herramienta se debe sincronizar la cuenta de *Git Hub* en este *IDE*, después para editar el código *index.html*, se accede a la opción de repositorio remoto.

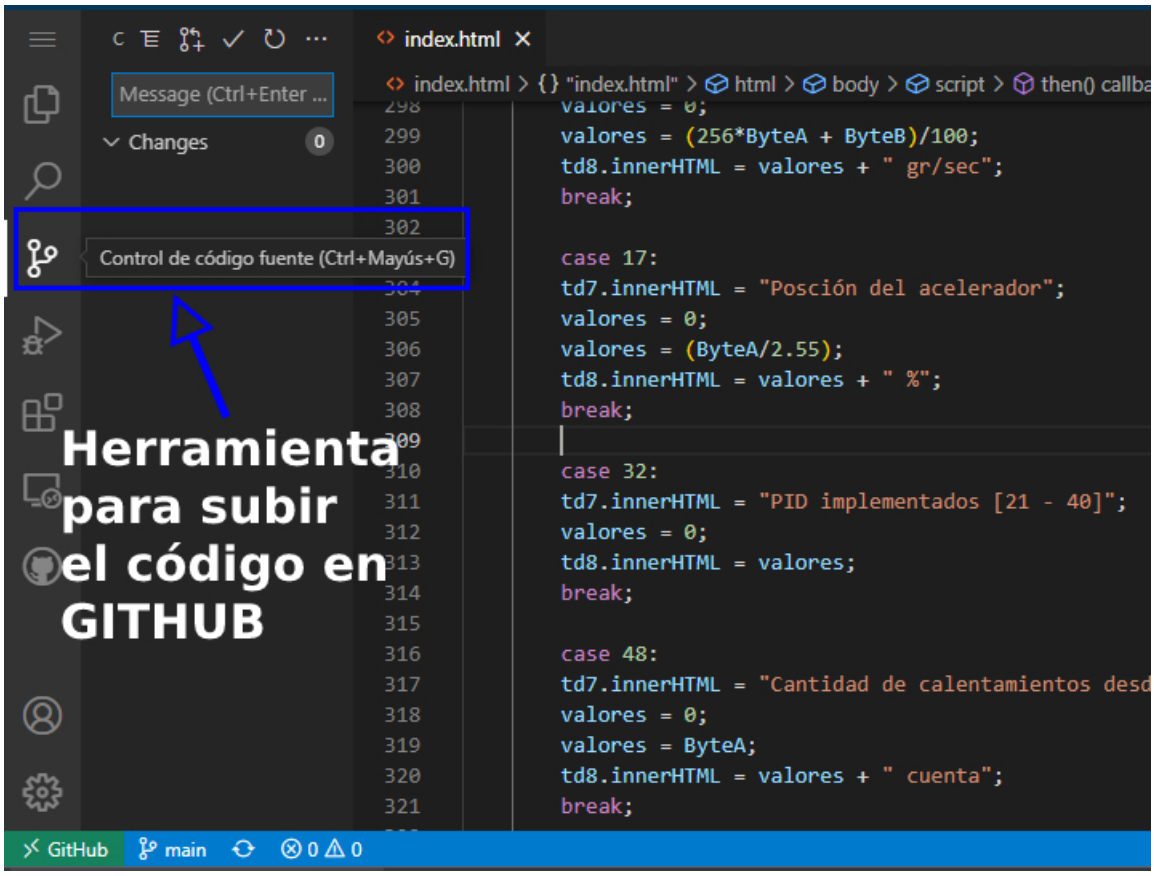


Figura 3.33: Vista del archivo *index.html* en el *IDE* de *Visual Studio Code*.

Cada vez que se modifica y se sube el repositorio *Nodejswebserver*, se debe nombrar a la versión de cambios, para esto se puede poner un nombre como *new7*, un simple número como 14.4 o bien se puede ser más específico, como indicar que cambios se realizaron, por ejemplo: *se añade las RPM y el gps*. Esto sirve para acceder a versiones previas, para el caso donde al modificar el código fuente, no se recuerde que se cambió y la página web no funcione correctamente, permitiendo entonces revisar códigos anteriores, para encontrar las fallas o bien corregirlas.

3.8.1. Importar datos desde *Firebase* a la página web

En la configuración del proyecto se encuentra un apartado donde proporciona código en lenguaje *JavaScript*, para obtener los datos almacenados en el gestor de *Firebase*. Si bien el archivo *index.html*, tiene formato *HTML*, el cual sirve para dar un formato a la página web, el lenguaje *JavaScript* tiene como propósito dar funcionalidad a la página, por ejemplo, al oprimir un botón se ejecuta una cierta acción, como guardar un archivo, enviar un mensaje, etc. En el código 3.17 se muestra un fragmento del archivo *index.html*, donde se empieza a programar en lenguaje *HTML*, declarando los títulos, subtítulos, colores, etc. Después para incluir el código 3.19 que permite acceder a la base de datos, se escribe $< script\ type = \text{“}module\text{”} >$ para indicar que las siguientes líneas van a ser en *JavaScript* y se pega lo proporcionado por *Firebase*. Para declara que se ha terminado de programar con este lenguaje, se hace uso de $< /script >$.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
</div>
        .
        .
        .

```

```
<script  type="module">
import { getDatabase, ref, child, get  }
  from "https://www.gstatic.com/firebasejs/9.6.10/firebase-database.js";
   // Import the functions you need from the SDKs you need
  import { initializeApp } from "
        .
        .
        .
</script>
</body>
</html>
```

Código 3.17: Estructura del código en *HTML* y *JavaScript* del archivo *index.html*.

Entonces el código 3.18 se indica los siguiente parámetros:

- Clave de la base de datos.
- Dominio del proyecto creado en *Firebase*.
- *URL* de la base de datos.
- *ID* del proyecto.

Además se añaden otros como son *storageBucket*, *messagingSenderId*, *appId* y *measurementId*. Estos son generados por *Firebase* y son necesarios para la importación de datos, sin embargo no se indagará en ellos, solo se deja en claro que no deben ser modificados. Los comentarios están en color verde y vienen seguido por //.

```
  const firebaseConfig = {
    apiKey: "AIzaSyBrovLUtIff15JyTx_7FbifhO5QBA2YKgM",
    authDomain: "pruebacanbus.firebaseapp.com",
    databaseURL: "https://pruebacanbus-default-rtdb.firebaseio.com",
    projectId: "pruebacanbus",
    storageBucket: "pruebacanbus.appspot.com",
    messagingSenderId: "391775351194",
    appId: "1:391775351194:web:3ac9a2a73675f21c9dca23",
    measurementId: "G-1ZVR73393X"
  };
  // Initialize Firebase
  const app = initializeApp(firebaseConfig);
  const db = ref(getDatabase());
```

Código 3.18: Declaración de las variables, funciones y constantes para acceder a la base de datos en *Firebase*.

El algoritmo para mostrar los valores de los sensores obtenidos en la base de datos mediante una página web se muestra en la figura 3.34, donde la constante *db* tiene todos los objetos de la base de datos, sin embargo solo nos interesa *datos_auto* ya que esta tiene lo obtenido por el *OBD-II*. Entonces si este objeto está vacío, se termina el algoritmo y no se muestra ningún dato. En caso contrario, se imprime por consola todo lo contenido en este y después se pasan los identificadores a la variable *data*.

El propósito de realizar lo anterior, es de guardar en dichas variables, lo obtenido en ese momento del objeto *datos_auto*, ya que al crear la tabla donde se mostrarán los parámetros de *OBD-II*, se definen 14 columnas, pero las filas vienen dadas por el número de valores guardados en cada identificador, por eso el algoritmo de la figura 3.34, está indefinido el número de filas. Entonces para llenar la tabla, el primer dato se enumera, esto se indica en la primera columna. Para la segunda, se obtiene el *ID* del automóvil. En la tercera va el modo y así consecutivamente, hasta llegar al *PID*, donde se tiene que reconocer que parámetro se está consultado.

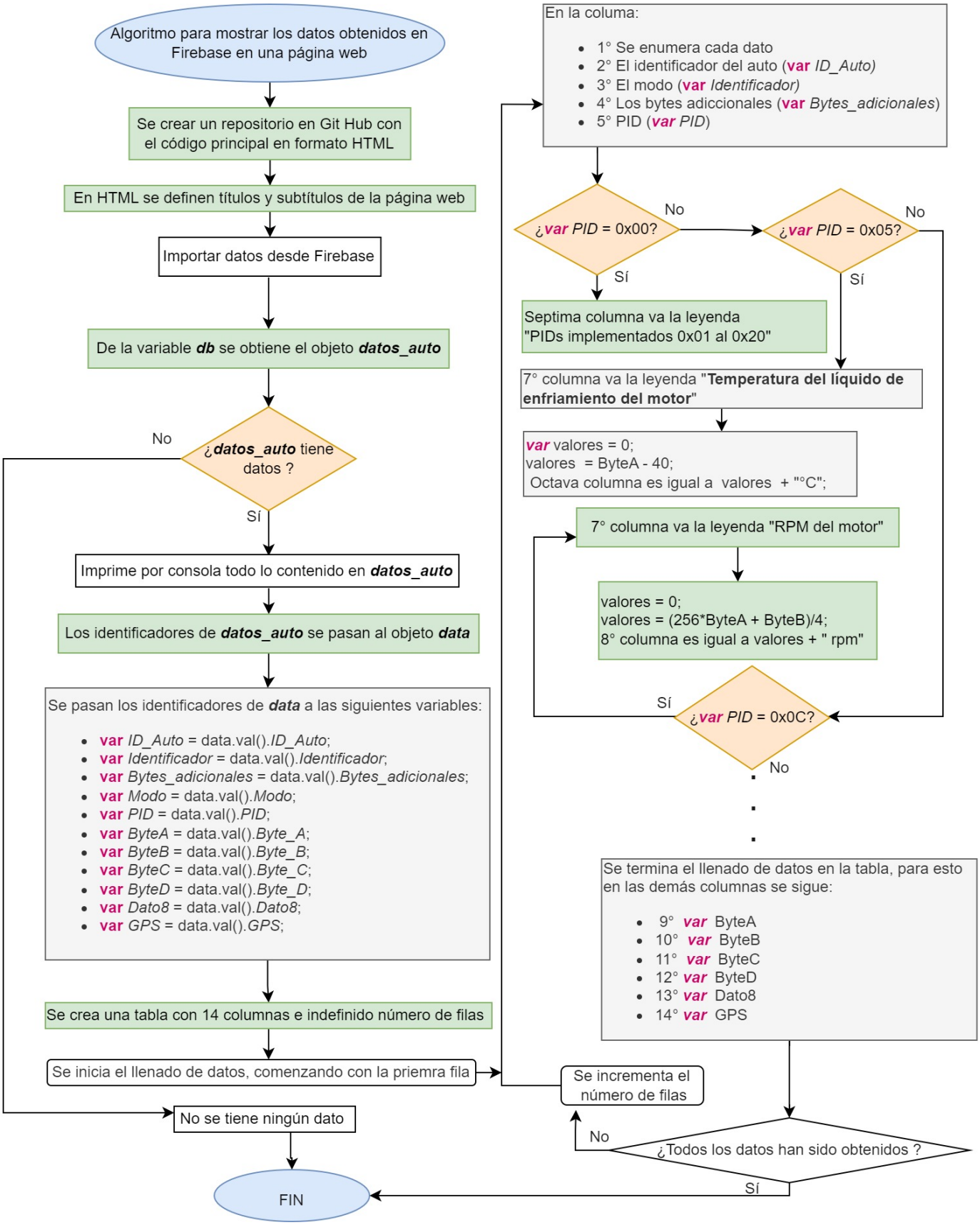


Figura 3.34: Algoritmo para obtener los valores almacenados en la base de datos y mostrarlos en una tabla por medio de una página web.

Por ejemplo si el *PID* es igual a 0x05, en la 7° columna va una leyenda que indica que se consul-

ta la *temperatura del líquido de enfriamiento del motor*. Después se tiene la variable *valores*, la cual sirve para convertir los datos de dicho parámetro en su representación de unidad y para este PID en particular, se espera su valor en el $Byte\_A$, luego se le resta 40, obteniendo así como resultado el valor en grados centígrados, el cual se representa en la 8° columna. Se puede dar el caso donde se consulte el PID 0x00, el cual muestra que parámetros se pueden consultar desde el PID 0x01 hasta el 0x20. Entonces no se realiza ninguna operación de conversión, por tanto la 8° columna no tiene valor o bien puede tener cero. Una vez que termina la comparación se procede a llenar las demás columnas con los bytes restantes del protocolo *CAN bus* como se observa en la figura 3.34. Por último se consulta si todos los datos han sido obtenidos, en caso de que falten, se incrementan filas y se sigue el procedimiento de llenado de columnas. Una vez completo, se procede a finalizar el algoritmo, aunque si la página web se recarga y la base de datos ha sufrido modificaciones, entonces las filas pueden reducir o aumentar, es por esto que no se definen cuando se crea la tabla.

La codificación del algoritmo de la figura 3.34, se observa se muestra en el código 3.19, el cual es un fragmento del archivo *index.html*. De la línea 1 hasta la 4, es una función proporcionada por *GoogleCloud* [85] para obtener los datos del objeto *datos_auto* en la variable *data*. Entonces *data* se maneja como una clase donde sus objetos son los identificadores. Para acceder a ellos se hace de la siguiente forma:
$data.val().PID$;
Accediendo a todos los PID guardados. Estos se van pasando a las variables ya mencionadas, como se observa desde la línea 6 hasta la 17.

Desde las líneas 18 hasta la 23 (ver código 3.19), se crea la tabla, para esto se tiene la variable *trow*, que será la encarga de ir incrementando las filas y de las variables $td1$ hasta $td14$ irán las columnas, siendo 14 en total. La variable *stdNo* incrementará su valor conforme se llene la tabla, siendo la encargada de llevar el conteo de las filas. La palabra reservada $innerHTML$ permite colocar el valor en cada columna, por ejemplo, si se declara $td4.innerHTML =$ Modo, y el valor de Modo es 1, entonces en la 4° columna aparece ese valor. Por último se hace el llenado de las demás filas, para esto se hace uso de la variable antes mencionada *trow*, la cual con la palabra reservada *appendChild*, llena las filas con el valor que se le indique, por ejemplo, para $trow.appendChild(td14)$;, llenara en la columna 14, todas sus filas con el valor del GPS, pero esto no funciona de forma simultánea, siendo importante declarar $body.appendChild(trow)$;, que incrementará el conteo de las filas, hasta llegar hasta la última.

```
1  get(child(db, `datos_auto/`)).then((snapshot) => {
2          if (snapshot.exists()) {
3            console.log(snapshot.val());
4            snapshot.forEach(function(data)
5      { var  tbody = document.getElementById('tbody1');
6      var ID_Auto = data.val().ID_Auto;
7      var Identificador = data.val().Identificador;
8      var Bytes_adicionales = data.val().Bytes_adicionales;
9      var Modo = data.val().Modo;
10     var PID = data.val().PID;
11     var valores = [];
12     var ByteA = data.val().Byte_A;
13     var ByteB = data.val().Byte_B;
14     var ByteC = data.val().Byte_C;
15     var ByteD = data.val().Byte_D;
16     var Dato8 = data.val().Dato8;
17     var GPS = data.val().GPS;
18         let trow = document.createElement('tr');
19     let td1 = document.createElement('td');
20                         .
21                         .
22                         .
23     let td14 = document.createElement('td');
24     td1.innerHTML = ++stdNo;
```

```
25      td2.innerHTML = ID_Auto;
26      td3.innerHTML = Identificador;
27      td4.innerHTML = Modo;
28      td5.innerHTML = Bytes_adicionales;
29      td6.innerHTML = PID;
30      switch (PID) {
31      case 0:
32      td7.innerHTML = "PIDs implementados 0x01 al 0x20";
33      break;
34                  .
35                  .
36                  .
37      default:
38      td7.innerHTML = "Velocidad del flujo del aire MAF";
39      valores = 0;
40      valores = (256*ByteA + ByteB)/100;
41      td8.innerHTML = valores + " gr/sec";
42      break;}
43                  .
44                  .
45                  .
46      td14.innerHTML = GPS;
47      trow.appendChild(td1);
48      trow.appendChild(td2);
49                  .
50                  .
51                  .
52      trow.appendChild(td14);
53      tbody.appendChild(trow);})
54              } else {
55            console.log("No data available"); }
56
```

Código 3.19: Fragmento de las declaraciones necesarias para crear la tabla con los datos obtenidos del *OBD-II* y del *GPS*

En la figura 3.35 se observa la vista de la página web, donde además se agregó los logos de la Benemérita Universidad Autónoma de Puebla y de la Facultad de Ciencias de la Electrónica. Se aclara que esta es una versión previa y la final se puede acceder en el siguiente link: https://nodejswebserver-steel.vercel.app/

3.8.2. Conclusión

Hasta este momento, donde se emuló el *OBD-II*, para hacer pruebas del firmware sin la necesidad de tener un vehículo físico. Después se envían los datos obtenidos a la nube, mediante el módulo *4G/LTE* y se hace uso de *Firebase* como gestor de la base de datos. En caso de fallar la comunicación con la red *4G/LTE*, se tiene un módulo para micro SD, con un almacenamiento máximo de 4 GB. La visualización de los datos es mediante una página web, para esto se usó *Vercel*, ya que permite guardar los cambios en el código fuente y a su vez visualizarlos.

Sin embargo, para realizar pruebas en un automóvil real, se requiere de una tarjeta de desarrollo, que tenga el conector del *OBD-II* y las características eléctricas para establecer comunicación con el *CAN bus*. También se integró un microcontrolador *ARM Cortex-M* con todos los periféricos usados hasta el momento, además de tener conectores, micro SD y LEDs que faciliten al usuario interactuar con el vehículo. Entonces en el siguiente capítulo se muestra el desarrollo de esta tarjeta y la metodología que se sigue para su diseño.

nodejswebserver-steel.vercel.app

Datos obtenidos por medio del CAN bus

Página desarrollada por Saul Luna Minor alumno de la Maestría en Ingeniería Electrónica, BUAP 2022

Exportar a Excel

Datos CAN bus 2022

Número	ID Auto	Identificador	Bytes adicionales	Modo 1	PID	Sensor	Valor	Byte A	Byte B	Byte C	Byte D	Dato 8	GPS
1	0001	000	1	3	10	Velocidad del flujo del aire MAF	141.58 gr/sec	55	78	0	41	100	1900.179309,N,09812.162652,W,010322,213014.0,2119.5,0.0,
2	0001	000	1	3	0	PIDs implementados 0x01 al 0x20		55	78	0	41	100	1900.179309,N,09812.162652,W,010322,213014.0,2119.5,0.0,
3	0001	7E8	1	6	0	PIDs implementados 0x01 al 0x20		190	31	236	19	85	,,,,,,,,
4	0001	7E8	1	3	5	Temperatura del líquido de enfriamiento del motor	56°C	96	0	0	0	85	,,,,,,,,
5	0001	7E8	1	4	12	RPM del motor	149.25 rpm	2	85	0	0	85	,,,,,,,,
6	0001	7E8	1	3	13	Velocidad del vehículo	69 km/h	69	0	0	0	85	,,,,,,,,

Figura 3.35: Vista preliminar de la página web.

