Universidad de Málaga

Escuela de Ingenierías Industriales

Departamento de Ingeniería de Sistemas y Automática

Trabajo Fin de Grado

Sistema de telemetría BusCAN de vehículo conectado empleando tecnologías de IoT

Grado en Ingeniería Electrónica Industrial

Autor: Carlos Domingo González García

Tutor: Ricardo Vázquez Martín
Cotutor: David Padial Allue

26 de mayo de 2025

Resumen

Sistema de telemetría BusCAN de vehículo conectado empleando tecnologías de IoT

Autor: Carlos Domingo González García
Tutor: Ricardo Vázquez Martín
Cotutor: David Padial Allue
Departamento: Departamento de Ingeniería de Sistemas y Automática
Titulación: Grado en Ingeniería Electrónica Industrial
Palabras clave Telemetría, Vehículo conectado, Bus CAN, OBD-II, IoT, MQTT, 5G, Node-Red, Python, Broker Mosquitto, Nissan Leaf, Linux.

El presente Trabajo Fin de Grado tiene como objetivo el desarrollo e implementación de un sistema de telemetría para vehículos, que permita la adquisición y transmisión de datos mediante tecnologías IoT haciendo uso del bus CAN (Controller Area Network) presente en el vehículo. El sistema se basa en la lectura de tramas a través del conector OBD-II, haciendo uso de un adaptador OBD-II a DB9. A partir de este conector, se extraen las señales diferenciales CANL y CANH, las cuales se cablean a un adaptador USB-CAN conectado a un equipo host. En este host se ejecuta un script encargado de la adquisición de datos y publicación vía MQTT, utilizando conectividad 5G hacia una interfaz gráfica desarrollada en Node-Red que muestrea en tiempo real señales relevantes del vehículo, con el propósito de habilitar un vehículo conectado con capacidad de monitorización remota.
Para llevarlo a cabo, se ha empleado un Nissan Leaf AZE0 del año 2014 y se ha habilitado un entorno Linux en el que se ha desarrollado un script en Python capaz de interpretar, decodificar y publicar las tramas CAN mediante protocolo MQTT. La red 5G del vehículo ha sido configurada para alojar un broker Mosquitto, permitiendo la visualización remota de las señales mediante la interfaz Node-Red.

Abstract

CANBus Telemetry System for a Connected Vehicle Using IoT technologies

Author: Carlos Domingo González García
Supervisor: Ricardo Vázquez Martín
Cosupervisor: David Padial Allue
Departamemt: Departamento de Ingeniería de Sistemas y Automática
Degree: Grado en Ingeniería Electrónica Industrial
Keywords: Telemetry, Connected vehicle, CAN bus, OBD-II, IoT, MQTT, 5G, Node-Red, Python, Mosquitto broker, Nissan Leaf, Linux.

This Bachelor's Thesis aims to develop and implement a vehicle telemetry system capable of acquiring and transmitting data using IoT technologies through the in-vehicle CAN bus (Controller Area Network). The system is based on reading CAN frames via the OBD-II connector, using an OBD-II to DB9 adapter. From this point, the differential CANL and CANH signals are extracted and wired to a USB-CAN adapter connected to a host device. On this host, a script is executed to acquire and publish the data via MQTT, using a 5G connection to a graphical interface developed in Node-Red. This interface displays relevant vehicle signals in real time, with the goal of enabling a connected vehicle with remote monitoring capabilities.
To carry out the project, a 2014 Nissan Leaf AZE0 was used, and a Linux environment was set up where a Python script was developed to interpret, decode, and publish CAN frames using the MQTT protocol. The vehicle's 5G network was configured to host a Mosquitto broker, allowing remote visualization of the signals through the Node-Red interface.

A mi madre, por su amor incansable

A mi padre, in memoriam

Agradecimientos

Quisiera comenzar agradeciendo a mi tutor Ricardo Vázquez Martín y a mi cotutor David Padial Allue por permitirme participar en su investigación y proporcionarme orientación y apoyo para el desarrollo de este Trabajo Fin de Grado.

A mi madre, Salud Irene García Ruíz, un ejemplo a seguir como persona y como madre, por su apoyo incondicional y constante durante toda mi carrera universitaria y por proporcionarme esa fuerza indescriptible con sus palabras para seguir peleando por mis sueños. Por estar siempre a mi lado creyendo en mí, en los mejores y peores momentos, gracias.

A todos los amigos que hice durante este camino y a los que ya estaban, ya que de una forma u otra, han aportado su granito de arena a lo largo de este proceso. Gracias por estar ahí en los momentos clave y por convertir esta etapa en algo mucho más llevadero.

Acrónimos y Notación Matemática

IoT	Internet of Things (Internet de las Cosas)
CAN	Controller Area Network (Red de área de control)
OBD-II	On-Board Diagnostics II (Diagnóstico a bordo)
ECU	Electronic Control Unit (Unidad de control electrónico)
DAQ	Data Acquisition (Adquisición de datos)
VCM	Vehicle Control Module (Módulo de Control del Vehículo)
CAR-CAN	Bus CAN principal del vehículo
EV-CAN	Bus CAN dedicado a la propulsión eléctrica
AV-CAN	Bus CAN para infoentretenimiento
MQTT	Message Queuing Telemetry Transport (Protocolo de telemetría ligero)
5G	Quinta generación de redes móviles
UDP	User Datagram Protocol (Protocolo de datagramas de usuario)
CRC	Cyclic Redundancy Check (Verificación por redundancia cíclica)
PID	Parameter ID (Identificador de parámetro)
API	Application Programming Interface (Interfaz de programación de aplicaciones)
BMS	Battery Management System (Sistema de gestión de baterías)
ADAS	Advanced Driver Assistance Systems (Sistemas avanzados de asistencia al conductor)
GUI	Graphical User Interface (Interfaz gráfica de usuario)
TLS	Transport Layer Security (Seguridad de la capa de transporte)
QoS	Quality of Service (Calidad de servicio)
DB9	Tipo de conector serie estándar
Node-RED	Entorno de desarrollo visual para IoT
Mosquitto	Broker MQTT de código abierto
SLCAN	Serial Line CAN (Protocolo para comunicación CAN por puerto serie)
STM32	Microcontrolador de 32 bits (serie de STMicroelectronics)
CH340	Chip convertidor USB a serie
Python	Lenguaje de programación de alto nivel usado en el sistema
python-can	Biblioteca Python para comunicación CAN
threading	Biblioteca de Python para manejo de hilos
time	Biblioteca de Python para control del tiempo
paho-mqtt	Biblioteca Python para cliente MQTT
dbc	Archivo de base de datos CAN (para decodificar tramas)

Índice

Índice de Figuras

Figura **Página**

Índice de Tablas

Tabla **Página**

CAPÍTULO 1

Introducción

Contenido

Este capítulo introduce el propósito general del proyecto, explicando la necesidad de sistemas de telemetría en el ámbito de los vehículos conectados y su relación con las tecnologías IoT (Internet of Things). Para ello se detalla la motivación que impulsa el desarrollo de este sistema, los objetivos a alcanzar y la estructura de la memoria, para facilitar una visión global del trabajo realizado.