Capítulo 4

Desarrollo del *hardware* del sistema de recolección de datos del OBD-II

Una vez que se ha probado todo el firmware mediante el prototipo, se procederá a diseñar una placa de circuito impreso (*PCB*) que disponga de conectores y circuitos que faciliten la interacción con el vehículo. Es necesario tener una claridad de los requerimientos eléctricos para el diseño del hardware, permitiendo así seleccionar que tipos de componentes se requieren, tamaño de pista, diámetro de vías (una vía es un conductor circular que permite interconectar dos señales en diferentes capas), cantidad de capas del *PCB* y su tamaño. Para lo anterior se sigue la metodología de la figura 4.1, definiendo un presupuesto, después se enlistan los requerimientos eléctricos:

1. Se debe tener el conector macho del *OBD-II*, para establecer comunicación con el *CAN bus*.
2. El transceptor *CAN bus* y el módulo *4G/LTE* se deben alimentar con 5 V.
3. El microcontrolador, la micro SD y los *leds* deben estar alimentados a 3.3 V.
4. El módulo *4G/LTE* debe tener conectores para la comunicación serial y su correspondiente alimentación.
5. El microcontrolador *ARM Cortex-M* debe tener un oscilador y su valor dependerá de la frecuencia máxima a la que se desee trabajar.
6. Mediante el *ST-LINK V2* se carga el firmware, por ende debe tener los pines correspondientes para conectarse al programador.
7. Se deja un conector *USB* (de aquí se puede obtener alimentación de 5 V) y pines para cargar un *bootoloader*.
8. Existen pines del microcontrolador que no se están utilizando, pero si en trabajos a futuro se requieren, se dejan los correspondientes conectores.

Se tuvo la asesoría de la empresa *INTESC*, líder de desarrollo electrónico en Puebla centro, que además será la encargada de supervisar, mandar a fabricar el *PCB* a su empresa socia en China (PCBWay) y ensamblar la tarjeta. Entonces de acuerdo al presupuesto planteado, recomiendan un diseño en dos capas, considerando buscar componentes de preferencia *SMD*, para que la tarjeta final no sea mayor a 10x10 cm y sea considerado como *prototipo* por la compañía colaboradora en China, en caso contrario los costos aumentan considerablemente, saliendo del presupuesto previsto.

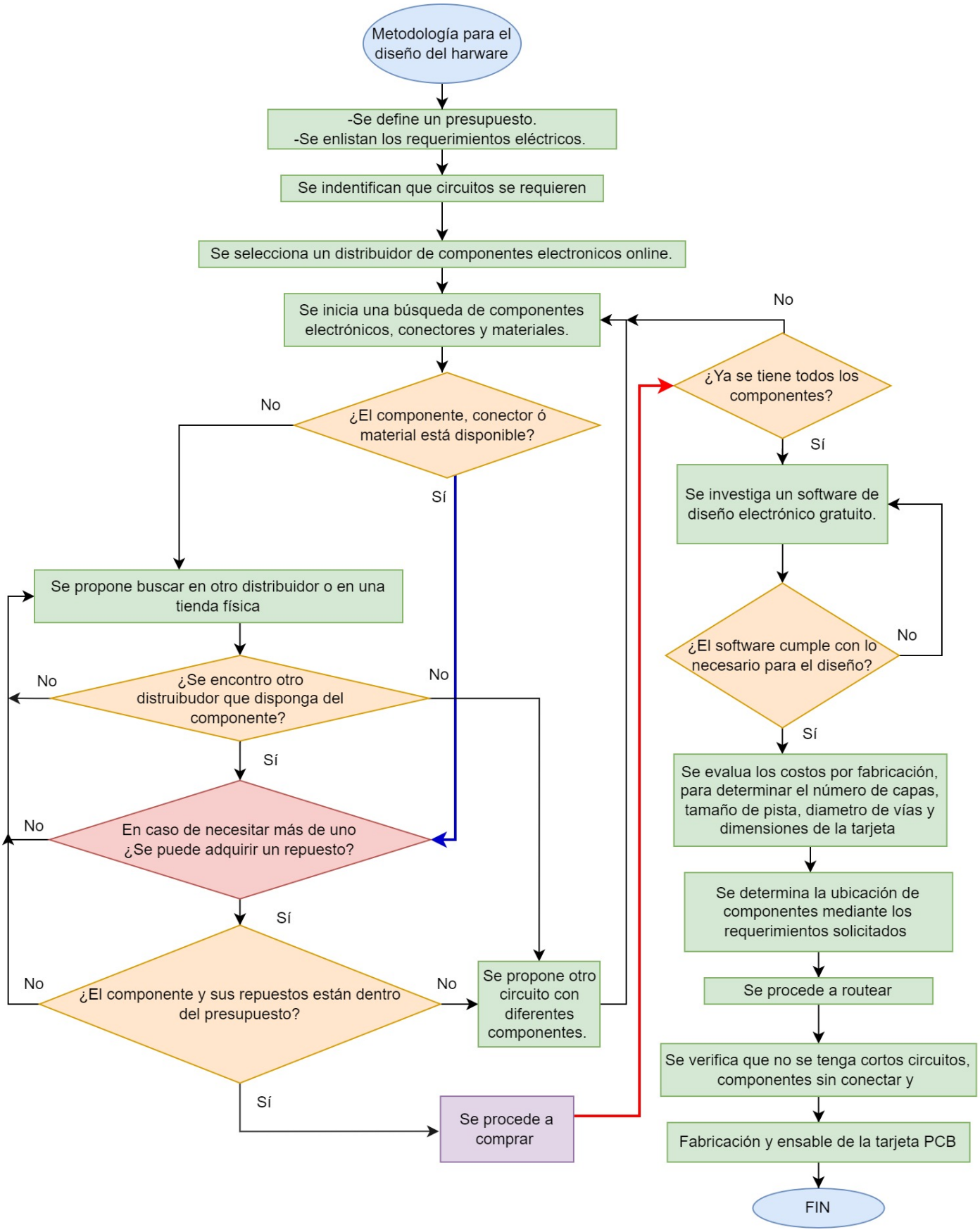


Figura 4.1: Diagrama de la metodología a seguir para seleccionar los circuitos y sus componentes para el desarrollo del SADO.

El desarrollo del esquemático y del *PCB* se realizará en el software *ALTIUM 22* en su versión estudiantil, la cual ofrece 6 meses de uso gratuito. Antes de profundizar en el diseño del *PCB*, se muestra el esquemático empleado.

4.1. Diseño del circuito eléctrico del sistema de adquisición de datos del *OBD-II* (SADO)

Se selecciona el microcontrolador *STM32F103C8T6* [86], ya que este tiene dos módulos *USART* (uno para el *4G/LTE* y otro para el monitor serie), un periférico *SPI* para la memoria SD y *CAN bus* para establecer comunicación por medio del *OBD-II*, además de tener un repuesto de este integrado. La figura 4.2 muestra las conexiones de voltaje a este microcontrolador y los periféricos involucrados.

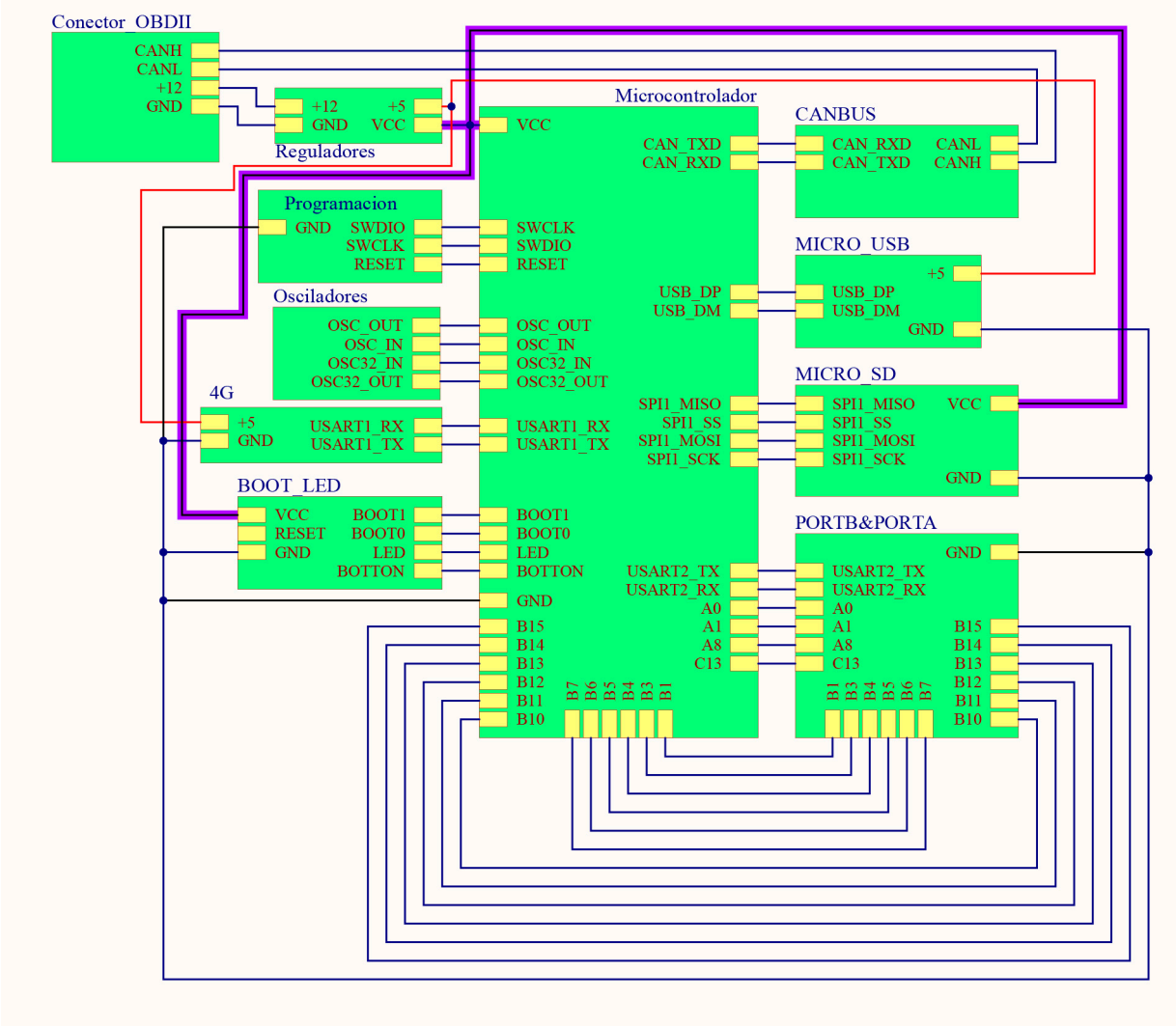


Figura 4.2: Diagrama de las conexiones de voltaje, periféricos (*SPI*, *CAN bus*, *USART*, etc.) y pines del microcontrolador *STM32F103C8T6*.

Se necesita para alimentar este microcontrolador 3.3 V (se hace referencia en este documento como *VCC*), obteniendo este voltaje de dos maneras, la primera por el conector del *OBD-II*, donde se obtienen los 12 V de la batería del vehículo, solo se requiere un regulador de voltaje que pase de 12 V a 5 V y después de 5 V a 3.3 V. La segunda forma es mediante el conector micro *USB*, que se observa en la figura 4.2, donde se establece conexión con el puerto *USB* del microcontrolador, alimentando con 5 V al transceptor *CAN bus* y al regulador de voltaje de 5 V a 3.3 V.

Después se tiene dos cristales, uno para el microcontrolador y otro para su *RTC* (reloj en tiempo real). El módulo *4G/LTE* requiere 5 V para su funcionamiento y conectarse a la *USART1*, dejándose los conectores correspondientes y también para el *ST-LINK V2* será el encargado de programar el microcontrolador. La micro *SD* se controla por medio del protocolo *SPI*, sin embargo se debe revisar que conector se usará, ya que existen diferentes modelos existentes en el mercado.

4.1.1. Selección de los circuitos electrónicos de acuerdo a los protocolos y módulos utilizados

En la figura 4.5 se muestra el esquemático del SADO y la lista de componentes está en la tabla 4.1, además se muestran todos los pines del microcontrolador *STM32F103C8T6* empleados, donde la asignación de estos fue establecida por los circuitos requeridos, de la siguiente forma:

- Para el regulador de voltaje se está utilizando dos integrados, el primero es el *MC7805ACD2TG* que a su entrada llegan los 12 V de la batería del automóvil mediante el conector del *OBD-II* y a su salida proporciona 5 V. Seguido esta el integrado *AMS1117* que regula de 5 V a 3.3 V. Para ambos se revisó su *datasheet* [87] [88], donde específica que tipo de capacitores requieren.
- Si bien el microcontrolador requiere 3.3 V, es importante revisar su *datasheet* [86] donde específica que para cada *PIN* nombrado con *VDD* requiere un capacitor de 100 nF (0.1 μF), por eso se utilizan 4, que son C1, C2, C3 y C4.
- Con base al circuito consultado [89], para el *USB* es necesario una resistencia de 20 Ω conectada a *USB_DP* (pin *PA11*) y otra para *USB_DM* (pin *PA12*), sin embargo al no encontrar dicho valor se usaron dos resistencias en serie de 10 Ω. El conector usado es el *10118194-000ILF*, donde se revisó sus especificaciones [90] para conectar correctamente tierra y 5 V, evitando así una conexión en polaridad inversa, que dañe los demás componentes.
- El módulo *4G/LTE* se conecta a la *USART1*, cuyos pines son: *PA9* para *USART1_TX* y *PA10* para *USART1_RX*.
- Se tienen dos botones, para cada uno se tiene un circuito *RC* o anti rebotes, formado por una resistencia de 10 kΩ y un capacitor de 1 μF. Un botón va al *reset* (pin 7) y otro al pin *PB0*, para propósitos de uso en general. Ambos alimentados por 3.3 V.
- Un *LED* va conectado a *VCC*, para indicar que se tiene alimentación en el microcontrolador y que funciona el integrado *AMS1117*. Otro led va conectado al pin *PA1*, para fines de uso por parte del usuario. Cada *LED* tiene una resistencia de 1 kΩ.
- Para el *bootloader*, se tienen un *header* macho de 6 pines, colocados en dos columnas de 3 pines cada uno, como se muestra en la figura 4.5, donde *BOOT0* va al pin 44 y *BOOT1* al pin *PB2*.
- El conector del *OBD-II* (ver figura 4.3) tiene los pines 5, 6 y 13 conectados a la tierra de la batería, mientras que el 12 proporciona 12 V. En el pin 14 corresponde a *CANL* y el 6 a *CANH*.
- El circuito del *CAN bus* tiene un transceptor *MCP2551*, con su correspondiente resistencia de 10 kΩ al pin *RS*, además de otra en paralelo de 120 Ω a *CANL* y *CANH*. Se le agrega una *clema* a estos dos cables del *CAN bus* (ver figura 4.6), esto sirve para observar dicho protocolo y agregar una resistencia de 120 Ω por si el *OBD-II* del vehículo no la tenga integrada. Se aclara que *CAN_TXD* y *CAN_RXD* son las señales que van desde el microcontrolador al transceptor y estas no tienen los niveles lógicos que maneja el *CAN bus*.
- La micro SD por su parte, requiere del protocolo *SPI* para ser controlada, sin embargo se debe pasar por una compuerta triestado, usando la *SN74LVC125AIP*, donde cada pin de este protocolo va acompañado de una resistencia de 3.3 kΩ (R14, R15, R16 y R17). Mientras que el conector de la SD, es el *MEM2061-01-188*, donde *DAT1*, *DAT2*, *CD1* y *CD2* no son utilizados, esta consideración se hizo con base en el circuito [91], que es ampliamente utilizado para módulos SD y de uso libre.

- Se utiliza un cristal de 16 MHz, ya que en la figura 4.4, se muestra la estructura de sistema de reloj de este microcontrolador y al emplear dicho componente se obtiene la máxima frecuencia de operación que es de 72 MHz. El *datasheet* [86] de este microcontrolador recomienda utilizar un cristal de 32.75 kHz para el *RTC*. Ambos componentes deben llevar dos capacitores conectados en cada pin de 20 pF, siendo estos C5, C6, C7 y C8.

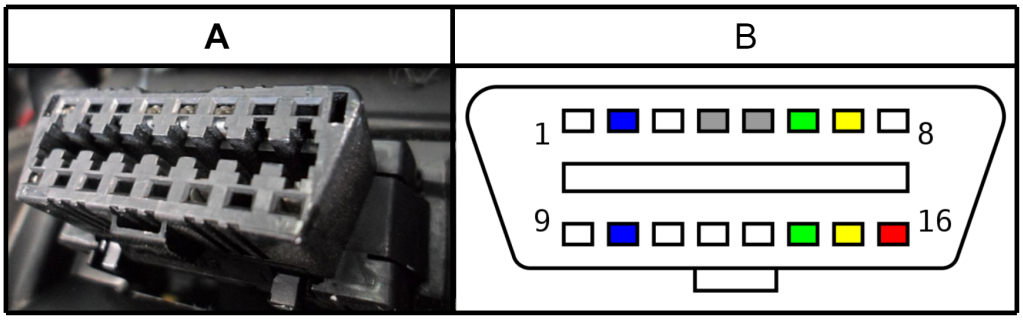


Figura 4.3: En el lado A se muestra el conector hembra del *OBD-II* y en B se muestra la numeración de sus pines [10].

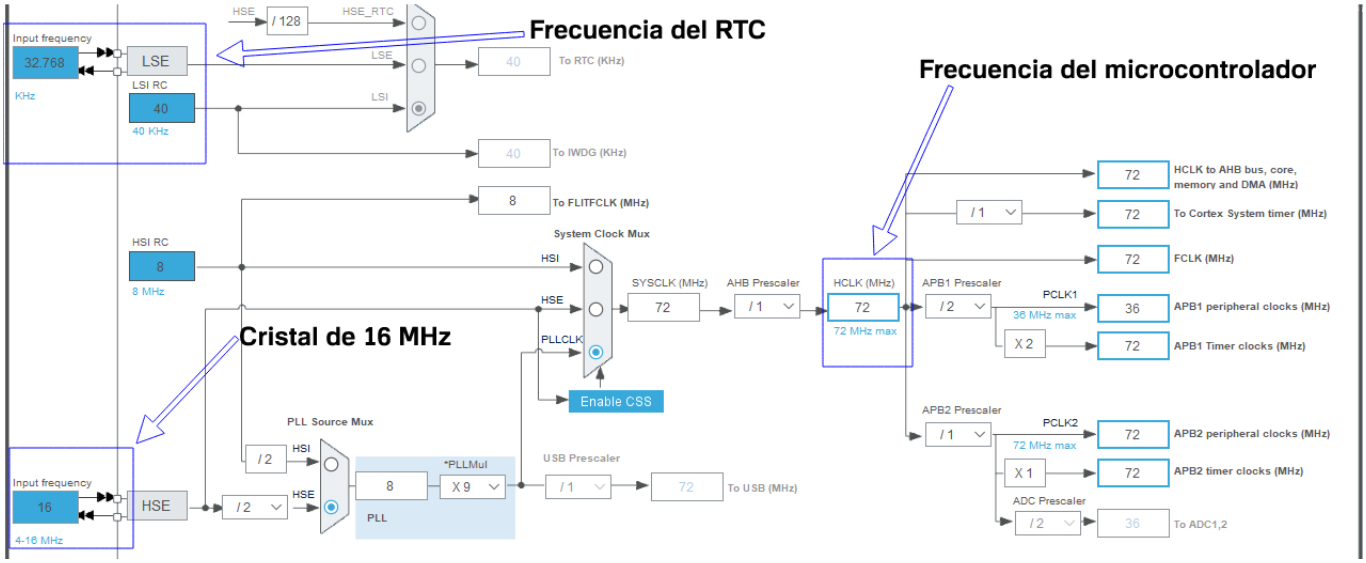


Figura 4.4: Estructura del sistema de reloj del microcontrolador *STM32F103C8T6*.

4.2. Ubicación de componentes en el *PCB* del prototipo

Entonces *Altium* permite ver la tarjeta en un modelo 3D como se observa en la figura 4.6, esto con el fin de acomodar los componentes dentro del tamaño mencionado. Para conectarse con el *OBD-II* hembra del vehículo, el conector macho del PCB debe ubicarse en un extremo como se muestra en la figura 4.6, mientras del otro lado se ubica los dos botones para que el usuario pueda presionarlos fácilmente, en medio de ellos esta el puerto *USB*, la entrada para la micro SD y los pines del *bootloader*, permitiendo que el usuario acceda a ellos cómodamente, sin interferir con otros elementos de la tarjeta.

La *clema* esta ubicada a un extremo para que se pueda conectar los cables de un osciloscopio y ver las señales del protocolo *CAN bus* o bien se conecte una resistencia de 120 Ω (ver figura 4.6). Los demás componentes como capacitores, resistencias y cristales, están cercanos a sus correspondientes

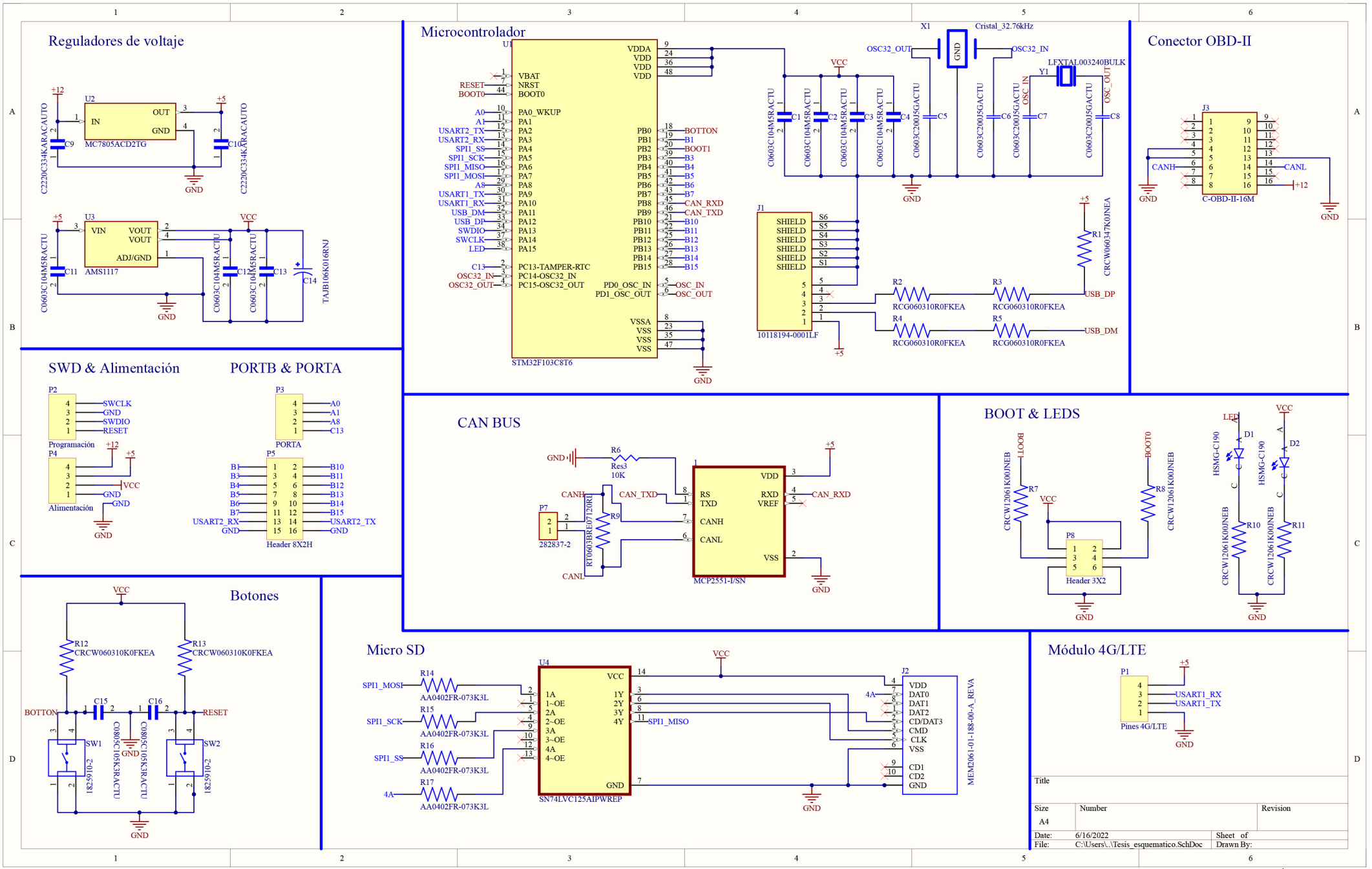


Figura 4.5: Circuito eléctrico completo del SADO.

Tabla 4.1: Lista de materiales del circuito eléctrico del SADO.

Cantidad	Material SMD	Valor	Tipo de encapsulado	Designator
Capacitores				
7	C0603C104M5RACTU	0.1 μF	0603	C1, C2, C3, C4, C11, C12, C13
4	C0603C200J5GACTU	20 pF	0603	C5, C6, C7, C8
2	C2220C334KARACAUTO	0.33 μF	2220	C9, C10
1	TAJB106K016RNJ	10 μF	1210	C14
2	C0805C105K3RACTU	1 μF	0805	C15, C16
Resistores				
1	CRCW060347K0JNEA	47 kΩ	0603	R1
4	RCG060310R0FKEA	10 Ω	0603	R2, R3, R4, R5
1	Res3	10 kΩ	0603	R6
4	CRCW12061K00JNEB	1 kΩ	1206	R7, R8, R10, R11
1	RT0603BRE07120RL	120 Ω	0603	R9
2	CRCW060310K0FKEA	10 kΩ	0603	R12, R13
Semiconductores				
2	MCP2551-I/SN	Tranceptor CAN	SOIC127P599X175	1
2	HSMG-C190	LED	1608	D1, D2
1	STM32F103C8T6	Microcontrolador ARM-Cortex M3	QFP50P900X900	U1
1	AMS1117	Regulador 5 V - 3.3 V	SOT229P700X180	U3
1	SN74LVC125AIPWREP	Compuerta Buffer cuadruple	SOP65P640X120	U4
Conectores SMD				
1	Micro USB		10118194-0001LF	J1
1	Connector microSD		MEM2061-01-188	J2
Material THT				
1	Conector OBD-II macho		OBD-II-16M	J3
1	Pines 4G/LTE		Header, 1x4 Pines	P1
1	Programación		Header, 1x4 Pines	P2
1	PORTA		Header, 1x4 Pines	P3
1	Alimentación		Header, 1x4 Pines	P4
1	Header 8X2H		Header, 2x8 Pines	P5
1	Clema		282837-2	P7
1	Header 8X2H		Header, 2x3 Pines	P8
2	Push-Botton		1825910-2	SW1, SW2
1	Cristal_32.76kHz	Cristal de 32.76 kHz		X1
1	LFXTAL003240BULK	Cristal de 16 MHz	HC49P488W45	Y1

integrados. Para los capacitores *C1*, *C2*, *C3* y *C4*, de 100 nF, deben estar lo más cercanos al microcontrolador [92] [93], como se muestra en la figura 4.7, puesto que estos se ponen para cada *VDD* del microcontrolador y tienen la función de minimizar la inductancia parásita de las pistas del *PCB*, dejando de funcionar correctamente al alejarse [92] [93].

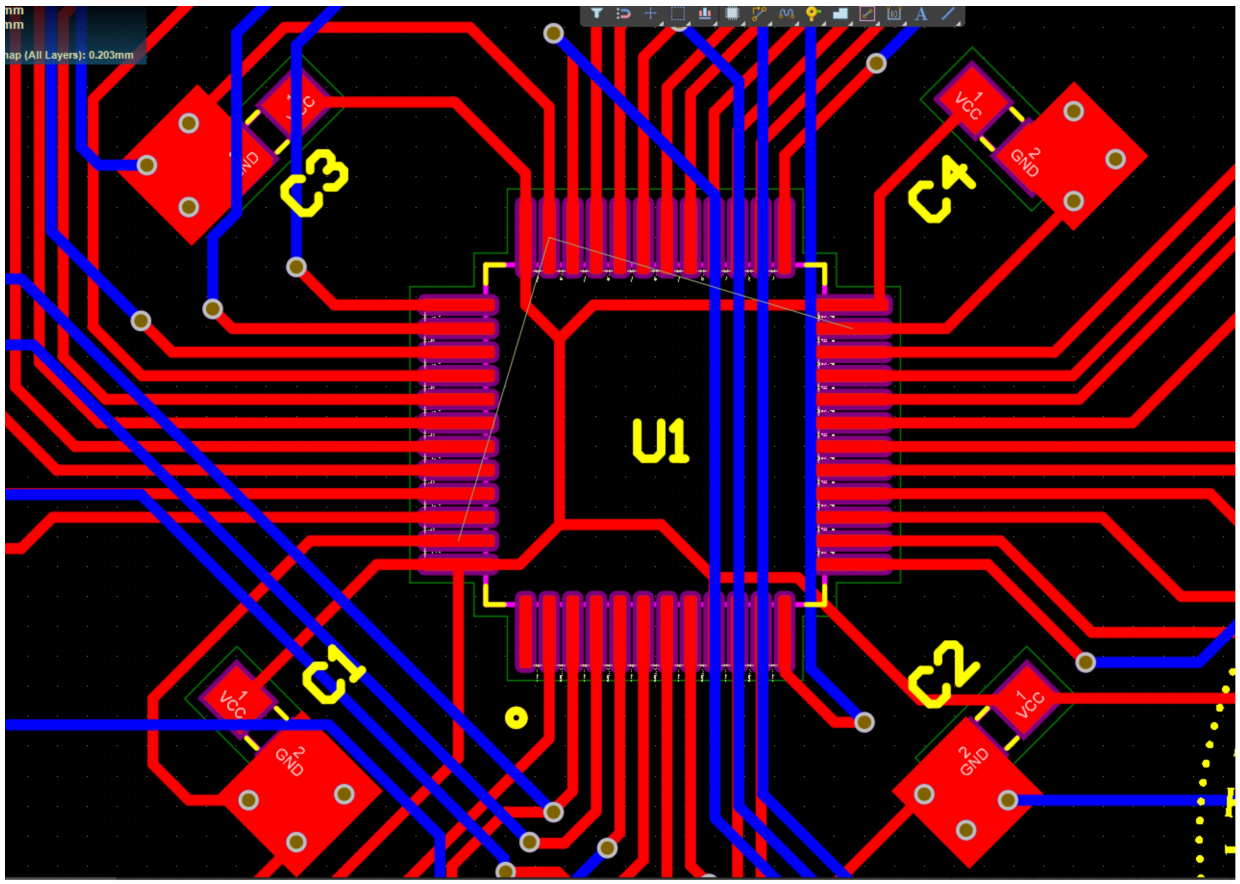


Figura 4.7: Se muestra la colocación de los capacitores C1, C2, C3 y C4 correspondientes a cada *VDD* del microcontrolador.

4.3. Consideraciones para la conexión de componentes y la fabricación del *PCB*

Las reglas de *routeo*, es una lista de configuraciones para la unión de componentes y diseño del *PCB* en *Altium*, donde se debe seguir las tolerancias de fabricación [94] para prototipo de *PCBWay* (empresa fabricante de *PCBs*), encareciendo el costo de manufactura al no considerarse. A continuación, se muestran las reglas:

- El ancho mínimo de pistas es de 0.254 mm.
- Las vías tienen un diámetro de 0.457 mm y un tamaño de agujero de 0.305 mm.
- El espacio entre componentes, vías y pistas (también conocido como *clearance*) es de 0.178 mm.
- El tamaño de las letras usadas para indicar señales como *VCC*, *GND*, +12 V, etcétera, es de 0.762 mm.
- Las pistas no deben tener un ángulo de 90°, esto no se configura, ya que viene por defecto en *Altium*.

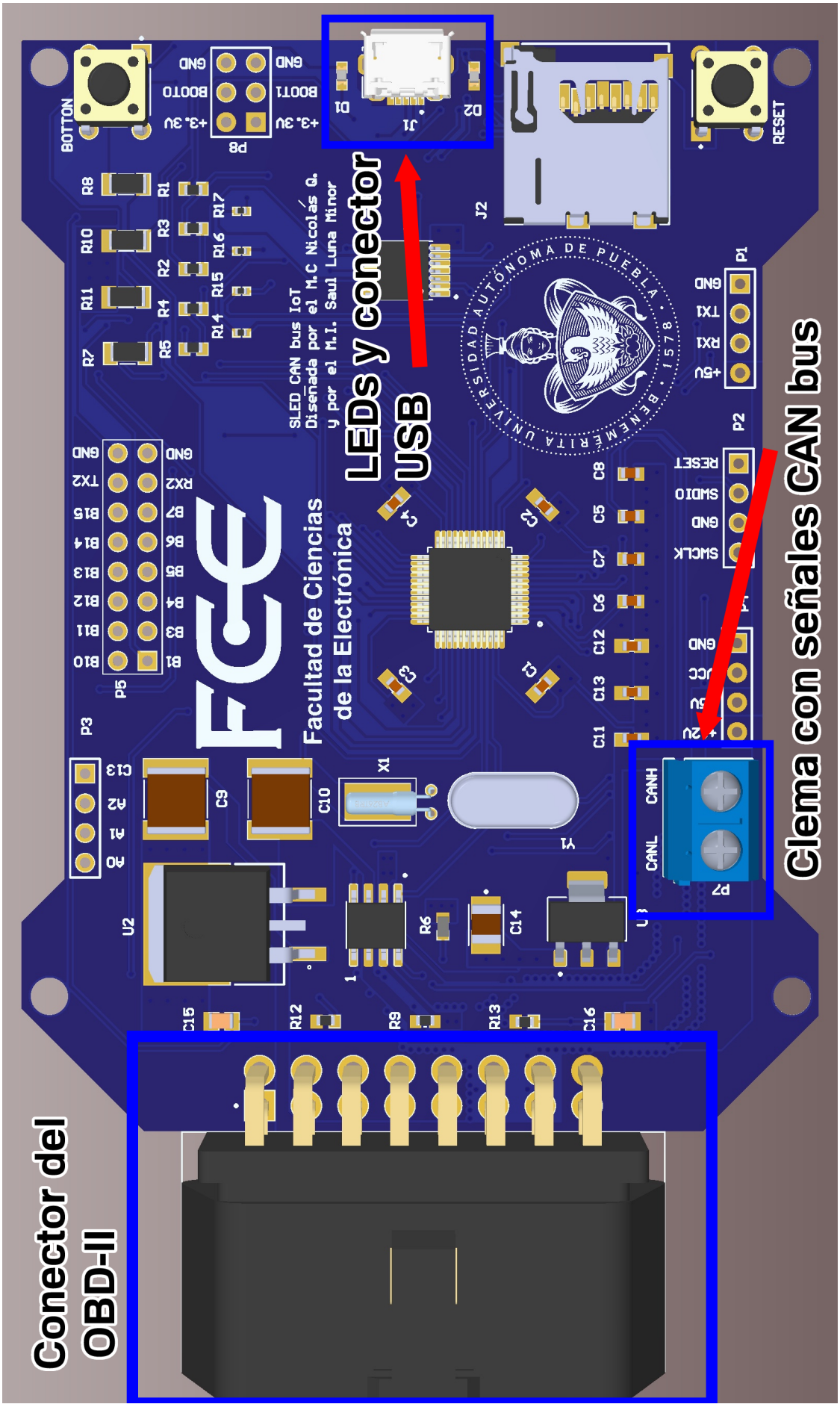


Figura 4.6: Vista frontal en 3D de la tarjeta a desarrollar.

En la figura 4.8, se empieza por unir las señales de 12 V y 5 V, mediante el uso de los polígonos de *Altium* que permiten dibujar figuras como un cuadrado, trapecio, círculo o similar, ampliando el ancho de las líneas de cable definidas (pistas), para así tener una mejor conducción de corriente. *VCC* y *GND* se dejan al final para conectarse por medio de los planos o también conocidos como *power planes*.

Las pistas en color azul son pertenecientes al *BOTTON* (cara inferior del PCB) y las que están en color rojo van al *TOP* (cara superior) como se observa en la figura 4.8. Si bien el ancho de pista usado, es el recomendado por la empresa *PCBWay*, este se puede determinar, mediante la calculadora *online* de *DigiKey*: https://www.digikey.com.mx/es/resources/conversion-calculators/conversion-calculator-pcb-trace-width, donde la variable de interés es la corriente que llevará la pista, ya que arriba de 100 mA, se requiere un ancho mayor al de 0.254 mm (equivalente a 1 milésima de pulgada).