1.1. Motivación

En los últimos años, el concepto de vehículo conectado ha cobrado una gran relevancia dentro del sector de la automoción, impulsado por la creciente integración de tecnologías de comunicación, computación embebida e IoT (Internet of Things). Esta evolución ha permitido que los vehículos no se limiten a ser simples medios de transporte, sino también a ser nodos inteligentes dentro de una red digital, capaces de recopilar, procesar e incluso compartir información en tiempo real.

Uno de los pilares fundamentales de esta transformación es la telemetría vehicular, que consiste en la obtención de datos del vehículo y su transmisión hacia plataformas externas para tareas de diagnosis, mantenimiento predictivo o gestión de flotas. La disponibilidad de buses de comunicación estandarizados como el CAN (Controller Area Network) y la presencia del conector OBD-II en prácticamente todos los vehículos modernos hacen posible el acceso a una gran cantidad de datos sin necesidad de modificar o añadir nada a la arquitectura interna del automóvil.

La demanda de soluciones de rápida implementación ha favorecido el desarrollo de dispositivos que permiten leer tramas CAN y enviar datos relevantes a través de protocolos de comunicación como MQTT, integrándose en infraestructuras IoT y servicios en la nube. El despliegue de redes 5G, gracias a proporcionar velocidades de transmisión elevadas y baja latencia, ha resultado un factor clave para sistemas que requieran de disponibilidad y sincronización en tiempo real.

En este contexto, se enmarca el presente trabajo, cuya motivación principal se basa en desarrollar un sistema de telemetría para vehículos eléctricos que aproveche las tecnologías existentes (MQTT, 5G, Node-RED...) para habilitar la monitorización remota del vehículo mediante la extracción de datos del bus CAN.

1.2. Objetivos del proyecto

El principal objetivo de este Trabajo Fin de Grado es el diseño e implementación de un sistema de telemetría que permita la adquisición, procesamiento y transmisión de datos del bus CAN de un vehículo real, mediante el uso de tecnologías IoT y conectividad 5G, facilitando así su monitorización remota en tiempo real.

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

- Integrar una arquitectura de adquisición de datos CAN, utilizando una interfaz USB-CAN conectada al vehículo a través del conector OBD-II.
- Desarrollar un script en Python, capaz de capturar y decodificar tramas CAN relevantes, empleando un entorno Linux.

- Configurar una red 5G funcional, que permita la conexión entre el dispositivo de adquisición y el servidor de destino mediante el protocolo MQTT.
- Diseñar una interfaz gráfica en Node-RED, para la visualización remota, en tiempo real, de las señales del vehículo.
- Validar el sistema sobre un vehículo eléctrico real, concretamente un Nissan Leaf AZE0 del año 2014, extrayendo parámetros clave como velocidad, estado de carga, temperatura de batería, entre otros.
- Demostrar la viabilidad y escalabilidad del sistema, como solución adaptable a otros vehículos y aplicaciones del ámbito de la automoción conectada.

1.3. Estructura de la memoria

La presente memoria se encuentra estructurada en ocho capítulos, cada uno de los cuales aborda una parte del desarrollo del sistema de telemetría BusCAN. A continuación, se describe brevemente el contenido de cada uno:

- **Capítulo 1 - Introducción**: se presenta el contexto general del proyecto, la motivación que lo impulsa, los objetivos marcados y la organización del documento.
- **Capítulo 2 - Fundamento teórico**: se explican los conceptos teóricos necesarios para entender el funcionamiento del sistema, como el bus CAN, el protocolo OBD-II, la arquitectura IoT, la conectividad 5G y el protocolo MQTT.
- **Capítulo 3 - Hardware empleado**: se detallan los componentes físicos utilizados en el proyecto incluyendo el vehículo, los adaptadores de comunicación y el equipo de adquisición de datos.
- **Capítulo 4 - Desarrollo del sistema software**: se describe la implementación del código de adquisición y publicación de datos, así como la configuración del entorno Linux y la herramienta Node-RED.
- **Capítulo 5 - Sistema a bordo**: se analiza la plataforma de muestreo de los datos obtenidos en tiempo real mediante el sistema de adquisición, repasando la configuración de la red 5G, el funcionamiento del broker MQTT y el flujo de datos entre los distintos elementos del sistema.
- **Capítulo 6 - Validación y resultados**: se muestran los resultados obtenidos durante la puesta en marcha y validación del sistema, evaluando su funcionamiento y capacidad de monitorización en tiempo real.
- **Capítulo 7 - Conclusiones y líneas futuras**: se extraen las conclusiones principales del trabajo, se reflexiona sobre los aprendizajes obtenidos y se plantean posibles mejoras o futuras líneas de investigación.

Finalmente, se incluye la bibliografía utilizada a lo largo del proyecto y el anexo que contiene un enlace a un repositorio GitHub con todo lo necesario para montar el sistema de adquisición.

CAPÍTULO 2

Fundamento teórico

Contenido

Este capítulo presenta los fundamentos teóricos y tecnológicos necesarios para comprender el funcionamiento del sistema desarrollado. Se abordan los conceptos clave relacionados con la red de comunicación CAN utilizada en vehículos, el protocolo OBD-II como punto de acceso a dicha red, y las tecnologías IoT que permiten la transmisión y visualización de los datos obtenidos. Además, se detallan los principios del protocolo MQTT y el papel de la conectividad 5G en la arquitectura global del sistema.

2.1. Arquitectura de red CAN en automoción

El protocolo de comunicación CAN (Controller Area Network) es un sistema de comunicación en serie desarrollado originalmente por la empresa Bosch en los años 80, con el objetivo de reducir la complejidad del cableado en los vehículos y facilitar la comunicación entre diferentes unidades electrónicas de control (ECUs).

Es un estándar de red serie utilizado frecuentemente en automoción para la comunicación entre múltiples ECU (Unidades de Control Electrónico) sin necesidad de un nodo central. Estas unidades se conectan mediante dos hilos trenzados (CAN High y CAN Low) con terminadores de 120 ohmios en los extremos para garantizar una impedancia constante y evitar reflexiones. Este protocolo opera con señales diferenciales, siendo en estado recesivo (bit lógico "1"), ambas líneas alrededor de 2.5V con diferencia de potencial aproximadamente 0V; en estado dominante (bit lógico "0"), CANH es ≈ 3.5V y CANL ≈ 1.5V, logrando unos 2V de diferencia. Esta decodificación diferencial proporciona alta inmunidad al ruido EMI y confiabilidad en entornos automotrices críticos. [5]

Una de las principales ventajas del protocolo CAN es su arquitectura de red distribuida, que permite a múltiples nodos compartir un mismo medio físico de comunicación sin necesidad de un coordinador central. Cada nodo puede emitir mensajes y escuchar los que circulan por el bus, actuando de forma autónoma según su identificación (ID) de mensaje.

2.1.1. Características técnicas del bus CAN

El bus CAN es una red multi-maestro donde todos los nodos pueden transmitir mensajes sin requerir un coordinador central. Entre sus características más relevantes se encuentran:

- **Comunicación basada en mensajes**: cada mensaje lleva un identificador (ID) único que determina su prioridad en el bus. A menor valor binario del ID, mayor prioridad.
- **Transmisión diferencial**: utiliza dos líneas (CANH y CANL) en modo diferencial para mejorar la inmunidad al ruido electromagnético..
- **Velocidades de transmisión**: CAN 2.0 alcanza hasta 1 Mbps. CAN FD (Flexible Data-rate) permite hasta 8 Mbps en la fase de datos.
- **Control de errores**: mediante mecanismos como el *CRC*[1], el *bit stuffing*[2] y los *counters de error*[3] internos en cada nodo.

Un módulo CAN está compuesto de dos elementos básicos:

[1] El término *CRC* es un campo de verificación de errores que sirve para detectar si los datos recibidos han sido alterados durante la transmisión (por interferencias eléctricas, ruido, etc.).

[2] *Bit stuffing* es una técnica que se usa para mantener la sincronización entre los nodos durante la transmisión de datos.

[3] Los *counters de error* son contadores internos que tiene cada nodo CAN para saber cuántos errores ha cometido recientemente.

- **Controlador**: gestiona la construcción de las tramas CAN, comprueba los errores en la transmisión, o en otros nodos, y la detección de colisiones.
- **Transmisor y/o receptor (también llamado transceptor)**: es el módulo encargado de la codificación y decodificación de los mensajes en el bus, sincronización, control de los niveles de la señal y del control de acceso al medio.

El controlador y el transceptor son módulos independientes de los nodos, permitiendo que no tengan que destinar recursos en gestión de comunicación, acceso al medio, colisiones, etcétera. Aunque algunos microcontroladores poseen módulos CAN en un único encapsulado, internamente son circuitos independientes en la mayoría de casos.

Cualquier dispositivo que se conecte al bus puede enviar mensajes, y todos los nodos conectados al mismo lo recibirán. Para filtrar los tipos de mensajes se hace uso de un identificador asociado. De esta forma se consigue que cada nodo procese los mensajes que interesen, o por el contrario, descartar los mensajes.

2.1.2. Topología y capas funcionales del CAN

En una implementación típica, la topología de red CAN es de tipo bus lineal, con terminadores de 120 ohmios en cada extremo para evitar reflexiones. Cada nodo se conecta en paralelo, normalmente mediante conectores o empalmes en T. A nivel de protocolo, CAN no sigue estrictamente el modelo OSI de 7 capas, pero puede representarse mediante las siguientes capas:

- **Capa física**: define el tipo de cableado, conectores, niveles eléctricos, etc...
- **Capa de enlace de datos**: sus funciones, como el arbitraje de mensajes, la detección de errores y el control de acceso al medio, son gestionadas por el controlador CAN (generalmente integrado en el microcontrolador o como periférico dedicado). Este se encarga de interpretar las tramas, controlar el acceso al bus mediante el mecanismo de arbitraje por prioridad (basado en el identificador del mensaje) y aplicar las rutinas de comprobación y notificación de errores.
- **Capa de aplicación** (en combinación con protocolos superiores como UDS o OBD-II): define los mensajes específicos y su interpretación.

En cuanto a la capa física, únicamente son necesarios dos cables trenzados con una impedancia característica de 120Ω, con el propósito de interconectar todos los dispositivos en una misma red. Las señales de estos cables se denominan (normalmente) *CAN High* y *CAN Low* y, dependiendo de parámetros como el voltaje, el bus puede encontrarse en modo recesivo (ambos cables al mismo nivel de tensión) o en modo dominante (con una diferencia de al menos 1,5V). Para garantizar una correcta transmisión de las señales y evitar reflexiones que puedan interferir en la comunicación, es imprescindible conectar resistencias de terminación de 120Ω en cada extremo del bus. Este modo de comunicación proporciona una elevada protección frente a interferencias electromagnéticas, ya que

la lectura de los bits se basa en la diferencia de potencial entre dos cables trenzados, una magnitud que se mantiene estable incluso en entornos con ruido electromagnético.

En los vehículos modernos, existen varias redes CAN segmentadas por función o criticidad: por ejemplo, una red CAN de alta velocidad para motor y frenado, una red CAN de confort para sistemas como climatización y ventanillas, e incluso redes CAN de seguridad para airbags y sensores de impacto. Estas redes pueden estar interconectadas mediante gateways (puentes), lo que permite gestionar el flujo de información entre dominios sin comprometer la seguridad o el rendimiento. En este proyecto se hace uso de la red CAN accesible a través del conector OBD-II, cuya descripción se detalla en la siguiente sección.

2.1.3. Tramas y arbitraje del bus

El protocolo CAN presenta distintos formatos de trama, cada una con un propósito distinto dentro del funcionamiento del propio protocolo, entre los más importantes se encuentran:

- **Tramas de datos**: consiste en tramas que transmiten información entre los distintos nodos, pudiendo ser esta información enviada y recibida por uno o varios de ellos.
- **Tramas de error**: tramas utilizadas cuando algún nodo de la red detecta un error en alguno de los mensajes transmitidos por otros nodos, violando las normas del formato de tramas CAN.
- **Tramas de petición remota**: cuyo uso se basa en la solicitud de envío de información a otro nodo. Su funcionamiento es el siguiente, se envía una trama con el identificador del nodo del que se requiere el envío de información, éste lo recibe y devuelve la información con una trama de datos.
- **Tramas de sobrecarga**: al igual que en las tramas de error, violan las normas del formato de tramas CAN. Es enviada por un nodo cuando está sobrecargado, provocando que el bus incluya un retardo extra entre tramas.

Cuando el bus está en reposo (no existe intercambio), el nivel recesivo del bus se mantiene constante. Cuando se requiere enviar mensajes que ocupen más de una trama de datos, se separan entre si por una secuencia predeterminada, la cual se le denomina espacio inter-trama, compuesta por 3 bits recesivos.

Las tramas CAN de datos se pueden clasificar en dos tipos principales:

- **Tramas estándar (CAN 2.0A)**: Con 11 bits de identificador.
- **Tramas extendidas (CAN 2.0B)**: Con 29 bits de identificador.

Las tramas contienen campos como el SOF (Start of Frame), ID, RTR (Remote Transmission Request), DLC (Data Length Code), datos (hasta 8 bytes en CAN 2.0, 74 en CAN FD), CRC, ACK y EOF. Tal y como se ve en la Figura 2.1.