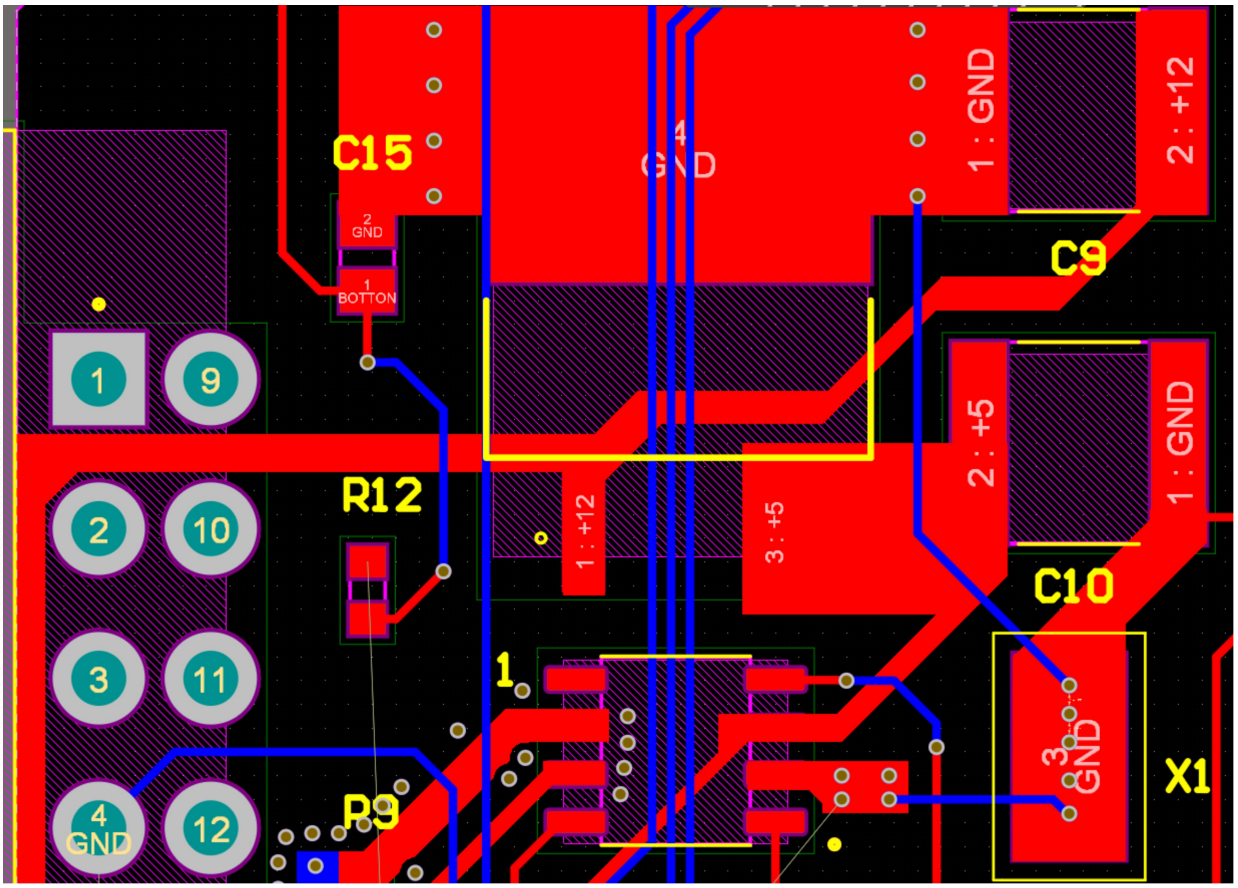


Figura 4.8: Se muestran algunas conexiones de los componentes que van a 12 V, 5 V y *GND* en la cara del *TOP*.

4.3.1. *Via Stitching* para reducir el efecto *EMI* en los cables del *CAN bus*

Una interferencia electromagnética por sus siglas en inglés *EMI* (*Electromagnetic Interference*) es el fenómeno en el que dos o más ondas se fusionan para formar una onda resultante de mayor, menor o igual medida que la de las que la componen, siendo en cualquier caso la onda resultante diferente a las originales y por tanto quizá inadecuadas para su función.

En este caso al tener este dispositivo conectado al vehículo se puede presentar señales parásitas en el *CAN bus*, que afectarían la recolección de datos, por lo que se recurre la técnica de *Via stitching* que se utiliza en los diseños de radiofrecuencia, para reducir el *EMI* y mantener una baja impedancia. Entonces como se observa en la figura 4.9, se tiene literalmente un *muro* creado por vías, con el propósito de aislar la señal de interés de las demás. En este caso se aísla *CANL* y *CANH* de las demás señales

que integran el *PCB*. En *ALTIUM* se tiene una función para agregar *Via stitching* automáticamente a una señal, para que no se realice esto de forma manual.

Cuando se fabrica un *PCB* como prototipo, se tiene un número ilimitado de vías, permitiendo usar el *Via stitching* en todas las señales, sin embargo, esto reduce el espacio donde pasan las pistas, por eso se deben tener en claro cuales señales son propensas al *EMI*. La máscara antisoldante, le da un color a la tarjeta y facilita soldar los componentes, pero es necesario especificar que todas las vías deben llevar esta máscara, ya que por defecto no lo tienen. Además, si se tiene un *logo* o alguna figura, esta se verá afectada sino tienen dicha máscara.

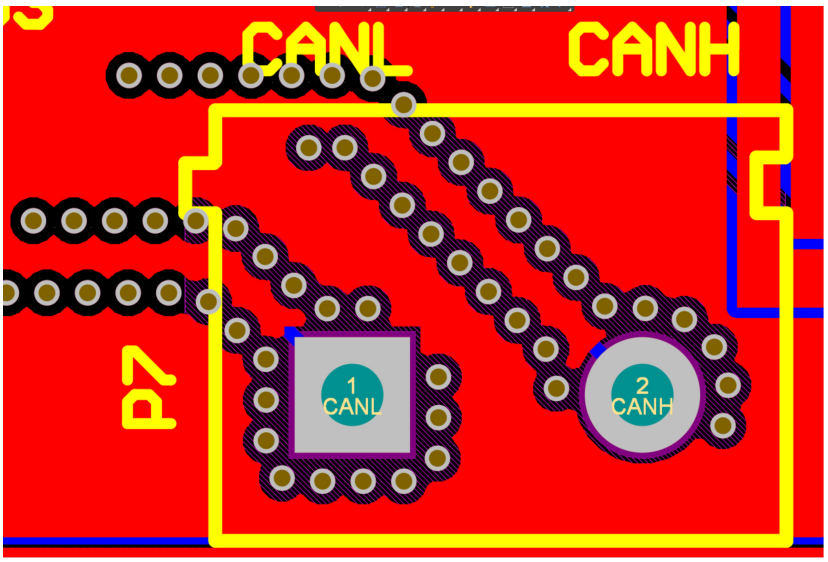


Figura 4.9: Implementación del *Via stitching* en las señales *CANL* y *CANH*.

Conexión a planos de tierra y *VCC*

Con la finalidad de reducir el ruido eléctrico y tener una mejor conducción para VCC y tierra [92] [93], se procede a marcar mediante la herramienta de polígono, toda la superficie del *PCB*, para después seleccionar a donde se va a conectar este polígono y en que capa. Primero se hace para el *TOP*, donde va a ir todo *VCC*, como se muestra en la figura 4.10 y los espacios en negro se tornan en color rojo. Para la parte de *BOTTON*, se hace de manera similar, solo se indica la conexión a *GND* y ahora esta capa toma una tonalidad azul.

Los componentes SMD que están en el *TOP* cuyos pines van a tierra, se les debe crear un polígono que abarque todo el *PAD* correspondiente a esa señal y después se van colocando alrededor vías. Dependiendo del tamaño del polígono resultante, es el número de vías a usar, pero se recomienda un mínimo de 3. Ya por último se colocan los logos y nombres de las señales (*VCC*, *GND*, *SWDIO*, *SWCLK*, etc.) en el *TOP*, estos quedarán en color blanco, a menos que en la fabricación se indique que serán de otro color.

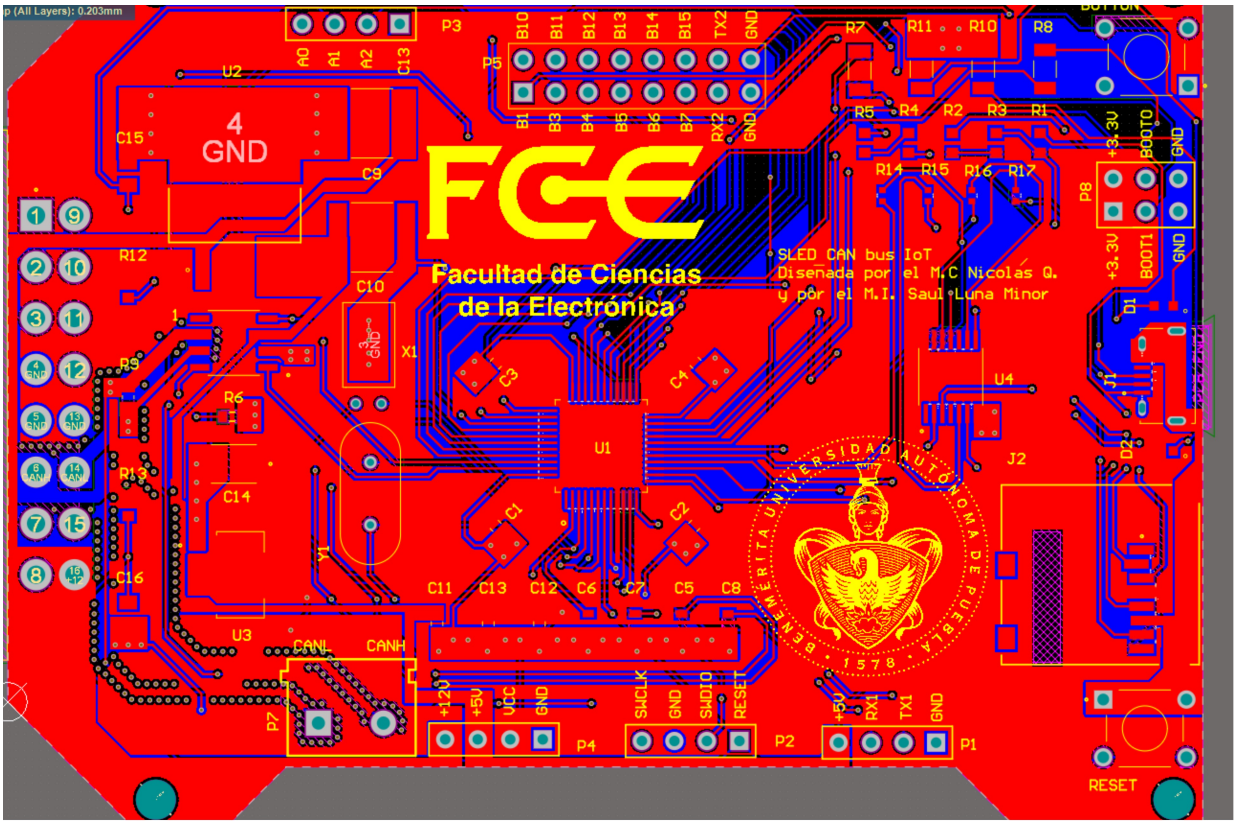


Figura 4.10: Conexión del plano de *VCC* en la cara *TOP*.

4.3.2. Verificación de las reglas de *routeo* mediante *Altium*

El último proceso antes de mandar a fabricar el *PCB*, es verificar que no se tengan cortos circuitos o no se incumpla alguna regla. Esto se puede hacer verificando de manera visual cada conexión realizada, sin embargo lleva un tiempo considerable y se pueden cometer equivocaciones. Entonces se recurre a la opción *Design Rule Checker*, donde se hace una verificación de las reglas antes impuestas y corroborar todas las conexiones. En caso de haber algún error el software lo indica en la placa de circuito impreso, para su corrección. Una vez que no se han encontrado errores, se procede a la fabricación y posteriormente su ensamble.

4.4. Conclusión

La fabricación y ensamble de la tarjeta se ha realizado de manera exitosa, puesto que todos los componentes, han podido ser soldados sin ninguna complicación como se muestra en la figura 4.11. Sin embargo esto no quiere decir que la tarjeta tenga un funcionamiento correcto, se pueden encontrar con componentes defectuosos, o bien alguna conexión no considerada en el desarrollo y por ende provoque un mal funcionamiento, ya sea en un circuito en específico o en toda la tarjeta. Por eso en el siguiente capítulo se realizan dos pruebas, la primera enfocada en establecer comunicación con el *OBD-II* de un vehículo real, si todo sale satisfactoriamente, se procede a la segunda se verifica la congruencia de los datos obtenidos, para esto se hace una comparativa con un escáner comercial.

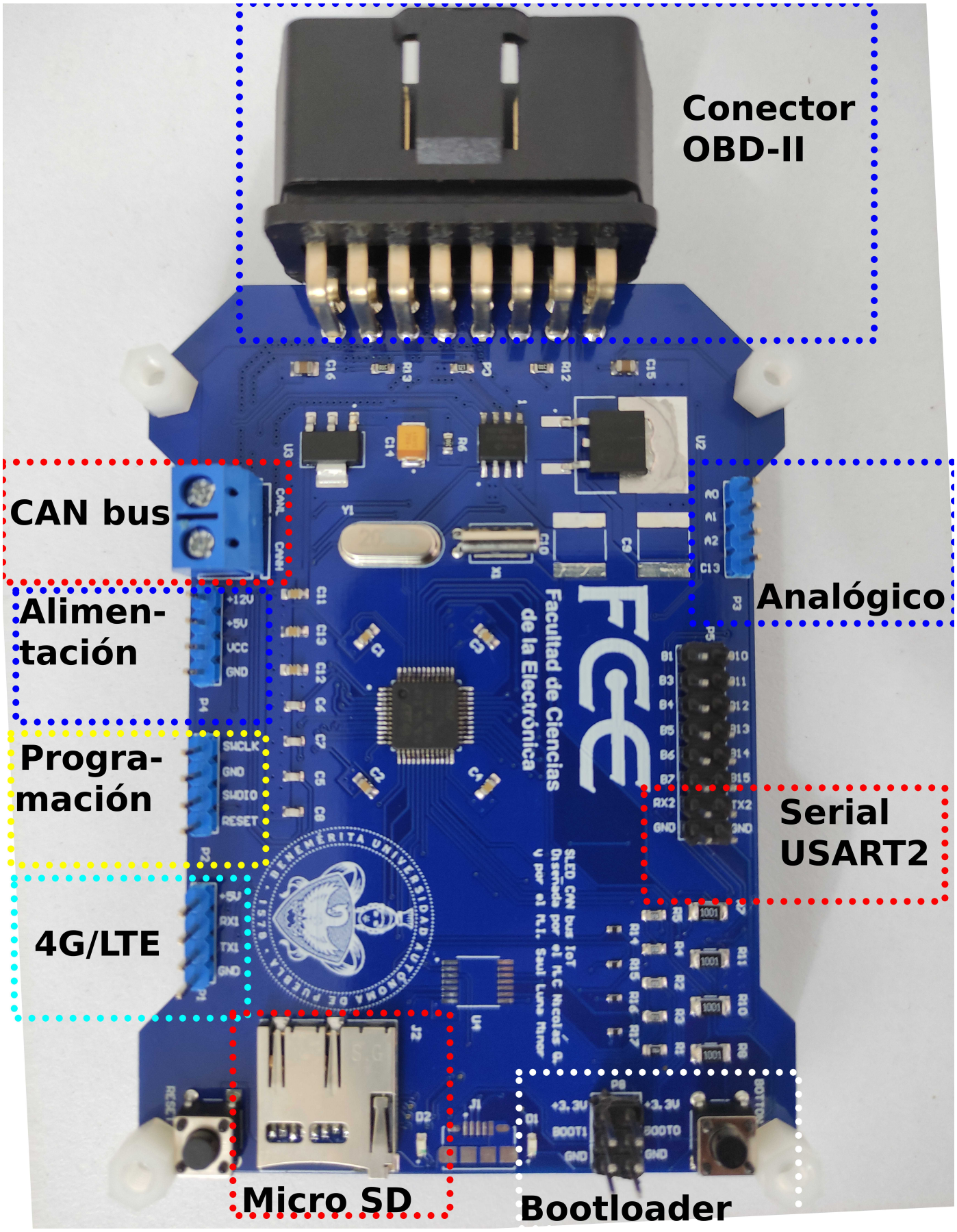


Figura 4.11: Se muestra la tarjeta ensamblada y sus respectivas partes.

Capítulo 5

Resultados

Para validar el *hardware* y *firmware* diseñado en los capítulos anteriores, se realizan dos pruebas, la primera consiste en obtener el valor de mínimo 5 sensores, utilizando el *CAN bus* mediante los códigos del *OBD-II* en un vehículo real. Los datos adquiridos se almacenan en una memoria SD y posteriormente son enviados a una base de datos en la nube, después mediante la página web desarrollada en el capítulo 3, se obtienen/visualizan dichos valores. Si lo anterior funciona correctamente, se procede a la segunda prueba, que tiene como finalidad verificar la congruencia de datos, para esto se compara lo visualizado en el sitio web con lo adquirido por un escáner comercial, para observar las discrepancias que pudieran existir entre ambos. Si el resultado de la comparativa es satisfactorio, entonces se contrastan lo almacenado en la memoria SD con lo representado en la página web, para verificar que el campo de datos de la trama *CAN bus*, con su respectivo identificador y la geolocalización concuerde. Con base a los resultados obtenidos, se pueden dar pie a las conclusiones del proyecto y el trabajo a futuro.

5.1. Prueba en un vehículo real

Para garantizar que el sistema es funcional, se debe realizar una prueba en un automóvil real siguiendo la metodología de la figura 5.1, para ello se debe tener un vehículo con *CAN bus*, esta condición se cumple para modelos 2000 en adelante. A continuación se tiene el siguiente registro de condiciones para la prueba:

- Periodo: Se realizó pruebas el 22, 29 y 30, del mes de Agosto del 2022, con una duración de 15 a 30 minutos.
- Hora: 11:00 AM a 11:30 AM.
- Tipo de vehículo: *Volkswagen Beetle* 2013.
- Condiciones del vehículo: Debido a la pandemia, este tiene dos años sin mantenimiento y requiere un cambio de aceite.
- Lugar: Laboratorio de sistemas automotrices, EMA 7 de la BUAP.
- Condiciones del lugar: Es un espacio cerrado, con una puerta principal que permite el ingreso de vehículos.
- Temperatura ambiente: 22 °C a 25 °C.
- Material de medición: Se utilizó el osciloscopio *MD04104-6*, para observar el protocolo CAN bus. La medición de voltaje y temperatura fue realizada con el multímetro *FLUKE 88*.

Es importante aclarar que las pruebas fueron con el automóvil estacionado, ya que este no tiene placas y por los años que estuvo sin mantenimiento en pandemia, podría fallar al intentar trasladarlo de un lugar a otro. Por otra parte, el laboratorio debe tener sus puertas abiertas, con el fin de que

salga el humo que genera el vehículo, cuando este está en ralentí.