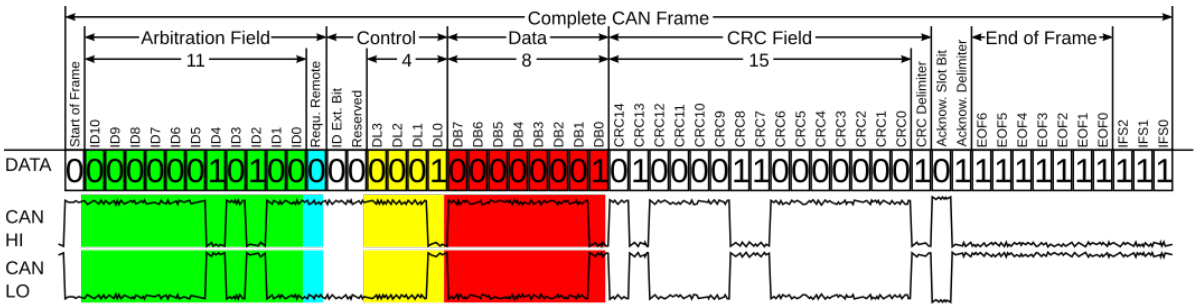

Figura 2.1: Trama CAN en formato base con las tensiones eléctricas del bus y sin bits de relleno. [1]

El protocolo CAN tiene una característica denominada arbitraje, mediante la cual se controla el acceso al medio por parte de los nodos y se evitan posibles colisiones en las comunicaciones. El arbitraje es un proceso no destructivo: si dos nodos intentan transmitir al mismo tiempo, el nodo con el ID de menor valor continua transmitiendo y el otro detiene su intento. Esto garantiza que siempre se priorice la información más importante sin perder datos.

En el bus de comunicación, se considera que los bits dominantes corresponden al nivel lógico "0", mientras que los bits recesivos se asocian con el valor lógico "1". Al comienzo de las tramas transmitidas por los nodos, se incluye un campo destinado al arbitraje, el cual coincide con el identificador del nodo emisor. Antes de iniciar la transmisión, se requiere que los nodos supervisen el estado del bus durante un intervalo sin actividad. En caso de que dos nodos intenten transmitir al mismo tiempo, los bits dominantes tienen prioridad sobre los recesivos. De este modo, el nodo que transmite una trama con bits dominantes (identificador más bajo) puede detectar una colisión si otro nodo transmite bits recesivos (identificador más alto). Para que la detección de colisiones sea efectiva, es necesario que los nodos mantengan una sincronización adecuada, asegurando que las frecuencias de reloj de los controladores CAN se encuentren dentro de los márgenes permitidos.

Cuando se detecta una colisión, se interrumpe inmediatamente la transmisión y se espera a que el otro nodo finalice su comunicación antes de reintentar el envío.

2.1.4. Aplicaciones en automoción y segmentación de redes

El diseño del bus CAN buscó facilitar la comunicación entre distintos dispositivos electrónicos dentro de un vehículo sin necesidad de conexiones punto a punto, reduciendo la cantidad de cableado y mejorando la fiabilidad de la comunicación [6]. Se encuentran distintas aplicaciones en automoción como pueden ser:

- Control de motor y transmisión.
- Sistemas de seguridad (airbags, ABS, EP, etc).
- Infoentretenimiento (control de audio, GPS, pantallas, etc).
- Climatización.

- Sistemas de asistencia al conductor (ADAS), como el control de crucero adaptativo, mantenimiento de carril, etc.
- Instrumentación (velocidad, combustible, temperatura de motor, etc).
- Gestión de batería en vehículos eléctricos (BMS). [7]

Haciendo del bus CAN un elemento crítico al ser rápido, robusto contra interferencias y con priorización de mensajes [8].

A medida que la electrónica en los vehículos se ha ido desarrollando, no es suficiente con tener únicamente una red CAN. Por ello se realiza lo que se conoce como segmentación de la red en buses CAN independientes o jerarquizados, siguiendo la siguiente estructura por norma general:

- **CAN de alta velocidad (High-Speed CAN, hasta 1 Mbps)**: se usa para comunicaciones críticas, como motor, transmisión o seguridad activa (ABS, ESP).
- **CAN de baja velocidad (Low-Speed Fault-Tolerant CAN, hasta 125 kpbs)**: sistemas menos críticos como ventanas eléctricas, luces o climatización [9].
- **Redes jerárquicas**: diferentes buses CAN interconectados a través de gateways que filtran y traducen mensajes de una red a otra [10].
- **Dominios funcionales**: en los vehículos eléctricos se tiende a organizar la arquitectura en dominios, cada uno con su propio bus CAN o incluso su propia red Ethernet [7].

Segmentar la red permite que la carga de datos se reparta mejor, se aíslen los fallos y se facilite la integración de nuevos sistemas.

2.2. Conector OBD-II: protocolo y disposición de pines

El conector OBD-II (On-Board Diagnostics II) es un estándar internacional que define tanto el conector físico como los protocolos de comunicación utilizados para el diagnóstico a bordo de vehículos. Su implementación fue inicialmente obligatoria en Estados Unidos a partir de 1996 para todos los vehículos ligeros, y posteriormente adoptada por la Unión Europea bajo la normativa E-OBD [11].

Este conector representa la principal interfaz de acceso al sistema de diagnóstico y a la red de comunicaciones interna del vehículo, lo que lo convierte en una herramienta fundamental tanto para talleres y fabricantes como para desarrolladores de soluciones de telemetría, mantenimiento predictivo y análisis de comportamiento [12].

2.2.1. Conector físico: pinout y ubicación

El conector OBD-II es de tipo D-sub de 16 pines y suele encontrarse ubicado bajo el salpicadero del lado del conductor. Aunque no todos los pines están activos en todos los vehículos, hay una asignación estándar que permite identificar los protocolos soportados y los canales disponibles para comunicación. En la Tabla 2.1 se muestra un resumen de la disposición de pines más relevante para este proyecto.

Pin	Nombre	Descripción
4	Chasis Ground	Masa de chasis del vehículo
5	Signal Ground	Masa de señal del sistema
6	CAN High	Línea CAN-H (ISO 15765-4 y variantes)
14	CAN Low	Línea CAN-L (ISO 15765-4 y variantes)
16	Battery Power	Alimentación de batería (12 V)

Tabla 2.1: PinOut OBD-II ISO 15765-4.

2.2.2. Protocolos de comunicación soportados por OBD-II

- **ISO 9141-2 / ISO 14230 (KWP2000)**: utilizados principalmente en vehículos asiáticos y europeos anteriores a 2008.
- **SAE J1850 PWM/VPW**: comunes en vehículos estadounidenses.
- **ISO 15765-4 (CAN)**: protocolo más extendido desde 2008, obligatorio en vehículos nuevos vendidos en la UE y EE. UU.

Este proyecto hace uso del protocolo ISO 15765-4 (CAN), que permite acceder a la red CAN de alta velocidad del vehículo a través de los pines 6 y 14. A partir de ahí, se pueden interceptar tramas que contienen datos como velocidad, estado de carga, temperatura de la batería, posición del acelerador, entre otros.

En este contexto, el OBD-II actúa como una puerta de entrada no intrusiva y estandarizada a los datos internos del vehículo, facilitando la implementación de sistemas de telemetría sin necesidad de intervenir en el cableado original o en las ECU.

2.2.3. Funcionamiento a nivel de mensaje: solicitud y respuesta

El protocolo OBD-II define un conjunto estandarizado de servicios (llamados *modes*) y parámetros (*PID's*). Los mensajes típicos siguen la siguiente estructura:

- El nodo externo (escáner, interfaz USB-CAN, etc...) envía una solicitud al ID 0x7DF, que es un *multidifusión funcional*[4].

[4] En el ámbito de buses CAN, *multidifusión funcional* es un tipo de comunicación en la que un nodo transmite un mensaje sin destinatario específico, y todos los nodos lo reciben, respondiendo solo aquellos cuya función corresponde con el mensaje.

- La ECU responde desde su ID específico (0x7EB, 0x7E9, etc...).

2.3. Principios de telemetría vehicular

La telemetría vehicular es la disciplina encargada de capturar, transmitir y procesar información generada por un vehículo en tiempo real o diferido. Esta tecnología se ha convertido en un pilar fundamental para múltiples aplicaciones: desde la gestión de flotas hasta el análisis predictivo, el mantenimiento remoto y los sistemas avanzados de seguridad [13].

En este proyecto, la telemetría se basa en la recopilación de datos a través del bus CAN mediante el conector OBD-II, con su posterior transmisión hacia una plataforma remota mediante tecnologías IoT.

2.3.1. Componentes de un sistema de telemetria vehicular

Un sistema de telemetría vehicular típico está compuesto por los siguientes elementos:

- **Unidad de adquisición de datos (DAQ)**: se encarga de obtener información directamente desde el vehículo, normalmente a través del bus CAN utilizando el conector OBD-II. Esta unidad puede ser un microcontrolador, un ordenador embebido o una interfaz especializada (como USB-CAN).

- **Procesador local**: interpreta y, si es necesario, filtra o transforma los datos antes de enviarlos. En este proyecto, esta tarea la realiza un script programado en Python.

- **Módulo de comunicaciones**: transmite los datos procesados a una plataforma remota mediante protocolos como MQTT, HTTP o TCP/IP, utilizando tecnologías como Wi-Fi, 4G/5G o satélite.

- **Servidor remoto o plataforma IoT**: recibe, almacena y visualiza los datos. En este proyecto se emplea Node-RED, una herramienta de desarrollo visual para flujos de datos, integrada con un broker MQTT (Mosquitto). En este proyecto se ha seguido la arquitectura que se ve en la Figura 2.2.

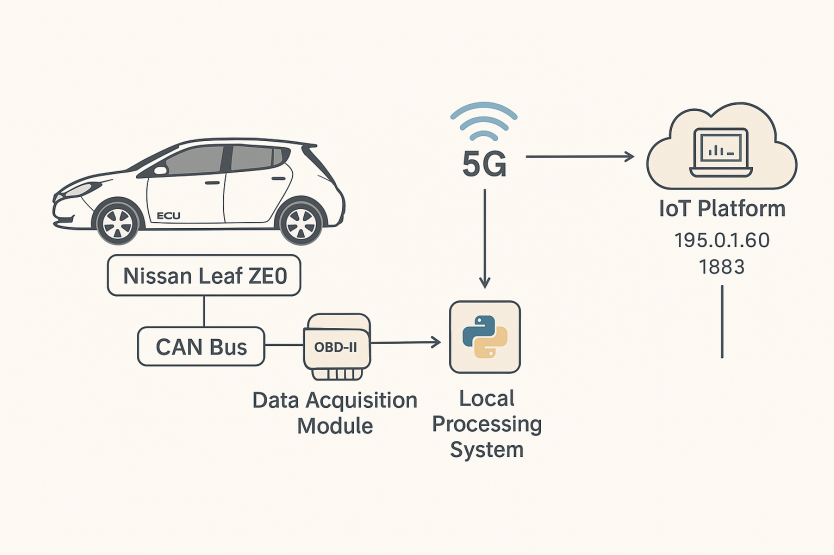

Figura 2.2: Arquitectura general del sistema de telemetria vehicular IoT.

2.3.2. Ventajas y retos de la telemetría vehicular

Entre las ventajas más destacadas se encuentran:

- Acceso remoto a parámetros clave en tiempo real.
- Reducción de costes de mantenimiento y operación.
- Mejora de la seguridad y anticipación de fallos.
- Posibilidad de integración con sistemas inteligentes o plataformas Big Data.

Sin embargo, también existen desafíos relevantes, como:

- Estándares no homogéneos entre fabricantes.
- Necesidad de decodificación específica de tramas (uso de archivos DBC).
- Gestión segura de los datos transmitidos.
- Requisitos de baja latencia y alta disponibilidad en tiempo real.

Este proyecto aborda una solución funcional y modular a muchos de estos retos, implementando un sistema de adquisición y transmisión de datos que se apoya en estándares abiertos y tecnologías ampliamente adoptadas.

2.4. Tecnologías IoT aplicadas a vehículos

El paradigma del Internet of Things (IoT) ha supuesto un cambio significativo en la forma en la que los sistemas físicos interactúan con su entorno digital. En el ámbito de la automoción, el IoT permite que los vehículos se comuniquen con infraestructuras externas, con otros vehículos y con plataformas en la nube, generando un ecosistema conectado que habilita aplicaciones avanzadas en áreas como la seguridad vial, el mantenimiento inteligente y la gestión eficiente de recursos.

2.4.1. Aplicaciones en automoción

La incorporación de tecnologías IoT en vehículos ha permitido el desarrollo de funcionalidades como la monitorización remota, el diagnóstico en tiempo real y la optimización de rutas y consumos. A través de sensores internos y externos, los vehículos pueden proporcionar información sobre su estado mecánico, eléctrico y energético, lo cual es especialmente útil en contextos como la gestión de flotas, la movilidad eléctrica o las ciudades inteligentes. La capacidad de recopilar y transmitir esta información en tiempo real permite tomar decisiones más rápidas, reducir costes operativos y mejorar la eficiencia general del sistema de transporte.

2.4.2. Arquitectura típica

Una arquitectura IoT aplicada a vehículos suele incluir un dispositivo embebido que actúa como nodo de adquisición y comunicación. Este nodo recoge datos internos del vehículo y los transmite hacia una plataforma remota utilizando protocolos de comunicación (como puede ser MQTT). Los datos pueden visualizarse o analizarse desde una interfaz web o integrarse en sistemas de mayor escala, como plataformas de análisis predictivo o paneles de gestión (dashboards).

En el caso de este proyecto, la arquitectura propuesta se basa en una interfaz USB-CAN conectada a un equipo con sistema Linux, desde el cual un script en Python captura y publica los datos a través del protocolo MQTT sobre una red 5G. Estos datos son finalmente presentados en una interfaz gráfica desarrollada en Node-RED, permitiendo así la supervisión remota del vehículo en tiempo real.

El uso de componentes modulares, tecnologías abiertas y protocolos estandarizados asegura la escalabilidad y flexibilidad del sistema, facilitando su aplicación en distintos tipos de vehículos o entornos industriales sin requerir modificaciones estructurales.