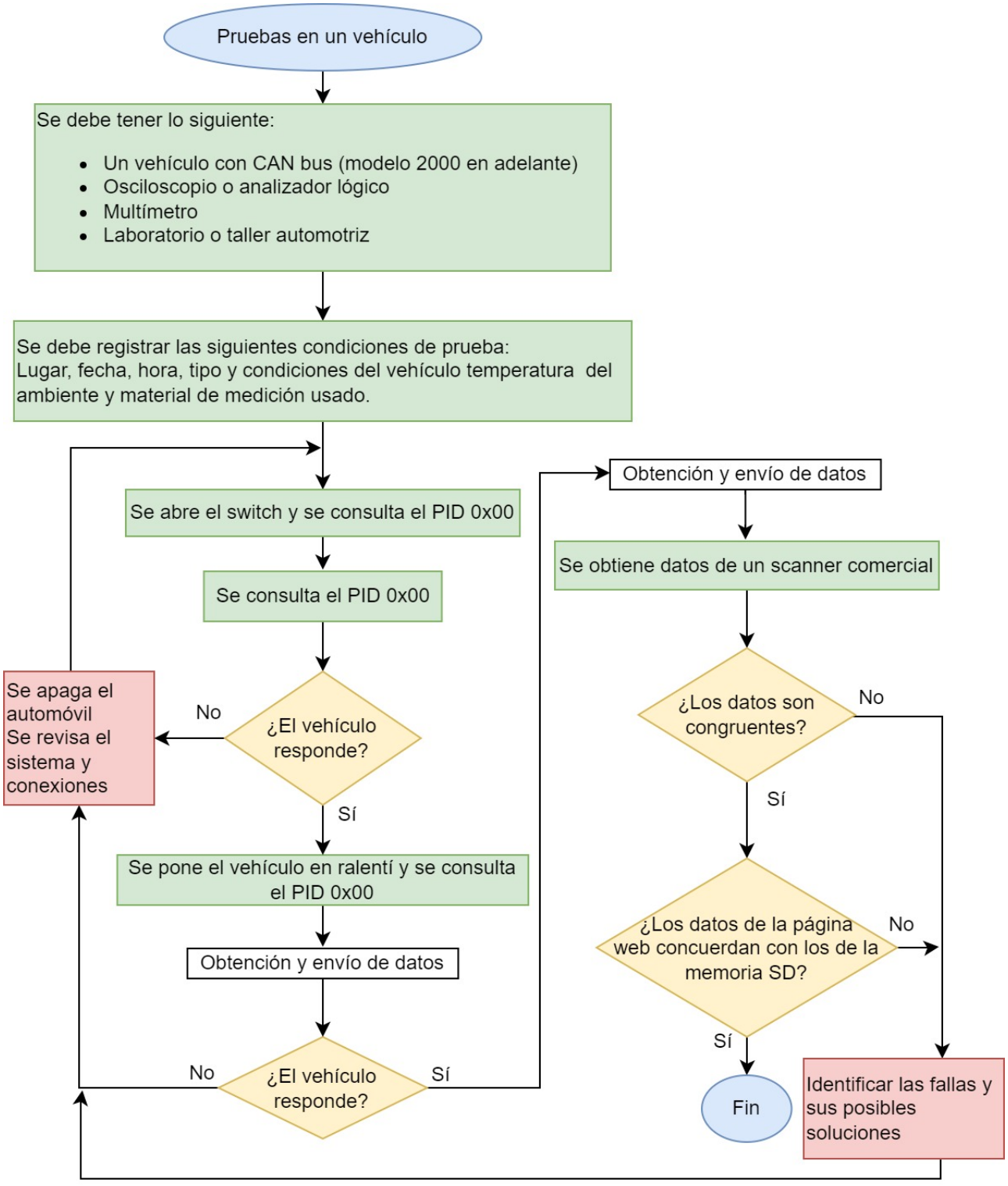


Figura 5.1: Metodología de la prueba realizada en un vehículo real.

Una vez que se ha conectado el sistema al vehículo, se tienen tres estados para hacer pruebas, el primero donde está apagado tanto el motor como el cuadro de instrumentación. Aquí no se obtuvo ninguna respuesta por parte del automóvil. El segundo se enciende el tablero de instrumentación pero no el motor, a este estado también se le conoce como *switch abierto*. Aquí si se tiene una respuesta

del vehículo y por ende se adquieren datos mediante el *OBD-II*. El tercer estado corresponde al de ralentí, donde el motor y el cuadro de instrumentación están encendidos pero el automóvil esta en todo momento en estado estacionario.

Tanto para *switch abierto* y ralentí, se envía el PID 0x00, con el propósito de verificar si el vehículo responde (para esto se tiene un osciloscopio conectado a las líneas del *CAN bus*). En dado caso donde no responda el automóvil, se debe revisar las conexiones y el *firmware* del sistema, sin embargo se dio con éxito la respuesta por parte del *Beetle* a este primer código. Con base en la referencia de los PIDs consultados del *Jetta Hybrid*, se procede a seleccionar estos 5 sensores para realizar la prueba inicialmente: temperatura del líquido de enfriamiento del motor (PID 0x05), RPM (PID 0x0C), velocidad (PID 0x0D), temperatura del líquido de enfriamiento (PID 0x0F) y el MAF (PID 0x10). Aunque es necesario realizar el algoritmo de la figura 5.2, para determinar que PIDs en específico del *VW Beetle* se pueden adquirir, ya que estos pueden variar de un modelo a otro, a pesar de que sean del mismo fabricante. Siendo necesario reprogramar el *firmware* y la página web, según lo que se puedan adquirir en este vehículo en específico.

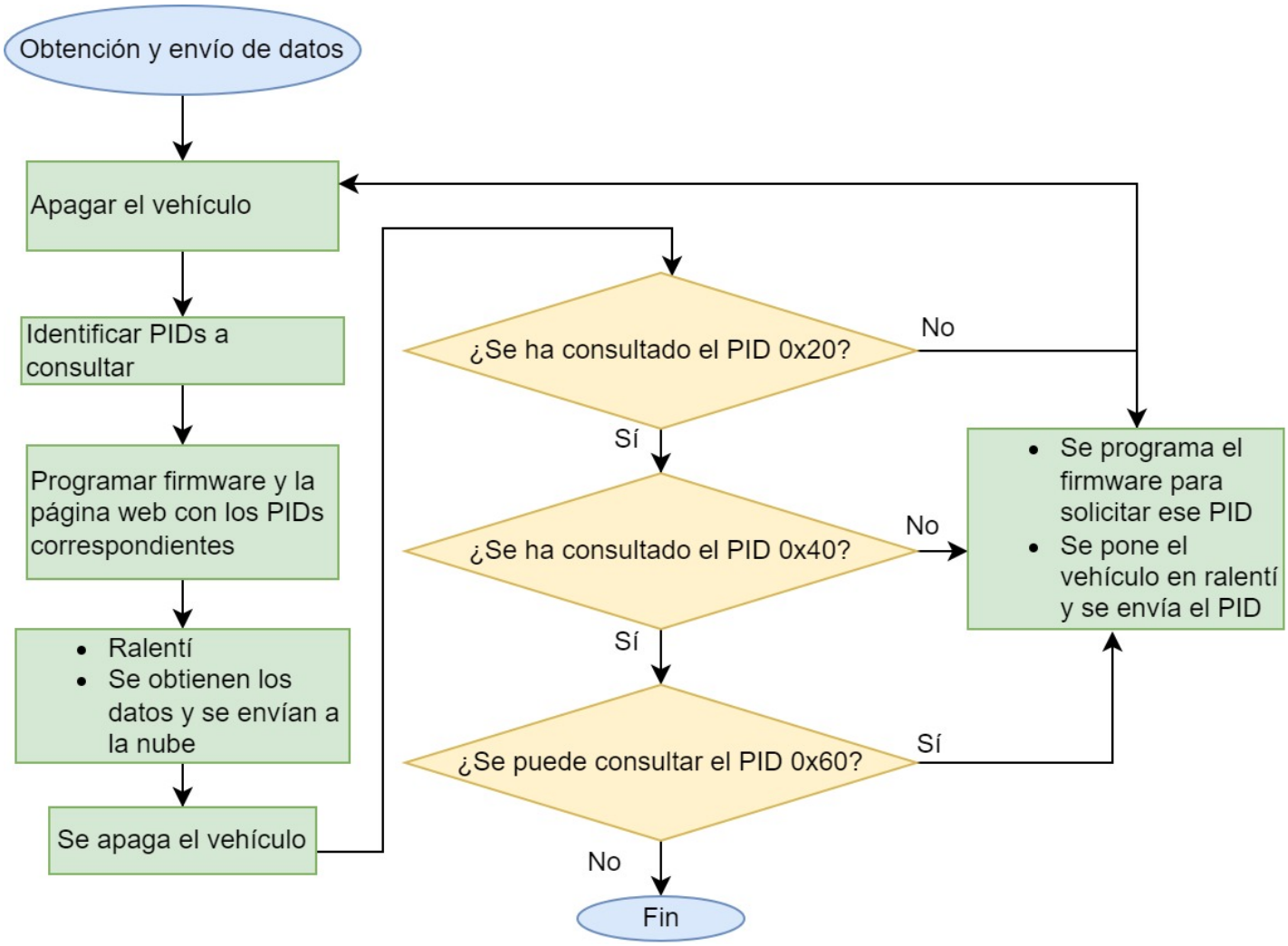


Figura 5.2: Algoritmo de obtención/envío de parámetros en el vehículo.

Entonces el algoritmo de la figura 5.2 tiene el siguiente funcionamiento: una vez mandado el PID 0x00, se debe realizar el proceso mostrado en el capítulo 2, sección 2.2 en el apartado de los códigos PIDs, donde los datos obtenidos son descompuestos en bits como se muestra en la figura 2.12, para así determinar que sensores se pueden consultar desde el PID 0x01 al 0x20. Después se reprograma el

firmware y la página web, aunque antes se debe apagar el vehículo, ya que este proceso lleva algo de tiempo y se debe ahorrar gasolina. Se hace lo mismo para el PID 0x20, que muestra los sensores que se pueden consultar del PID 0x21 al 0x40 y una vez que se ubica cuales son, se programa el *firmware* para adquirir/enviar los datos. Para el caso de este vehículo el PID 0x60 no se puede solicitar. En el apéndice B, se muestran la codificación de los PIDs 0x00, 0x20 y 0x40 en bits, con su respectiva tabla.

Esta prueba se puede resumir como se muestra en la figura 5.3, donde se obtienen datos del *Volkswagen Beetle* mediante el *OBD-II*, estos se convierten en formato *JSON* y se almacenan en una memoria SD, para después ser enviados a la base de datos alojada en la nube. Por último son mostrados en una página web (ver figura 5.4) diseñada en *Vercel*.

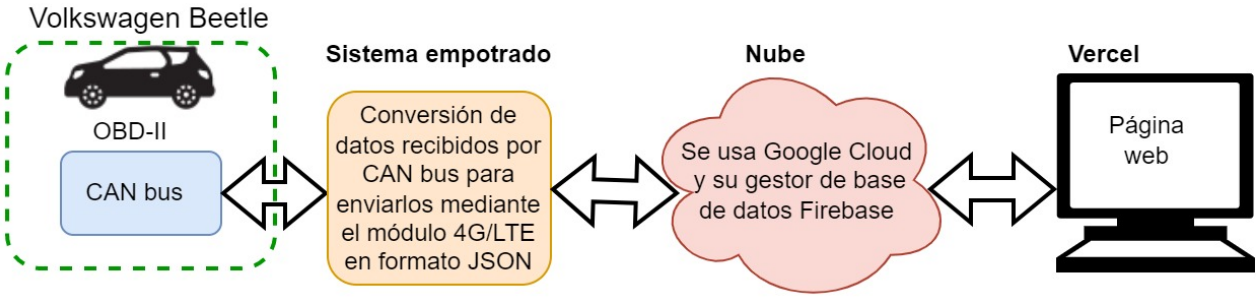


Figura 5.3: Diagrama a bloques de la prueba realizada en un vehículo real.

La programación del *firmware* es mediante la tarjeta *NUCLEO-F446RE*, esta integra el *ST-LINK V2* (se debe revisar el manual para identificar los pines de programación[95]) y un convertidor de Serial a *USB*. Para realizar lo anterior es necesario identificar los *jumpers* que se muestran en la figura 5.5, los cuales al desconectarlos permiten programar microcontroladores externos y una comunicación serial. Si se hace lo anterior correctamente, entonces *KEIL* por defecto reconoce el programador *ST-LINK V2*, una vez que se conecta a la *PC*, permitiendo cargar el *firmware* al microcontrolador.

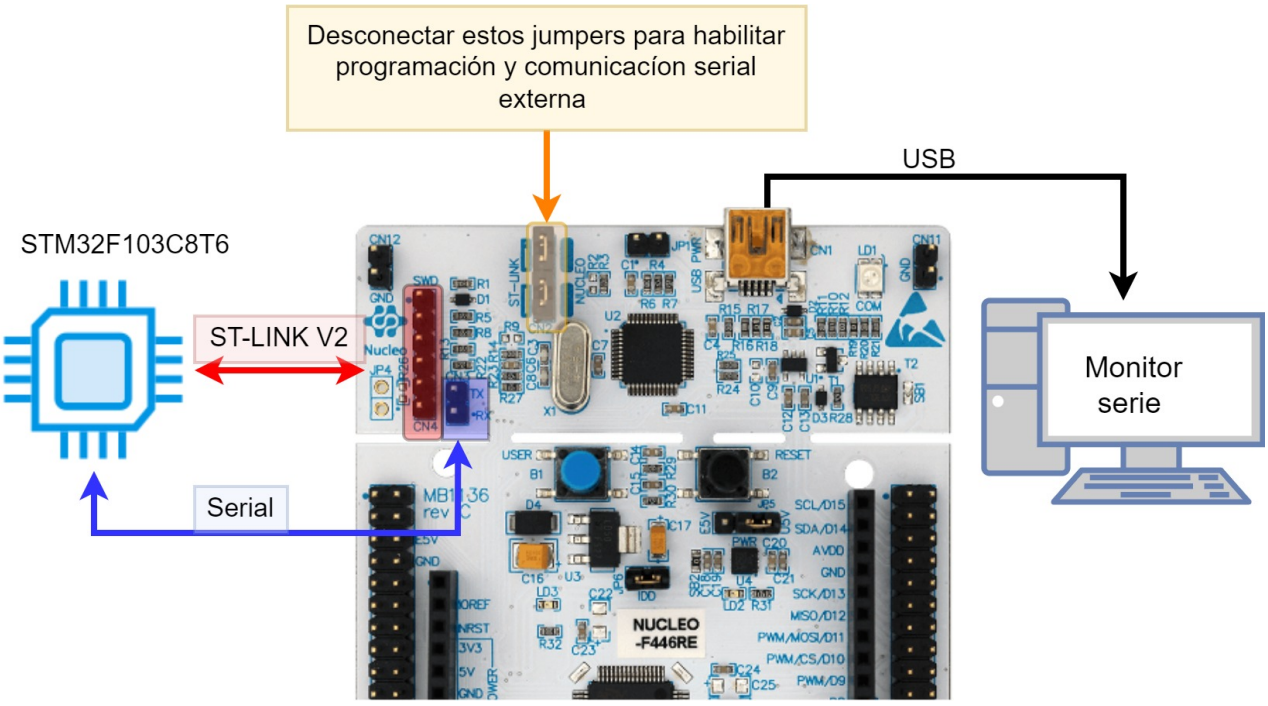
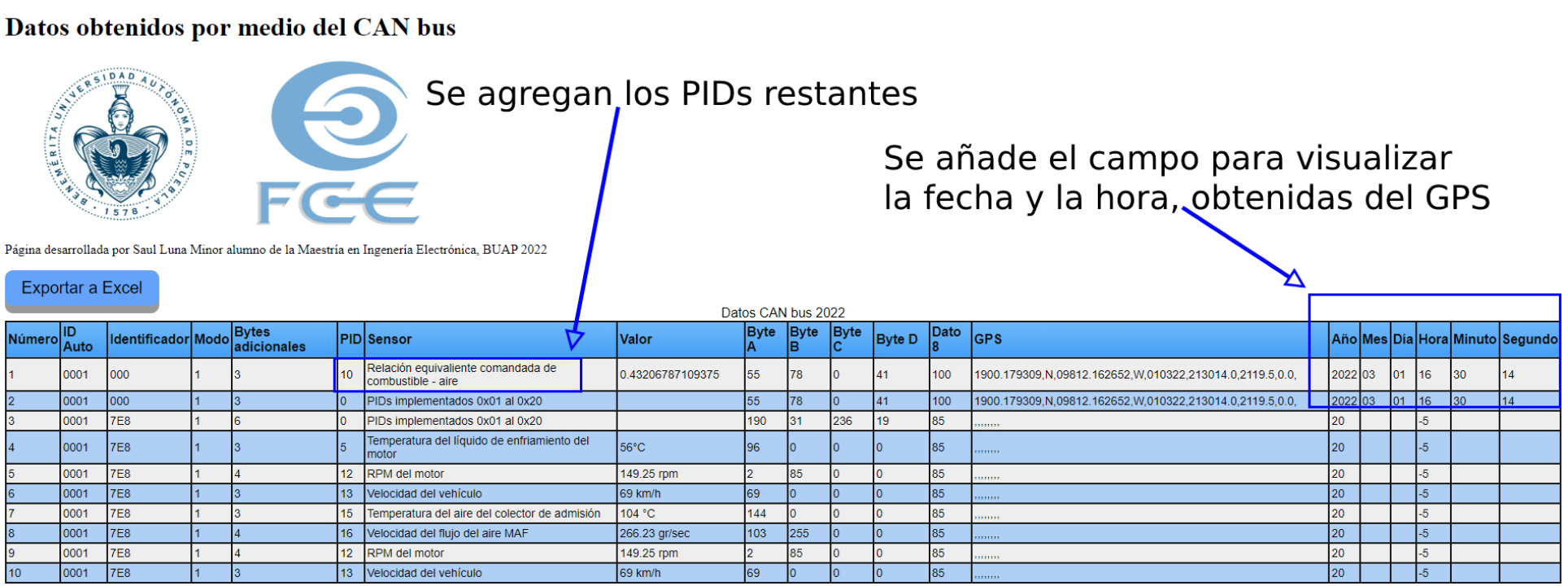


Figura 5.5: Diagrama de la composición del programador *ST-LINK V2* y del convertidor de *Serial* a *USB* de la tarjeta *NUCLEO-F446RE*.