2.5. Tecnologías de comunicación inalámbricas para IoT

Las tecnologías del Internet de las Cosas (IoT) proporcionan una conectividad fluida entre diversos objetos físicos y virtuales, el creciente número de aplicaciones IoT plantea la necesidad de transmitir, almacenar y procesar una gran cantidad de datos. Por tanto, es necesario habilitar un sistema capaz de gestionar los crecientes requisitos del tráfico con el nivel requerido de QoS (Calidad de Servicio). Los dispositivos IoT se vuelven más complejos debido a la diversidad de componentes, sensores

e interfaces de red, exigiendo fuentes de alimentación móviles, QoS, fiabilidad, seguridad y otros requisitos. Por lo tanto, se requieren nuevas tecnologías de comunicación IoT, en los últimos años se han desarrollado nuevas tecnologías de comunicación inalámbrica, como diferentes WSN (Redes de sensores inalámbricas), WLAN (Redes de Área Local Inalámbricas), LR-WPAN (Redes de Área Amplia de Baja Potencia), muchas de estas basadas en la arquitectura del protocolo IP. En la Figura 2.3 se pueden ver las distintas tecnologías actuales de comunicación inalámbrica con soporte IoT.

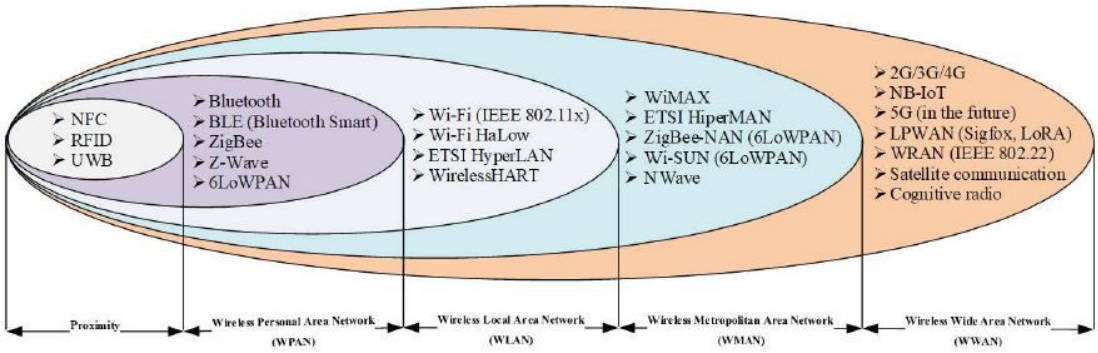

Figura 2.3: Tecnologías de comunicación inalámbrica para IoT. [2]

2.5.1. Protocolos de comunicación IoT

El desarrollo de nuevos protocolos y arquitecturas de comunicación para futuras aplicaciones de conceptos IoT tendrá uno de los papeles clave en los próximos años. Los protocolos utilizados en los servicios de internet tradicionales normalmente no son apropiados para IoT porque se tienen en cuenta cosas como la eficiencia energética. Como consecuencia a esto, se han desarrollado un conjunto de protocolos de capa de aplicación como CoAP (Constrained Application Protocol), MQTT (Message Queue Telemetry Transport), MQTT-SN (MQTT for Sensor Networks), AMQP (Advanced Message Queuing Protocol), DDS (Data Distribution Service) y WebSocket. Según los protocolos usados, los 'objetos' IoT se pueden clasificar en dos grupos, los que soportan y los que no soportan la arquitectura del protocolo TCP/IP. En la Tabla 2.2 se puede ver una tabla con los protocolos de aplicación IoT principales y algunas de sus características.

Protocol	Standard	RESTful	Transport	Security	QoS
CoAP	IETF RFC 7252	Yes	UDP	DTLS	Yes
MQTT	OASIS Standard	No	TCP	TLS/SSL	Yes
MQTT-SN	IBM Zurich Research website	No	TCP	TLS/SSL	Yes
XMPP	IETF RFC 6120, 6121	No	TCP	TLS/SSL	No
AMQP	ISO and IEC	No	TCP	TLS/SSL	Yes
DDS	OMG *(Object Management Group)*	No	UDP	DTLS	Yes
HTTP	IETF RFC (2068, 2616, 7230), W3C	Yes	TCP	SSL	No
WebSocket	IETF Internet Draft	Yes	TCP	TLS/SSL	No

Tabla 2.2: Protocolos de aplicación en IoT. [2]

2.5.2. Wi-Fi

El IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) definió la serie de especificaciones 802.11 para redes inalámbricas locales WLAN (Wireless Local Area Network) que se conocen mayormente por el término Wi-Fi.

El Wi-Fi (Wireless Fidelity) está diseñado para su uso en longitudes de transferencia de datos medias y cortas (hasta unos pocos cientos de metros), pero las longitudes exactas no se pueden definir porque dependen de múltiples factores, como son variaciones en los estándares, distancia entre dispositivos, condiciones atmosféricas, visibilidad óptica entre antenas, calidad del hardware, etc. El pricipal desafío en la implementación de un sistema IoT que incluya tecnología Wi-Fi es el alto consumo de energía en comparación con Bluetooth y ZigBee.

Es por ello que a lo largo del tiempo se han desarrollado diferentes versiones del estándar IEEE 802.11 con el objetivo de lograr mejoras constantes y eliminar ciertas restricciones. En comparación con las primeras versiones, hay mejoras para superar algunos de los problemas que se han mencionado anteriormente. Por ejemplo, IEEE 802.11ah (Low-Power Wi-Fi) admite un amplio espectro de aplicaciones de IoT al mismo tiempo que es energéticamente eficiente, admite QoS, escalabilidad, así como soluciones con bajos gastos. En la Tabla 2.3 se pueden ver distintas características de los estándares Wi-Fi actuales.

Standard	Year	Frequency (GHz)	Bandwidth (MHz)	Throughput (Mbps)	Modulation	Indoor	Outdoor
802.11-1997	Jun. 1997	2.4	22	1, 2	DSSS, FHSS	20 m	100 m
802.11b	Sep. 1999	2.4	22	1, 2, 5.5, 11	DSSS	35 m	140 m
802.11a	Sep. 1999	5	–	–	OFDM	35 m	120 m
802.11j	Nov. 2004	4.9/5.0	–	–	OFDM	?	?
802.11p	Jul. 2010	5.9	5/10/20	6–54	OFDM	?	1000 m
802.11y	Nov. 2008	3.7	–	–	OFDM	?	5000 m
802.11g	Jun. 2003	2.4	–	–	OFDM	38 m	140 m
802.11n	Oct. 2009	2.4/5	20	Up to 288.8	MIMO-OFDM	70 m	250 m
802.11n	Oct. 2009	2.4/5	40	Up to 600	MIMO-OFDM	70 m	250 m
802.11ac	Dec. 2013	5	20	Up to 346.8	MIMO-OFDM	35 m	?
802.11ac	Dec. 2013	5	40	Up to 800	MIMO-OFDM	35 m	?
802.11ac	Dec. 2013	5	80	Up to 1733.2	MIMO-OFDM	35 m	?
802.11ac	Dec. 2013	5	160	Up to 3466.8	MIMO-OFDM	35 m	?
802.11ad	Dec. 2012	60	2160	Up to 6757	OFDM	3.3 m	?
802.11af	Feb. 2014	0.054–0.79	6–8	Up to 568.9	MIMO-OFDM	?	?
802.11ah	Dec. 2016	0.7/0.8/0.9	1–16	Up to 8.67	MIMO-OFDM	?	?