Número	ID Auto	Identificador	Modo	Bytes adicionales	PID	Sensor	Valor	Byte A	Byte B	Byte C	Byte D	Dato 8	GPS	Año	Mes	Día	Hora	Minuto	Segundo
1	0001	000	1	3	10	Relación equivaliente comandada de combustible - aire	0.43206787109375	55	78	0	41	100	1900.179309,N,09812.162652,W,010322,213014.0,2119.5,0.0,	2022	03	01	16	30	14
2	0001	000	1	3	0	PIDs implementados 0x01 al 0x20		55	78	0	41	100	1900.179309,N,09812.162652,W,010322,213014.0,2119.5,0.0,	2022	03	01	16	30	14
3	0001	7E8	1	6	0	PIDs implementados 0x01 al 0x20		190	31	236	19	85	,,,,,,,,	20			-5		
4	0001	7E8	1	3	5	Temperatura del líquido de enfriamiento del motor	56°C	96	0	0	0	85	,,,,,,,,	20			-5		
5	0001	7E8	1	4	12	RPM del motor	149.25 rpm	2	85	0	0	85	,,,,,,,,	20			-5		
6	0001	7E8	1	3	13	Velocidad del vehículo	69 km/h	69	0	0	0	85	,,,,,,,,	20			-5		
7	0001	7E8	1	3	15	Temperatura del aire del colector de admisión	104 °C	144	0	0	0	85	,,,,,,,,	20			-5		
8	0001	7E8	1	4	16	Velocidad del flujo del aire MAF	266.23 gr/sec	103	255	0	0	85	,,,,,,,,	20			-5		
9	0001	7E8	1	4	12	RPM del motor	149.25 rpm	2	85	0	0	85	,,,,,,,,	20			-5		
10	0001	7E8	1	3	13	Velocidad del vehículo	69 km/h	69	0	0	0	85	,,,,,,,,	20			-5		

Figura 5.4: Página web con la fecha y hora, obtenida del *geolocalización*, además de integrar los parámetros que se pueden consultar en el *Volkswagen Beetle*.

Figura 5.6: *Volkswagen Beetle* donde se realizaron las pruebas con el sistema desarrollado.

En la figura 5.6, se observa el *Volkswagen Beetle* y en la figura 5.7, se muestra la placa de circuito impreso diseñada para el sistema de adquisición de datos del *OBD-II* (por su acrónimo SADO), conectada al *OBD-II* de este vehículo, donde se visualiza una resistencia de 120 Ω en las terminales de su clema, la cual sirve para conectar una punta de osciloscopio y ver las señales *CAN bus*. En la figura 5.4 se muestra algunos datos obtenidos por el SADO, con lo cual se finaliza satisfactoriamente esta primera prueba. Para ver todos los valores adquiridos se puede ir al siguiente dirección url de la página web : https://nodejswebserver-steel.vercel.app/

Figura 5.7: Conexión del SADO al *OBD-II* del *Volkswagen Beetle* 2013.

5.1.1. Escáner *AUTEL* para la comparativa de datos

En la figura 5.8 se ve el escáner *AUTEL MaxiFlash Elite* utilizado para comparar los parámetros obtenidos del *Volkswagen Beetle*. Las características de este son:

- Cumple con los estándares *SAE* J2534-1 y *SAE* J2534-2.
- Compatible con aplicaciones de agencia como *Techstream* de Toyota, VIDA de Volvo, HDS de Honda, IDS de *Jaguar-Land Rover* y 3G de *BMW* para el diagnóstico desde la fuente *OEM*.
- Voltaje de entrada: 12 VCC a 24 VCC.
- Corriente de alimentación: 170 mA a 12 VCC 100 mA a 24 VCC.
- Temperatura de funcionamiento: -10 °C a +60 °C.
- Incluye *Bluetooth* y *WIFI*.
- Protocolos: *CAN bus*, *Ford SCP* (J1850PWM), *GM Class2* (J1850VPW), KWP2000 (ISO9141/14230) y *Chrysler SCI* (J2610).

Además tiene una interfaz gráfica, donde el usuario puede consultar los PIDs de un vehículo en específico, siempre y cuando este sea compatible con este escáner. Además de hacer un diagnóstico en tiempo real y detectar las posibles fallas de automóvil, sin embargo esto no se realizó.

Figura 5.8: Escáner *AUTEL MaxiFlash Elite* y su interfaz gráfica.

En la figura 5.9 se muestra la forma en como se conectó este escáner al *OBD-II* del *Volkswagen Beetle*. Cabe mencionar que este escáner a diferencia de la tarjeta diseñada para el SADO, no tiene clemas o algún conector en el cual se pueda conectar una punta de osciloscopio o analizador digital, para así observar la trama *CAN bus*.

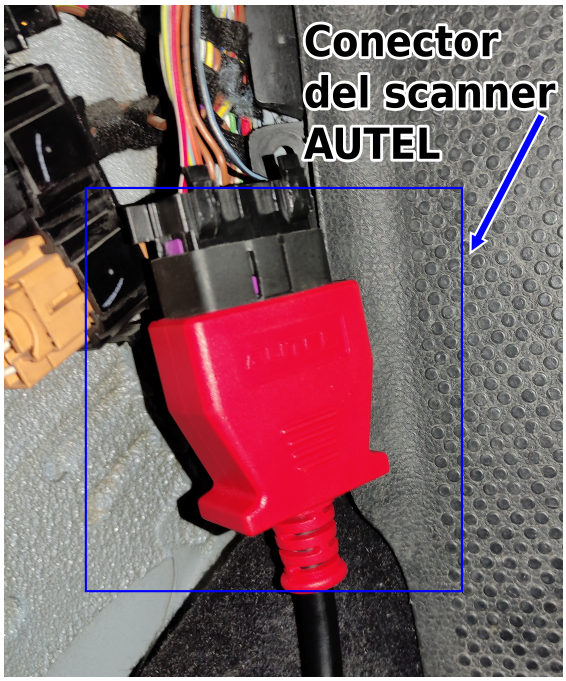


Figura 5.9: Conexión del escáner AUTEL MaxiFlash Elite al OBD-II del Volkswagen Beetle.

Al final este escáner genera un archivo en formato PDF (ver figura 5.10) con los PIDs que puede obtener y al igual que el sistema desarrollado, se debe poner en *switch* abierto o ralentí el automóvil para que se comience con la adquisición de datos. Si bien este escáner tiene la posibilidad de obtener el valor de los sensores continuamente, este no los guarda en una memoria o envía a la nube, perdiendo los datos obtenidos, a menos que se guarden en un archivo PDF.

MaxiSys **INFORME DE DIAGNÓSTICO DE VEHÍCULO** **AUTEL®**

Información sobre el vehículo

Eobd

VIN: 3VW7A7AT3DM801065 **Kilometr.:**

Diagnósticos hora: **Ruta de acceso:** Escaneo Automático > DATOS EN LINEA >

Live Data

NO.	Nombre	Valor	MIN	MAX	Unidad
1	$7E8 DTCs GUARDADOS EN ESTA ECU	1	0	127	
2	$7E9 DTCs GUARDADOS EN ESTA ECU	0	0	127	
3	$7E8 SISTEMA COMBUSTIBLE 1 ESTADO	OL			
4	$7E8 SISTEMA COMBUSTIBLE 2 ESTADO	--			
5	$7E8 VALOR CARGA CACULADA	16.9	0	100	%

Figura 5.10: Fragmento del informe de diagnóstico generado por el escáner *AUTEL MaxiFlash Elite*.

5.2. Resultados

Lo primero que se observa, es el intervalo de tiempo que le toma al *OBD-II* del vehículo en responder, esto se observa en la figura 5.11, siendo de 4.362 ms y tiempos menores a este, no se tendría una respuesta por parte del automóvil.

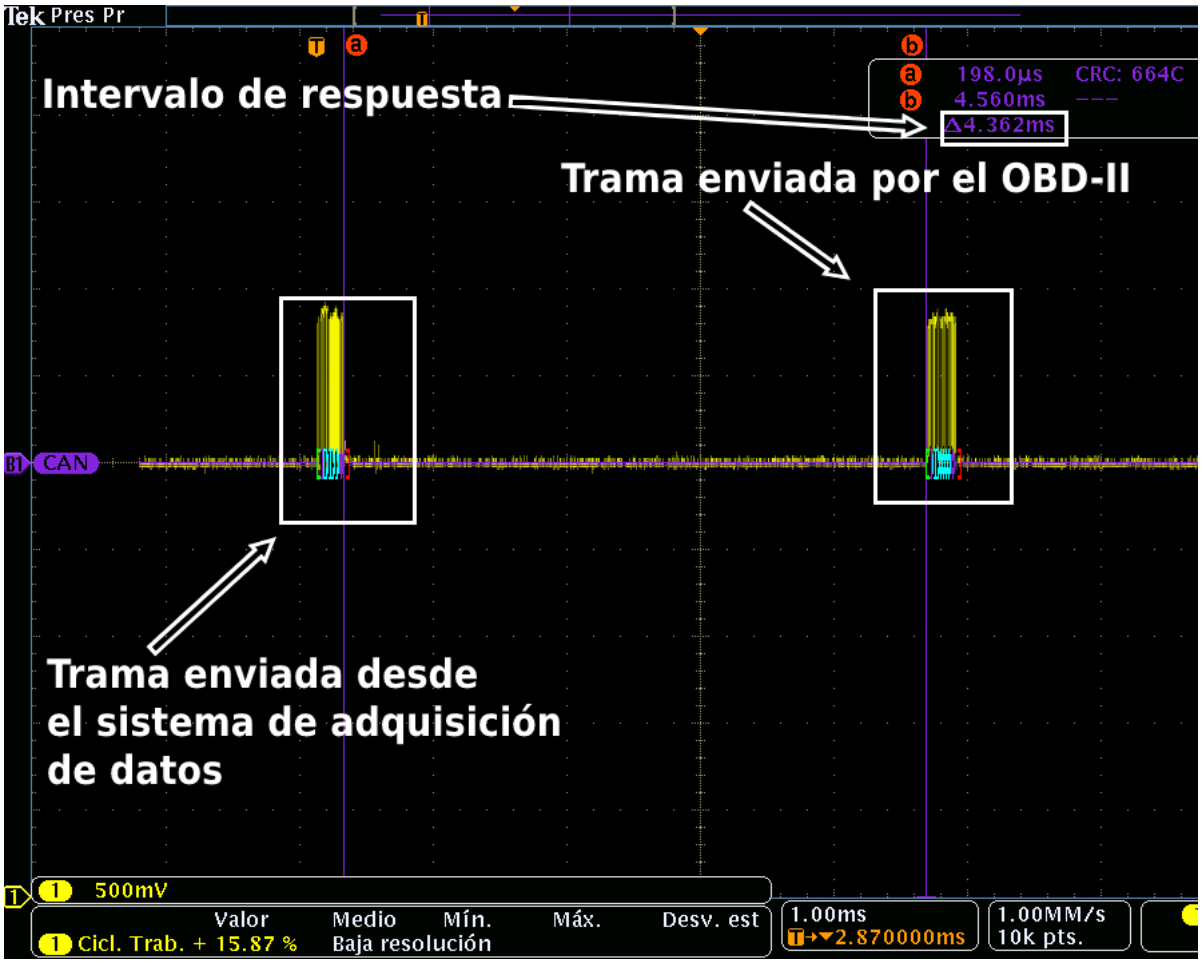


Figura 5.11: Captura tomada desde el osciloscopio, para observar el intervalo de respuesta por parte del *OBD-II*.

5.2.1. Comparativa de los parámetros obtenidos del *OBD-II* entre el sistema y el escáner *AUTEL*

Una vez que se han obtenido/enviado los datos del *OBD-II* del vehículo mediante el SADO, lo que sigue es también adquirir el valor de los sensores utilizando el escáner *AUTEL MaxiFlash Elite*, para así hacer una comparativa entre ambos como se muestra en la figura 5.12, donde se puede apreciar datos con un valor muy cercano entre sí, como por ejemplo la *temperatura refrigerante del motor* o la *RPM*, mientras que en la velocidad, se tiene el mismo valor, ya que el vehículo en todo momento de las pruebas estaba estacionado.

A

Temperatura del líquido de enfriamiento del motor	75°C	115	170
RPM del motor	762 rpm	11	232
Velocidad del vehículo	0 km/h	0	170

B

$7E9 TEMPERATURA REFRIGERANTEE MOTOR	73	0	130	°C
$7E8 AJUSTE CORTO COMBUSTIBLE -FILA 1	0.0	-100	99.22	%
$7E8 AJUSTE LARGO COMBUSTIBLE - FILA 1	-18.8	-100	99.22	%
$7E8 MOTOR RPM	764	0	7000	RPM
$7E9 MOTOR RPM	758	0	7000	RPM
$7E8 SENSOR VELOCIDAD VEHICULO	0	0	240	km/h
$7E9 SENSOR VELOCIDAD VEHICULO	0	0	240	km/h

Figura 5.12: En el lado A se observan los datos de la página web desarrollada y en B los valores adquiridos del escáner *AUTEL MaxiFlash Elite*.

En la tabla 5.1 se muestran cuales PIDs se obtuvieron entre el escáner y el SADO. Mediante la decodificación de la respuesta al solicitar el PID 0x00, es como se sabe que el MAF no se puede adquirir y no se espera obtener un valor en concreto, sin embargo el escáner no solicita el PID 0x00, dando paso a una mala interpretación al desconocer lo anterior, pudiendo intuir que el MAF está fallando. Como no se adquieren datos simultáneamente entre ambos sistemas, algunos valores como la temperatura (PID 0x05 y 0x0F) o las RPM (PID 0x0C), los cuales tienen variaciones durante el ralentí, no deben coincidir exactamente entre los dos sistemas, pudiendo tener ligeras diferencias. Algunos PIDs como *el voltaje del módulo de control* (PID 0x42) o *la distancia recorrida desde que se borraron las fallas* (PID 0x31), son exactamente iguales para ambos, concluyendo así que los datos obtenidos por el SADO, son congruentes, sin embargo para una mayor certeza, se debería realizar una adquisición síncrona entre los dos sistemas. Cabe mencionar que el escáner obtiene más PIDs (en el apéndice A se muestra todo lo adquirido) en comparación al SADO.

Tabla 5.1: Comparativa de los parámetros obtenidos entre el sistema de adquisición de datos en contraste con el escáner *AUTEL*.

PID		**Datos del SADO**	**Datos del scanner AUTEL**
0x00	PIDs implementados [0x01 al 0x20]	152, 24, 128 y 1	
0x05	Temperatura del líquido de enfriamiento del motor	75 °C	73 °C
0x0C	RPM del motor	762 RPM	764 RPM
0x0D	Velocidad del vehículo	0 km/h	0 km/h
0x0F	Temperatura del aire del colector de admisión	27 °C	33 °C
0x10	Velocidad del flujo del aire MAF	0 gr/sec	0 gr/sec
0x11	Posción del acelerador	12.55 %	12.50 %
0x20	PID implementados [0x21 - 0x40]	0, 1, 128 y 1	
0x30	Número de avisos desde que los errores fueron borrados	16	16
0x31	Distancia recorrida desde que se borraron los fallas	2 km	2 km
0x40	PID implementados [0x41 - 0x60]	224, 128, 0 y 0	
0x42	Voltaje del módulo de control	13.6 V	13.6 V
0x43	Valor absoluta de carga	18.82 %	18.80 %
0x44	Relación equivalente comandada de combustible - aire	1	0.999

5.2.2. Intervalos de tiempo y cantidad de bytes utilizados en el envió/almacenamiento de datos

Otro resultado interesante que se puede obtener, es la cantidad de tiempo requerido para obtener, convertir y enviar un dato a la nube, siendo este el mostrado en la tabla 5.2, el cual es de 13.14 segundos. La forma en como se obtuvo este dato, fue mediante el *RTC* que integra el módulo *4G/LTE*, que empieza un conteo en minutos y segundos una vez energizado. Entonces en el minuto 37 con 36

segundos se adquiere el PID 0x00 y cuando obtiene el PID 0x05, por el monitor serial, envía el conteo de tiempo actual, siendo 37 minutos con 48 segundos y así sucesivamente, hasta llegar al último que es la *relación equivalente comandada de combustible- aire* (PID 0x44). Entonces se hace un promedio de los tiempos de adquisición de datos, siendo como resultado 13.1428 y teniendo un intervalo de 2 minutos con 51 segundos para enviar/adquirir todos los parámetros que se muestran en la tabla 5.2.

Tabla 5.2: Intervalos de tiempo para la adquisición/envío de datos del SADO.

PID	**Minuto (min)**	**Segundo (seg)**	**Intervalo de tiempo (seg)**	**Intervalo de tiempo (min:seg)**
PIDs implementados [0x01 al 0x20]	37	36	12	2:52
Temperatura del líquido de enfriamiento del motor	37	48		
RPM del motor	38	3	15	
Velocidad del vehículo	38	18		
Temperatura del aire del colector de admisión	38	32	13	
Velocidad del flujo del aire MAF	38	45		
Posción del acelerador	38	59	14	
PID implementados [0x21 - 0x40]	39	14		
Número de avisos desde que los errores fueron borrados	39	27	13	
Distancia recorrida desde que se borraron los fallas	39	40		
PID implementados [0x41 - 0x60]	39	52	12	
Voltaje del módulo de control	40	2		
Valor absoluta de carga	40	15	13	
Relación equivalente comandada de combustible - aire	40	28		
Media aritmética			13.14285714 segundos	

La finalidad de calcular la media aritmética de la tabla 5.2 es determinar la cantidad de bytes enviados en un año con el SADO y así elegir un plan de *IoT*. Entonces en la tabla 5.3, se muestra el número de bytes enviados por el sistema, esto varía según sea con o sin geolocalización, ya que para lugares cerrados como el laboratorio de automotriz, no se logró obtener respuesta por parte del GPS. Mediante el editor de texto, *Notepad++* se escribe el formato *HTTP* explicado en la sección 3.6.2, determinado 388 bytes utilizados para enviar un mensaje con geoposicionamiento, ocupando 201 bytes para su almacenamiento en la base de datos, ya que este descarta el encabezado y solo guarda los datos del *OBD-II* del vehículo en formato *JSON*. Entonces se tienen registrados hasta el 24 de octubre del 2022 un total de 492 datos almacenados, requiriendo 186 KB para su transmisión y solamente para su alojamiento se requieren 96 KB. Al tener algunos registros de mensajes con y sin geolocalización, se tiene un registro de 94 KB en *Firebase*, siendo entonces bueno considerar 388 bytes para el envió de un mensaje.

Tabla 5.3: Comparativa de la cantidad de datos transmitidos en contraste con los almacenados en la base de datos.

Cantidad de bytes (con geolocalización)		**Cantidad de bytes (sin geolocalización)**		**Cantidad total de bytes por los 492 datos (con geolocalización)**		**Cantidad de bytes registrados en Firebase (con y sin *geolocalización*)**
Enviados por el SADO	**Almacenados en *la base* de datos**	**Enviados por el SADO**	**Almacenados en la base de datos**	**Enviados por el SADO**	**Almacenados en la base de datos**	
388	201	340	152	190896 (186 KB)	98892 (96 KB)	94 KB

Con los resultados de la tabla 5.2 y 5.3, se puede aproximar la cantidad de bytes enviados en un año con la geolocalización, para esto primero se divide la cantidad de segundos en un año (3.154^7 seg.) entre los 13.14 segundos que toma adquirir/enviar un dato. El resultado se multiplica por 388 bytes considerados para enviar un mensaje. El resultado es 0.8 GB necesarios para transmitir datos durante un año por el SADO y si se tienen 500 dispositivos igual a este, los cuales envían datos simultáneamente, se requiere de 4 GB (ver tabla 5.4). Lo anterior es bajo la consideración de un envío constante, pero esto en la realidad no es posible, ya los vehículos no pueden estar trabajando perpetuamente.

Tabla 5.4: Comparativa de las diferentes cantidades de bytes transmitidos por el SADO.

Cantidad de bytes registrados en en la base de datos (con y sin geolocalización)	**Tiempo minimo de adquisición y envío de datos**	**Bytes enviados durante un mes (con geolocalización)**	**Bytes enviados durante un año (con geolocalización)**	
			1 dispositivo	**500 dispositivos**
94.68 KB	13.14 segundos	77600000 (74 MB)	931200000	4 GB

Por último queda determinar el tiempo máximo en que son guardados los datos en la memoria SD antes de llegar a los 4 GB. Para esto se realiza el siguiente cálculo de la fórmula 5.1 , donde se divide los 4 GB entre 201 bytes (ya que estos son los bytes que se almacenan en la memoria SD para el valor de un sensor) y el resultado se multiplica por los 13.14 segundos que toma adquiridlos/enviarlos. Como resultado tenemos 65373134.33 segundos, equivalente a 2 años de almacenamiento.

$$Temp_{Máx} = \frac{4\ GB}{201\ B}(13.14\ s) = 65373134.33\ s = 756\ días = 24\ meses = 2\ años \tag{5.1}$$

Capítulo 6

Conclusiones y trabajo a futuro

A partir de las pruebas realizadas en el capítulo anterior y con base a los resultados obtenidos, se tiene las siguientes conclusiones:

- El sistema es funcional, ya que adquiere los parámetros del *OBD-II* de un vehículo, además de guardarlos en una memoria SD y enviarlos a una base de datos.
- Se obtienen los valores de 11 sensores utilizando *CAN bus* mediante los códigos del *OBD-II*, cumpliendo con el 5° objetivo específico.
- Al hacer la comparativa con el escáner comercial, se puede ver congruencia con los datos obtenidos, sin embargo faltaría hacer una adquisición simultánea entre ambos sistemas para garantizar que el SADO no tiene pérdidas o errores en sus datos.
- No es necesario el uso de un microcontrolador con punto flotante, ya que las operaciones matemáticas las realiza la nube.
- Los datos obtenidos en la tabla 5.3 y 5.4 permiten aproximar la cantidad de datos requeridos para un cierto número de dispositivos en un periodo de tiempo y así determinar que plan de IoT es el más conveniente para el proyecto, sin embargo esto también depende de otros factores como presupuesto, la cobertura, consumo de energía, etc.
- Se genera un base de datos, la cual se puede escalar a más vehículos y con diferentes parámetros, permitiendo probar algoritmos de inteligencia artificial enfocados en el IoV, realizar un diagnóstico en línea, etc.
- Se realizó pruebas en los siguientes dos automóviles: *Volkswagen Jetta* 2021 y *Chevrolet Aveo* 2016, donde no se tuvo una respuesta por parte de ambos, siendo necesario conocer el identificador con el cual se solicitan datos, es por esta razón que se plantea en el 5° objetivo específico, solamente 5 sensores, ya que resulta complicado realizar pruebas al no conocer los identificadores de las empresas automotrices.

6.1. Trabajo a futuro

- Se propone una segunda versión con microcontroladores de empresas asiáticas, por ejemplo *GigaDevice*, esto debido a la escasez de semiconductores que dificulta el desarrollo de sistemas empotrados actualmente. Además de integrar el módulo *4G/LTE*, en el mismo *PCB* y se sugiere el uso del *SIM7000LTE*, ya que este tiene el comando $AT+CLTS = 1$, que sincroniza el *RTC* del módulo con la hora de la red telefónica o de IoT.
- Se debe hacer pruebas en vehículos en movimientos, con la finalidad de probar la cobertura, en distintos planes de IoT y así seleccionar el idóneo.

- Mediante una interfaz de usuario se debe programar los tiempos de adquisición de datos, esto según el plan de *IoT* a usar y el enfoque del proyecto, por ejemplo, para una cierta ruta de transporte público, tendrá un presupuesto asignado, que solo permita contratar un plan de *IoT* económico, donde se envíen datos en intervalos de horas y para rutas transporte privadas, pueden contratar planes que permitan enviar datos en cuestión de minutos o menos.
- Cuando este proyecto se enfoque en ruta de transporte, entidad pública (como los automóviles de los policías), taller automotriz, etcétera, se debe hacer pruebas en diferentes marcas y modelos de vehículo, para tener identificado los tipos de automóviles en donde se puede usar.
- A partir de las pruebas realizadas en diferentes vehículos se puede determinar el tamaño idóneo con su respectiva carcasa para un dispositivo comercial, este debe se debe producir en masa, para disminuir costos y que sea más viable, por ende, se debe conseguir un proveedor con suficientes componentes, como es el caso de los microcontroladores *ARM Cortex-M* de la empresa china *GigaDevice*.
- En la página web diseñada se puede notar que algunos *PIDs* no se llegan a recibir, siendo necesario hacer un análisis a profundidad que permita determinar si este es un problema relevante y cuales serían sus posibles soluciones.
- Investigar sobre algoritmos o técnicas de encriptado con la finalidad de implementar un sistema robusto que no permita ataques cibernéticos o bien asegure la protección de los datos enviados.
- Utilizar *MQTT* y realizar las pruebas necesarias que permitan determinar si este protocolo es óptimo para el sistema desarrollado.
- Se debe implementar un algoritmo que permita detectar cuando el módulo *4G/LTE* no tenga cobertura y por tanto no pueda enviar los datos, siendo necesario que se almacenen en la memoria SD. Después cuando se restablezca conexión con el servicio de *IoT*, se envíen los datos guardados previamente.

Bibliografía

[1] L. Nkenyereye and J. Jang, "Integration of big data for querying can bus data from connected car," in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2017, pp. 946–950.

[2] O. Avatefipour, A. S. Al-Sumaiti, A. M. El-Sherbeeny, E. M. Awwad, M. A. Elmeligy, M. A. Mohamed, and H. Malik, "An intelligent secured framework for cyberattack detection in electric vehicles' can bus using machine learning," *IEEE Access*, vol. 7, pp. 127 580–127 592, 2019.

[3] (2021, Oct.) Modulo Mazda. talleresat. [Online]. Available: https://talleresat.com/producto/modulo-mazda/

[4] *Electrical and electronic systems in the vehicle*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014, pp. 10–69. [Online]. Available: https://doi.org/10.1007/978-3-658-01784-2_1

[5] J. Garcia. (2015, Oct.) CAN - BUS. [Online]. Available: http://jairgarcia26.blogspot.com/2015/09/can-bus.html

[6] R. Fernandez. (2022, Sep.) Características de un sistema Can Bus. PetrolheadGarage. [Online]. Available: https://petrolheadgarage.com/cursos-automocion/caracteristicas-de-un-sistema-can-bus/

[7] X. Lv and T. Xu, "Design of communication node based on can bus," in *2021 IEEE International Conference on Electronic Technology, Communication and Information (ICETCI)*, 2021, pp. 190–193.

[8] P. Richards. (2022, Sep.) A can physical layer discussion. Microchip Technology Inc. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/00228a.pdf

[9] J. M. A. L. O. N. S. O. PEREZ, *Circuitos eléctricos auxiliares del vehículo Rústica*. Ediciones Paraninfo, S.A.

[10] I. G. Miguel. (2015, Jul.) Integración de un sistema para la obtención de datos de vehículos automotores basados en los protocolos CAN bus y OBD-II. Universidad Nacional Autónoma de México. [Online]. Available: http://132.248.52.100:8080/xmlui/handle/132.248.52.100/8008

[11] STMicroelectronics. (2021, Mar.) STM32F446xx advanced Arm®-based 32-bit MCUs. STMicroelectronics. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[12] D. S. Dawoud and P. Dawoud, *6 Serial Peripheral Interface (SPI)*, 2020, pp. 191–244.

[13] ——, *4 RS-232 Implementation, Applications, and Limitations*, 2020, pp. 157–168.

[14] R. Walter and E. Walter, *Chapter 8 Diagnostics (J1939/73)*, 2016, pp. 61–70.

[15] (2021, Nov.) MQTT – The Nerve System of IoT. catchpoint. [Online]. Available: https://www.catchpoint.com/blog/mqtt-iot

[16] V. Electrónica. (2019, Jun.) Introducción al LPWA: NB-IOT CAT-M vs LORA SIGFOX. vencoel. [Online]. Available: https://www.vencoel.com/introduccion-al-lpwa-nb-iot-cat-m-vs-lora-sigfox/

[17] U. Winkelhake, *The digital transformation of the automotive industry : catalysts, roadmap, practice*. Cham, Switzerland: Springer, 2018.

[18] R. Dobbs, J. Manyika, and J. Woetzel, *No Ordinary Disruption*. INGRAM PUBLISHER SERVICES US, 2016. [Online]. Available: https://www.ebook.de/de/product/25684190/richard_dobbs_james_manyika_jonathan_woetzel_no_ordinary_disruption.html

[19] C. Ventura, T. Silva, and A. Duarte, "State of the art of on demand mobility services," in *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, jun 2020.

[20] I. S. BULUT and H. ILHAN, "Cloud based vehicle and traffic information sharing application architecture for industry 4.0 (iot)," in *2019 International Conference on Information and Telecommunication Technologies and Radio Electronics (UkrMiCo)*, 2019, pp. 1–7.

[21] M. Castro. (2013, Aug.) 8 apps con información de tráfico y estado de las carreteras. computerhoy. [Online]. Available: https://computerhoy.com/listas/apps/8-apps-informacion-trafico-estado-carreteras-2940

[22] X. Li and L. Da Xu, "A review of internet of things—resource allocation," *IEEE Internet of Things Journal*, pp. 1–1, 2020.

[23] F. Yang, S. Wang, J. Li, Z. Liu, and Q. Sun, "An overview of internet of vehicles," *China Communications*, vol. 11, no. 10, pp. 1–15, 2014.

[24] G. Signoretti, M. Silva, A. Dias, I. Silva, D. Silva, and P. Ferrari, "Performance evaluation of an edge obd-ii device for industry 4.0," in *2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0 IoT)*, 2019, pp. 432–437.

[25] A. M. Elshaer, M. M. Elrakaiby, and M. E. Harb, "Autonomous car implementation based on can bus protocol for iot applications," in *2018 13th International Conference on Computer Engineering and Systems (ICCES)*, 2018, pp. 275–278.

[26] A. Guiu, *La industria 4.0 en la sociedad digital*. Barcelona: Marge Books, 2019.

[27] L. Bassi, "Industry 4.0: Hope, hype or revolution?" in *2017 IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*, 2017, pp. 1–6.

[28] N. Moustafa and J. Hu, *Security and Privacy in 4G/LTE Network*. Cham: Springer International Publishing, 2020, pp. 1265–1271. [Online]. Available: https://doi.org/10.1007/978-3-319-78262-1_119

[29] Z. Lin, Y. Huang, Y. J. Xie, Q. Wu, X. Huang, J. Li, and X. Liu, "Remote monitoring system of track cleaning vehicles based on 4g-network," in *2020 IEEE International Conference on Networking, Sensing and Control (ICNSC)*, 2020, pp. 1–5.

[30] P. Pyykönen, A. Lumiaho, M. Kutila, J. Scholliers, and G. Kakes, "V2x-supported automated driving in modern 4g networks," in *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2020, pp. 271–275.

[31] R. Vannithamby and S. Talwar, *5G*, 2017, pp. 34–51.

[32] S. A. Nugroho, E. Ariyanto, and A. Rakhmatsyah, "Utilization of onboard diagnostic ii (obd-ii) on four wheel vehicles for car data recorder prototype," in *2018 6th International Conference on Information and Communication Technology (ICoICT)*, 2018, pp. 7–11.

[33] W. Gay, *CAN Bus*. Berkeley, CA: Apress, 2018, pp. 317–331. [Online]. Available: https://doi.org/10.1007/978-1-4842-3624-6_18

[34] A. X. A. Sim and B. Sitohang, "Obd-ii standard car engine diagnostic software development," in *2014 International Conference on Data and Software Engineering (ICODSE)*, 2014, pp. 1–5.

[35] K. L. Oropeza. (2020, Jul.) ARIDRA. 32.5 millones de unidades registradas es el parque vehicular de México durante 2019. Periódico México Automotriz. Granados 29 Col. Lomas de San Mateo C.P. 53200 Naucalpan, Estado de México. [Online]. Available: https://www.mexicoautomotriz.mx/transporte/aridra-32-5-millones-de-unidades-registradas-es-el-parque-vehicular-de-mexico-durante-2019/

[36] Y. Zhou, Y. Zhang, M. Liu, H. Li, and F. Gao, "Design of vehicle remote monitoring system based on 4g and flexray," in *2018 IEEE 18th International Conference on Communication Technology (ICCT)*, 2018, pp. 478–482.

[37] A. BinMasoud and Q. Cheng, "Design of an iot-based vehicle state monitoring system using raspberry pi," in *2019 International Conference on Electrical Engineering Research Practice (ICEERP)*, 2019, pp. 1–6.

[38] A. Srinivasan, "Iot cloud based real time automobile monitoring system," in *2018 3rd IEEE International Conference on Intelligent Transportation Engineering (ICITE)*, 2018, pp. 231–235.

[39] A. Chandrakasan and R. W. Brodersen, *SRAM*, 1998, pp. 352–366.

[40] W. Gay, *Real-Time Clock (RTC)*. Berkeley, CA: Apress, 2018, pp. 175–193. [Online]. Available: https://doi.org/10.1007/978-1-4842-3624-6_10

[41] D. V. Gadre and S. Gupta, *Serial Communication: SPI and I2C*. New Delhi: Springer India, 2018, pp. 211–238. [Online]. Available: https://doi.org/10.1007/978-81-322-3766-2_15

[42] W. Gay, *Introduction*. Berkeley, CA: Apress, 2018, pp. 1–16. [Online]. Available: https://doi.org/10.1007/978-1-4842-3624-6_1

[43] P. Prabhat, B. Labbe, G. Knight, A. Savanth, J. Svedas, M. J. Walker, S. Jeloka, P. M. Fan, F. Garcia-Redondo, T. Achuthan, and J. Myers, "27.2 m0n0: A performance-regulated 0.8-to-38mhz dvfs arm cortex-m33 simd mcu with 10nw sleep power," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 422–424.

[44] W. Gay, *USART*. Berkeley, CA: Apress, 2018, pp. 73–96. [Online]. Available: https://doi.org/10.1007/978-1-4842-3624-6_6

[45] *Introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–5. [Online]. Available: https://doi.org/10.1007/978-3-540-72715-6_1

[46] M. T. Abdelazim, N. Abdelbaki, and A. F. Shosha, *Experimental Digital Forensics of Subscriber Identification Module (SIM) Card*. Cham: Springer International Publishing, 2018, pp. 391–405. [Online]. Available: https://doi.org/10.1007/978-3-319-58424-9_22

[47] M. Unger, G. Fries, T. Steinecke, C. Waghmare, and R. Ramaswamy, "Functional safety test strategy for automotive microcontrollers during electro-magnetic compatibility characterization," in *2019 12th International Workshop on the Electromagnetic Compatibility of Integrated Circuits (EMC Compo)*, 2019, pp. 49–51.

[48] J. S. Potdar and Y. B. Mane, "Hardware design and development of engine control unit for four cylinder engine," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, 2018, pp. 1–5.

[49] L. Nan and G. Sheng, "Research on air conditioning for new energy vehicle based on artificial intelligence," in *2020 International Conference on Computer Engineering and Application (ICCEA)*, 2020, pp. 726–730.

[50] "E/E Diagnostic Test Modes."

[51] D. G. Vrachkov and D. G. Todorov, "Automotive diagnostic trouble code (dtc) handling over the internet," in *2018 IX National Conference with International Participation (ELECTRONICA)*, 2018, pp. 1–3.

[52] T. Denton, *Sistemas electrico y electronico del automovil.* Mexico, D.F: Alfaomega, 2015.

[53] M. Rohith and K. Sreelakshmi, "Design and integration of gateway electronic control unit (ecu) for automotive electronics applications," in *2021 Asian Conference on Innovation in Technology (ASIANCON)*, 2021, pp. 1–4.

[54] P. R. Sawant and Y. B. Mane, "Design and development of on-board diagnostic (obd) device for cars," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, 2018, pp. 1–4.

[55] (2022, Aug.) EL SISTEMA DE REFRIGERACIÓN. CHAID NEME HERMANOS S.A. [Online]. Available: http://www.revistaautopartes.co/en-profundidad/ver/?tx_ttnews[tt_news]=109&cHash=29bd622adeda7f8fb5a94b616a79a7b6#:~:text=Setratadeunsistema,yelfabricantedelmismo.

[56] C. Valladares. (2022, Sep.) Sensor de temperatura de aire de admisión (IATS). PetrolheadGarage. [Online]. Available: https://petrolheadgarage.com/cursos-automocion/sensor-de-temperatura-de-aire-de-admision-iats/

[57] José Luis Gómez. (2020, Nov.) Caudalímetro o sensor MAF: qué es, cómo funciona y cuáles son las averías más frecuentes. [Online]. Available: https://www.diariomotor.com/que-es/mecanica/caudalimetro-averias-sensor-maf/

[58] (2022, Sep.) Control de Voltaje Regulado. PEDRO NOSSOVITCH Y CIA. S.A. [Online]. Available: https://nosso.com/esp/biblioteca_detalle/48#:~:text=Elm\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\def{}\endgroup\relax\let\ignorespaces\relax\accent19o\egroup\spacefactor\accent@spacefactor\futurelet\@let@token\protect\penalty\@M\hskip\z@skipdulodecontrolentraenestemodocuandoel,15.0voltdurante5minutos.

[59] BOSCH, *Manual de la tecnica del automovil.* Reverte.

[60] A. Lisdorf, *Cloud Foundations.* Berkeley, CA: Apress, 2021, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-1-4842-6921-3_1

[61] ——, *The Genealogy of Cloud Computing.* Berkeley, CA: Apress, 2021, pp. 31–45. [Online]. Available: https://doi.org/10.1007/978-1-4842-6921-3_3

[62] M. Bahrami, "Cloud computing for emerging mobile cloud apps," in *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2015, pp. 4–5.

[63] P. G. M. Jorge Villarino Marzo, *La privacidad en el entorno del cloud computing.* Editorial Reus S.A., Sep. 2018. [Online]. Available: https://www.ebook.de/de/product/35411291/jorge_villarino_marzo_pablo_garcia_mexia_la_privacidad_en_el_entorno_del_cloud_computing.html

[64] G. Liu, "Research on independent saas platform," in *2010 2nd IEEE International Conference on Information Management and Engineering*, 2010, pp. 110–113.

[65] B. Liu, X. Chang, Y. Yang, Z. Chen, and Z. Han, "Evaluating performance of active containers on paas fog under batch arrivals: A modeling approach," in *2019 IEEE Symposium on Computers and Communications (ISCC)*, 2019, pp. 1–6.