Tabla 2.3: Características de los estándares IEEE 802.11 (Wi-Fi). [2]

2.5.3. Tecnologías de comunicación móvil

Muchas aplicaciones de IoT se basan en la transmisión de datos a través de sistemas móviles, como 2G (GSM, D-AMPS, PDC), 2.5G (GPRS), 3G (UMTS/WCDMA, HSPA, HSUPA, EVDO) y 4G (LTE, LTE-A), el desarrollo de ciertas aplicaciones también está condicionado por el desarrollo de las tecnologías 5G. La conectividad del IoT en el contexto de las redes celulares se conocen como M2M (Máquina a Máquina) o MTC (Comunicación de Tipo Máquina) dentro del 3GPP. Las tecnologías 3G y 4G, como 3GPP LTE, permiten una amplia cobertura, compatibilidad con QoS, movilidad e itinerancia, escalabilidad, un alto nivel de seguridad, facilidad de gestión y conectividad de sensores mediante

una API estandarizada. LTE-A y WiMAX (IEEE 802.16m) ofrecen mayor velocidad y escalabilidad, además de bajos costos, satisfaciendo la mayoría de requisitos del IoT aunque presentando algunos problemas y desafíos pendientes, como la QoS y la congestión de red debido a la gran cantidad de dispositivos/nodos.

Satisfacer las crecientes demandas del mercado de IoT, que incluye la necesidad de superar el problema de la fragmentación tecnológica y el desafío correspondiente de garantizar la interoperabilidad, el 3GPP ha realizado mejoras relacionadas a través de nuevas versiones.

2.5.4. Conclusiones de las tecnologías de IoT

Se concluye que el crecimiento del IoT (Internet of Things) ha impulsado el uso de diversas tecnologías inalámbricas, cuyo objetivo es permitir una conectividad fluida entre dispositivos en cualquier momento y lugar. No obstante, se identifican importantes desafíos, como la elevada demanda de datos, la eficiencia energética y la necesidad de garantizar un rendimiento adecuado del sistema.

Para dar respuesta a estas exigencias, se han desarrollado nuevas arquitecturas de red y protocolos de comunicación más eficientes, tanto dentro como fuera del modelo TCP/IP. Dado que muchos protocolos tradicionales no son aptos para aplicaciones IoT, se ha puesto especial atención en soluciones que reduzcan el consumo energético y mejoren la calidad del servicio.

CAPÍTULO 3

Hardware empleado

Contenido

En este capítulo se describen los elementos físicos que componen el sistema desarrollado. Se detallan los dispositivos utilizados para la adquisición de datos del vehículo, el equipo informático encargado del procesamiento y transmisión de información, así como los componentes necesarios para establecer la comunicación entre el bus CAN del vehículo y el sistema de telemetría. Esta sección es fundamental para comprender la arquitectura física del sistema y su integración en un entorno real.

3.1. Vehículo utilizado: Nissan Leaf AZE0

Para la validación del sistema de telemetría propuesto en este proyecto, se ha empleado un Nissan Leaf AZE0 del año 2014, que se puede apreciar en la Figura 3.1, una de las primeras generaciones de vehículos eléctricos de producción masiva, ampliamente disponible en el mercado europeo y japonés.

El Nissan Leaf es un vehículo 100 % eléctrico equipado con una arquitectura de red CAN distribuida que conecta múltiples unidades electrónicas de control (ECUs), responsables de la gestión del sistema de propulsión, baterías, climatización, sistema de carga, frenos regenerativos, entre otros. Esta arquitectura lo convierte en una plataforma idónea para la adquisición de datos mediante sistemas externos no intrusivos.

Figura 3.1: Fotografía del vehículo Nissan Leaf AZE0.

Entre las características más relevantes del modelo AZE0 destacan:

- **Motorización eléctrica** con una potencia nominal de aproximadamente 80 kW.
- **Pack de baterías de ion-litio** de 24 kWh de capacidad, refrigeradas por aire.
- **Conector OBD-II** accesible desde el habitáculo, que permite acceder a la red CAN de alta velocidad.
- **Protocolo de comunicación ISO 15765-4 (CAN)** para diagnóstico y adquisición de datos.

En el Leaf AZE0 se identifican claramente tres redes CAN principales, cada una especializada en diferentes dominios funcionales, estos son:

- **CAR-CAN**, bus de comunicaciones principal, encargado de integrar y coordinar la mayoría de sistemas de control y seguridad.

- **EV-CAN**, dedicado exclusivamente a la gestión del sistema eléctrico de alta tensión del vehículo.

- **AV-CAN**, red CAN de baja prioridad destinada a la gestión de sistemas de infoentretenimiento y componentes audiovisuales.

Además el vehículo cuenta con un **VCM** (Vehicle Control Module) que actúa como *gateway* entre el CAR-CAN y el EV-CAN, permitiendo intercambiar información entre la propulsión eléctrica y los sistemas generales del vehículo [14].

3.2. Adaptador OBD-II a DB9

Para acceder físicamente a la red de comunicaciones del vehículo se ha utilizado un adaptador OBD-II a DB9, que permite establecer la conexión eléctrica entre el conector de diagnóstico estándar del Nissan Leaf y la interfaz USB-CAN empleada en el sistema de adquisición.

El conector OBD-II dispone de 16 pines con asignación definida por normativa, aunque no todos se utilizan en todos los vehículos. El protocolo CAN usado en el Leaf es ISO 15765-4 CAN (11-bit ID, 500 kBd).

En la siguiente Figura 3.2 se puede ver la vista del conector de diagnosis del vehículo. Los colores de los símbolos bus CAN están seleccionados con el código de color del cable Ethernet de cuatro pares.

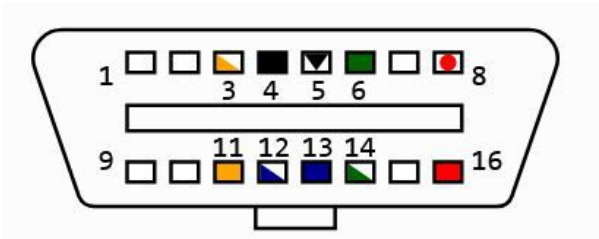

Figura 3.2: PinOut contactos conector diagnosis Nissan Leaf OBD-II (mating-end view). [3]

Como se muestra en la Tabla 3.1, el Leaf utiliza cinco contactos, masa del chasis, masa de señal, Car-CAN high, Car-CAN low y +12V DC permanente. La designación de los pines restantes es específica del Leaf: AV-CAN low, +12 DC cuando el vehículo está encendido, AV-CAN high, EV-CAN low, EV-CAN alto.