[66] M. Kozlovszky, M. Törőcsik, T. Schubert, and V. Póserné, "Iaas type cloud infrastructure assessment and monitoring," in *2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2013, pp. 249–252.

[67] A. Lisdorf, *Google*. Berkeley, CA: Apress, 2021, pp. 85–94. [Online]. Available: https://doi.org/10.1007/978-1-4842-6921-3_8

[68] ——, *Amazon*. Berkeley, CA: Apress, 2021, pp. 75–84. [Online]. Available: https://doi.org/10.1007/978-1-4842-6921-3_7

[69] "HiveMQ MQTT broker." John Wiley & Sons, Ltd, aug 2018, pp. 285–294.

[70] J. Min and Y. Lee, "An experimental view on fairness between http/1.1 and http/2," in *2019 International Conference on Information Networking (ICOIN)*, 2019, pp. 399–401.

[71] Y. Chung, J. Y. Ahn, and J. Du Huh, "Experiments of a lpwan tracking(tr) platform based on sigfox test network," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, 2018, pp. 1373–1376.

[72] P. Seneviratne, *Introduction to LoRa and LoRaWAN*. Berkeley, CA: Apress, 2019, pp. 1–22. [Online]. Available: https://doi.org/10.1007/978-1-4842-4357-2_1

[73] J. Yiu, *The definitive guide to ARM Cortex-M3 and Cortex-M4 processors*. Amsterdam: Newnes, 2013.

[74] (2020) FreeRTOS FAQ -Memory Usage, Boot Times Context Switch Times. Amazon Web Services. [Online]. Available: https://www.freertos.org/FAQMem.html#RAMUse

[75] J. Edge. (2017, Jun.) Memory use in CPython and MicroPython. LWN. [Online]. Available: https://lwn.net/Articles/725508/

[76] (2021, Nov.) PIC32MZ1025DAK176-V/2J. Mouser. [Online]. Available: https://www.mouser.mx/ProductDetail/Microchip-Technology/PIC32MZ1025DAK176-V-2J?qs=sPbYRqrBIVmDhTQcROmhPw==

[77] (2021, Nov.) STM32F446RET6. Mouser. [Online]. Available: https://www.mouser.mx/ProductDetail/Microchip-Technology/PIC32MZ1025DAK176-V-2J?qs=sPbYRqrBIVmDhTQcROmhPw==

[78] (2021, Nov.) WiFi Development Tools - 802.11 ESP8266 SMT Module - ESP-12F. Mouser. [Online]. Available: https://www.mouser.mx/ProductDetail/Adafruit/2491?qs=sGAEpiMZZMsKEdP9slC0Yd0U71sLOOuUYBWFq4cMPvk=

[79] Miguel López. (2022, Jul.) Tarjetas microSD, memorias flash USB, discos duros, SSD: mejores ofertas en almacenamiento por el Prime Day 2022. applesfera. [Online]. Available: https://www.applesfera.com/seleccion/tarjetas-microsd-memorias-flash-usb-discos-duros-ssd-mejores-ofertas-almacenamiento-prime-day-2022

[80] X. Chang, F. Hao, J. Wu, and G. Feng, "File recovery of high-order clearing first cluster based on FAT32," in *Cyberspace Safety and Security*. Springer International Publishing, 2019, pp. 467–476.

[81] STMicroelectronics. (2018, Oct.) STM32L41xxx/42xxx/43xxx/44xxx/45xxx/46xxx advanced Arm®-based 32-bit MCUs. STMicroelectronics. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0394-stm32l41xxx42xxx43xxx44xxx45xxx46xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[82] L. G. Feliciano Fuentes, "Identificación y control de parámetros de clúster de instrumentos automotriz mediante Red Can," Nov. 2019.

[83] (2020, Aug.) SIM7500-SIM7600 Series-AT CommandManual. SIMCom Wireless SolutionsLimited. [Online]. Available: https://www.waveshare.net/w/upload/6/68/SIM7500_SIM7600_Series_AT_Command_Manual_V2.00.pdf

[84] Y. Ranjan. (2022, Aug.) Is Firebase an MQTT? Quora, Inc. [Online]. Available: https://qr.ae/pvZ1h1

[85] K. Cheung. (2022, Mar.) firebase/snippets-web. Firebase. San Franciso, CA. [Online]. Available: https://github.com/firebase/snippets-web/blob/7403e77cc6b0c9bae8e9cd9d43f58eb93df2241a/snippets/database-next/read-and-write/rtdb_read_once_get.js

[86] STMicroelectronics. (2021, Feb.) STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced Arm®-based 32-bit MCUs. STMicroelectronics. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0008-stm32f101xx-stm32f102xx-stm32f103xx-stm32f105xx-and-stm32f107xx-advanced-armbased-32bit-mc pdfhttps://www.st.com/resource/en/reference_manual/rm0008-stm32f101xx-stm32f102xx-stm32f103xx-stm3 pdf

[87] S. Components. (2021, Nov.) MC7800, MC7800A, MC7800AE, NCV7800. ON SEMI. [Online]. Available: https://www.onsemi.com/pdf/datasheet/mc7800-d.pdf

[88] (2022, Jun.) AMS1117-3.3V Voltage Regulator. pcboard. [Online]. Available: https://www.pcboard.ca/ams1117-3-volt-regulator

[89] ABC. (2021, Dec.) USB pull-up in STM32F103. electricalengineering. [Online]. Available: https://electronics.stackexchange.com/questions/598925/usb-pull-up-in-stm32f103-d

[90] D. Young. (2019, Jul.) BQ24650: Ringing on switching node ('PH' node) and attempts at mitigation. Texas Instruments. [Online]. Available: https://e2e.ti.com/support/power-management-group/power-management/f/power-management-forum/681960/bq24650-ringing-on-switching-node-ph-node-and-attempts-at-mitigation

[91] NONE. Micro SD Card Module Catalex Schematic. EasyEda. [Online]. Available: https://easyeda.com/modules/Micro-SD-Card-Module-Catalex-Schematic_ddbd2ab5b4e2493fb59b953b63f7e83b

[92] J. Buenrostro. (2018, Apr.) Recomendaciones de diseño del microcontrolador PIC de 8 bits. Blogger. [Online]. Available: http://joelbuenrostroblog.blogspot.com/2018/04/recomendaciones-de-diseno-del.html

[93] Desconocido. (2020, Sep.) Microcontroladores: Condensadores y Alimentación. JMN ELECTRONICS. Alicante (España). [Online]. Available: http://jmnelectronics.com/archives/726

[94] PCBWay. (2020, Sep.) PCB Manufacturing tolerances. PCBWay. [Online]. Available: https://www.pcbway.es/pcb_prototype/PCB_Manufacturing_tolerances.html

[95] STMicroelectronics. (2022, Mar.) STM32 Nucleo-64 development board with STM32F446RE MCU, supports Arduino and ST morpho connectivity. STMicroelectronics. [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-f446re.html

Apéndice A

Informe de diagnóstico generado por el escáner *AUTEL*

El escáner *AUTEL* genera un reporte en formato PDF con los PIDs obtenidos y su valor correspondiente. Algunos parámetros están repetidos como es el caso del valor de carga calculada o las RPM, ya que este escáner solicita los datos con dos identificadores, el primero es 0x7E8 y el segundo 0x7E9. La columna valor indica la medición del PID en ese momento y las demás columnas el rango que tiene con su correspondiente unidad (°C, g/s, km/h, etc.). Para que el lector logre apreciar de mejor manera todo el informe, se divide en tres figura, sin embargo se le aclara que es un único documento.

MaxiSys

INFORME DE DIAGNÓSTICO DE VEHÍCULO

AUTEL®

Información sobre el vehículo

Eobd

VIN: 3VW7A7AT3DM801065 **Kilometr.:**

Diagnósticos hora: **Ruta de acceso:** Escaneo Automático > DATOS EN LINEA >

Live Data

NO.	Nombre	Valor	MIN	MAX	Unidad
1	$7E8 DTCs GUARDADOS EN ESTA ECU	1	0	127	
2	$7E9 DTCs GUARDADOS EN ESTA ECU	0	0	127	
3	$7E8 SISTEMA COMBUSTIBLE 1 ESTADO	OL			
4	$7E8 SISTEMA COMBUSTIBLE 2 ESTADO	--			
5	$7E8 VALOR CARGA CACULADA	16.9	0	100	%
6	$7E9 VALOR CARGA CACULADA	16.9	0	100	%
7	$7E8 TEMPERATURA REFRIGERANTEE MOTOR	73	0	130	°C
8	$7E9 TEMPERATURA REFRIGERANTEE MOTOR	73	0	130	°C
9	$7E8 AJUSTE CORTO COMBUSTIBLE -FILA 1	0.0	-100	99.22	%
10	$7E8 AJUSTE LARGO COMBUSTIBLE - FILA 1	-18.8	-100	99.22	%
11	$7E8 MOTOR RPM	764	0	7000	RPM
12	$7E9 MOTOR RPM	758	0	7000	RPM
13	$7E8 SENSOR VELOCIDAD VEHICULO	0	0	240	km/h
14	$7E9 SENSOR VELOCIDAD VEHICULO	0	0	240	km/h
15	$7E8 AVANCE TIEMPO ENCENDIDO CILINDRO #1	0.0	-64	63.5	°
16	$7E8 AIRE ADMISION TEMPERATURA	33	-40	215	°C
17	$7E8 FLUJO AIRE CONSUMOO DESDE FLUJO MASA AIRE	0.00	0	655.35	g/s
18	$7E8 POSICION MARIPOSA/ACELERADOR ABSOLUTO	12.5	0	100	%
19	$7E9 POSICION MARIPOSA/ACELERADOR ABSOLUTO	12.5	0	100	%

Figura A.1: Primer parte de los PIDs reportados por el escáner.

19	$7E9 POSICION MARIPOSA/ACELERADOR ABSOLUTO	12.5	0	100	%
20	$7E8 ESTADO AIRE SEGUNDOSUNDARIO MANDADO	OFF			
21	$7E8 LOCALIZACION SENSORES O2	B1S123-B2S----			
22	$7E8 SENSORES O2 VOLTAJE SALIDA FILA 1-SENSOR 2	0.455	0	1.27	V
23	$7E8 AJUSTE CORTO COMBUSTIBLE FILA 1-SENSOR 2	99.2	-100	99.22	%
24	$7E8 SENSORES O2 VOLTAJE SALIDA FILA 1-SENSOR 3	0.455	0	1.27	V
25	$7E8 AJUSTE CORTO COMBUSTIBLE FILA 1-SENSOR 3	99.2	-100	99.22	%
26	$7E8 REQUISITOS OBD PARA LOS QUE EL VEHICULO SE HA DISEÑADO	OBD&OBDII			
27	$7E8 TIEMPO DESDE QUE EL MOTOR ARRANCO	162	0	65535	SEGUNDOS
28	$7E8 DISTANCIA RECORRIDA CON LUZ MIL ENCENDIDA	3	0	65535	Km

Figura A.2: Segunda parte de los PIDs reportados por el escáner.

MaxiSys

INFORME DE DIAGNÓSTICO DE VEHÍCULO

AUTEL®

29	$7E8 PRESION RAIL COMBUSTIBLE	3940	0	655350	kPa
30	$7E8 MANDADO PURGA EVAPORACIONES	0.0	0	100	%
31	$7E8 NUMERO AVISOS DESDE QUE LOS ERRORES FUERON BORRADOS	8	0	255	
32	$7E9 NUMERO AVISOS DESDE QUE LOS ERRORES FUERON BORRADOS	16	0	255	
33	$7E8 DISTANCIA DESDE QUE LOS ERRORES FUERON BORRADOS	3	0	65535	Km
34	$7E9 DISTANCIA DESDE QUE LOS ERRORES FUERON BORRADOS	2	0	65535	Km
35	$7E8 PRESION BAROMETRICA	78	0	255	kPa
36	$7E8 EQUIVALENCIA RATIO (LAMBDA)(FILA 1-SENSOR 1)	0.997	0	2	
37	$7E8 SENSORES O2 CORRIENTE(FILA 1-SENSOR 1)	-0.023	-128	128	mA
38	$7E8 TEMPERATURA CATALIZADOR FILA 1,SENSOR 1	301.1	-40	6513.5	°C
39	$7E8 MODULO CONTROL VOLTAJE	13.923	0	20	V
40	$7E9 MODULO CONTROL VOLTAJE	13.600	0	20	V
41	$7E8 VALOR CARGA ABSOLUTO	21.6	0	25700	%
42	$7E9 VALOR CARGA ABSOLUTO	18.8	0	25700	%
43	$7E8 MANDADO EQUIVALENCIA RATIO	0.999	0	2	
44	$7E8 POSICION RELATIVO/A MARIPOSA/ACELERADOR	2.4	0	100	%
45	$7E8 TEMPERATURA AMBIENTE AIRE	22	-40	215	°C
46	$7E8 POSICION MARIPOSA/ACELERADOR ABSOLUTO B	12.5	0	100	%
47	$7E8 POSICION PEDAL ACELERADOR D	14.9	0	100	%
48	$7E9 POSICION PEDAL ACELERADOR D	14.9	0	100	%
49	$7E8 POSICION PEDAL ACELERADOR E	14.9	0	100	%
50	$7E8 MANDADO MARIPOSA/ACELERADOR CONTROL ACTUADOR	3.1	0	100	%
51	$7E8 TIPO COMBUSTIBLE ACTUAL UTILIZADO POR VEHICULO	GAS			
52	$7E8 AJUSTE LARGO COMBUSTIBLE SENSOR O2 SECUNDARIO - FILA 1	0.0	-100	99.22	%

Figura A.3: Tercera parte de los PIDs reportados por el escáner.

Apéndice B

Tabla de PIDs que se pueden consultar en el *Volkswagen Beetle.*

A continuación se muestran los PID que se pueden consultar mediante el sistema desarrollado. Como se ven en la figura D.1, al consultar el PID 0x00, se tiene una respuesta de 4 bytes, los cuales se les asigna la letra A, B, C y D, esto con el fin de poder descomponerlo en bits y darles la asignación de la tabla D.1. Con esto se puede emparejar el bit MSB asignado con A0 al PID 0x01 y si este tiene valor 1, entonces se puede consultar, caso contrario no se solicita. La tabla D.2 se muestran los PID que se pueden consultar en el vehículo mencionado.

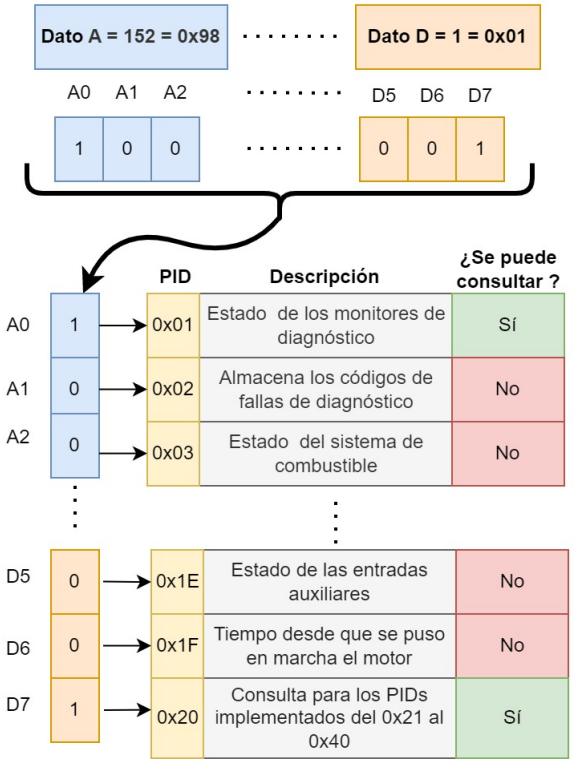


Figura B.1: Mediantes los bytes A,B,C y D se determinan cuales PIDS se pueden consultar .

MSB	PIDs implementados [01 - 20]						LSB
152 = 0x98							
1	0	0	1	1	0	0	0
A0	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**
24 = 0x18							
0	0	0	1	1	0	0	0
B0	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**
128 = 0x80							
1	0	0	0	0	0	0	0
C0	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
1 = 0x01							
0	0	0	0	0	0	0	1
D0	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**
PIDs implementados [21 - 40]							
0 = 0x00							
0	0	0	0	0	0	0	0
E0	**E1**	**E2**	**E3**	**E4**	**E5**	**E6**	**E7**
1 = 0x01							
0	0	0	0	0	0	0	1
F0	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**
128 = 0x80							
1	0	0	0	0	0	0	0
G0	**G1**	**G2**	**G3**	**G4**	**G5**	**G6**	**G7**
1 = 0x01							
0	0	0	0	0	0	0	1
H0	**H1**	**H2**	**H3**	**H4**	**H5**	**H6**	**H7**
PIDs implementados [41 - 60]							
224 = 0xE0							
1	1	1	0	0	0	0	0
I0	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**
128 = 0x80							
1	0	0	0	0	0	0	1
J0	**J1**	**J2**	**J3**	**J4**	**J5**	**J6**	**J7**
0 = 0x00							
0	0	0	0	0	0	0	0
K0	**K1**	**K2**	**K3**	**K4**	**K5**	**K6**	**K7**
0 = 0x00							
0	0	0	0	0	0	0	0
L0	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**

Tabla B.1: Al consultar los PIDs 0x00, 0x20 y 0x40, se tiene 4 bytes de respuesta, donde se les asigna a cada bit una codificación correspondiente.

	PID (Hex)	Descripción	¿Se puede consultar?
Datos en binario -2 de respuesta al consultar el PID 0	0x00	PIDs implementados [01 - 20]	Sí
A0=1	0x01	Estado de los monitores de diagnóstico.	Sí
A1 = 0	0x02	Almacena los códigos de fallas de diagnóstico DTC de un evento	No
A2 = 0	0x03	Estado del sistema de combustible	No
A3 = 1	0x04	Carga calculada del motor	Si
A4 = 1	0x05	Temperatura del líquido de enfriamiento del motor	Si
. . .			
B3 = 1	0C	RPM del motor	Si
B4 = 1	0D	Velocidad del vehículo	Si
C0 = 1	0x11	Posición del acelerador	Si
D7 = 1	0x20	PID implementados [21 - 40]	Si
F7 = 1	30	Cantidad de calentamientos desde que se borraron los fallas	Si
G0 = 1	31	Distancia recorrida desde que se borraron los fallas	Si
H7 = 1	40	PID implementados [41 - 60]	Si
I0 = 1	41	Estado de los monitores en este ciclo de manejo	Si
I1 = 1	42	Voltaje del módulo de control	Si
I2 = 1	43	Valor absoluta de carga	Si
I3 = 0	44	Relación equivaliente comandada de combustible - aire	No
J0 = 1	49	Posición del pedal acelerador D	Si

Tabla B.2: A partir de la tabla D.1 se pueden identificar que parámetros se pueden consultar del *Volkswagen Beetle*.