Pin	Símbolo	Designación
1		No connection
2		No connection
3		AV-CAN low
4		Chassis ground
5		Signal ground
6		Car-CAN high
7		No connection
8		+12 V DC when vehicle powered on
9		No connection
10		No connection
11		AV-CAN high
12		EV-CAN low
13		EV-CAN high
14		Car-CAN low
15		No connection
16		Permanent +12V DC

Tabla 3.1: PinOut OBD-II Nissan Leaf (símbolos según ISO 15031-3). [3]

Para cumplir con el objetivo de este proyecto, se va a realizar el conexionado, a través del puerto OBD-II, con el bus CAR-CAN del vehículo, debido a que es el bus principal y como se ha mencionado anteriormente cuenta con intercambio de información con los otros nodos.

Por otro lado, el conector DB9 es una interfaz ampliamente utilizada en módulos USB-CAN, siguiendo comúnmente la especificación de pines de CiA (CAN in Automation). Para la correcta correspondencia entre ambos conectores, se ha utilizado el siguiente mapeo de pines que se muestra en la Tabla 3.2.

Pin OBD-II	Señal	Pin DB9
6	Car-CAN High	3
14	Car-CAN Low	5

Tabla 3.2: Conversión PinOut OBD-II ISO 15765-4 a DB9.

Este adaptador es pasivo, es decir, no realiza ningún procesamiento o conversión eléctrica. Su única función es redirigir correctamente las señales físicas desde el conector OBD-II hacia el dispositivo USB-CAN, garantizando la compatibilidad con los niveles eléctricos de la red CAN del vehículo (normalmente 2.5 V de nivel medio, con oscilaciones diferenciales para los estados dominantes y recesivos del bus).

3.3. USB-CAN: funcionamiento e integración

Para establecer la comunicación entre el bus CAN del vehículo y el sistema de adquisición, se ha empleado el adaptador Robotell USB-CAN, una solución económica y ampliamente utilizada en aplicaciones de diagnóstico y monitorización de redes CAN.

El motivo esta elección para la adquisición de datos del vehículo a través del bus CAN se basa en la amplia compatibilidad con protocolos CAN, incluido el ISO 15765-4, la facilidad que presenta para integrarlo en los entornos de desarrollo basados en Python y la no necesidad de instalar drivers propietarios, accediento mediante USB.

A diferencia de otras interfaces comerciales más costosas, esta solución ofrece una buena calidad-precio, soporte para varios comandos serie y una gran comunidad detrás facilitando su uso para investigación.

Durante las fases iniciales del proyecto se evaluaron distintas interfaces de comunicación con el bus CAN del vehículo. En primer lugar, se consideró el uso de dispositivos ELM327, muy económicos, pero se descartaron por sus limitaciones de velocidad, acceso restringido a mensajes crudos y pobre rendimiento en entornos que requieren de lectura intensiva de tramas. Por otro lado, interfaces profesionales como Kvaser o Peak-System ofrecían altas prestaciones, soporte oficial y robustez industrial, pero su elevado coste y dependencia de entornos propietarios los hacían poco adecuados para un proyecto académico.

3.3.1. Características técnicas

El adaptador Robotell USB-CAN (ver Figura 3.3) está basado en un microcontrolador STM32 y utiliza un chip CH340 para la conversión de USB a serie. Este dispositivo implementa un protocolo binario propio, no compatible con SLCAN, y se comunica a través de un puerto serie virtual, generalmente reconocido por el sistema operativo como /dev/ttyUSBx en Linux o COMx en Windows.

Entre sus características destacan:

- Compatibilidad con velocidades de hasta 1 Mbps, incluyendo 500 kbps, que es la utilizada en el Nissan Leaf AZE0.
- Soporte para tramas estándar (11 bits) y extendidas (29 bits).
- Configuración de filtros de recepción, permitiendo la selección de mensajes específicos según ID y máscara.
- Interfaz de comunicación mediante protocolo binario, facilitando la integración con diversas aplicaciones y lenguajes de programación.

Figura 3.3: PCB Robotell USB-CAN interface. [4]

3.3.2. Integración con el sistema de adquisición

El adaptador se conecta al puerto USB del host, se ha hecho uso de la biblioteca Python *python-can*, que incluye soporte nativo para este tipo de dispositivos mediante su backend específico. El adaptador se conecta al host donde se ejecuta un script en Python encargado de la adquisición y procesamiento de las tramas CAN.

Esta configuración permite la recepción en tiempo real de las tramas CAN emitidas por el vehículo. El script procesa estas tramas, decodificando las señales de interés y publicándolas a través del protocolo MQTT hacia una interfaz gráfica desarrollada en Node-RED.

3.4. Hardware de adquisición (host)

El hardware de adquisición actúa como unidad central del sistema de telemetría, encargándose de la recepción, interpretación y publicación de las tramas CAN obtenidas del vehículo. Para esta función se ha utilizado un ordenador portátil convencional con sistema operativo Linux Ubuntu, sobre el cual se ha desplegado el entorno de desarrollo, las herramientas de comunicación y los servicios de publicación MQTT.

La elección de un entorno Linux responde a la necesidad de utilizar herramientas de código abierto como python-can, mosquitto, Node-RED y otras utilidades que ofrecen una mayor flexibilidad y control en el desarrollo de sistemas basados en CAN y MQTT. Además, Linux proporciona soporte nativo para la gestión de dispositivos USB en tiempo real, facilitando la integración con el adaptador Robotell USB-CAN.

3.4.1. Función del host en el sistema

Dentro de la arquitectura del sistema, el host cumple varias funciones clave:

- **Recepción de tramas CAN** desde el adaptador USB-CAN mediante un puerto serie virtual.
- **Procesamiento y decodificación** de los mensajes CAN, aplicando filtros, escalado, offset y selección de señales específicas.
- **Publicación de los datos** en formato estructurado (por ejemplo, JSON) utilizando el protocolo MQTT, hacia un broker alojado localmente o en red.
- **Interacción con la interfaz gráfica Node-RED**, donde los datos son visualizados en tiempo real para tareas de monitorización remota.

En fases iniciales del proyecto, el host también se utilizó como entorno de desarrollo para la escritura y depuración del script Python, así como para la configuración y pruebas del broker MQTT (Mosquitto). Posteriormente, este entorno se consolidó como el núcleo funcional del sistema, permitiendo realizar las pruebas de campo en el propio vehículo.

La modularidad del sistema permite sustituir este host por alternativas más compactas, como una Raspberry Pi o un dispositivo embebido con Linux, en caso de que se desee miniaturizar la solución para aplicaciones industriales o comerciales.

3.5. Disposición del sistema físico en el vehículo

La correcta integración física del sistema de telemetría en el vehículo es un aspecto clave para garantizar la seguridad, la fiabilidad del funcionamiento y la no intrusividad sobre el sistema eléctrico original del automóvil. En este proyecto, todos los elementos han sido dispuestos de forma accesible y segura, sin modificar componentes originales ni comprometer la operación del vehículo.

3.5.1. Ubicación de los componentes

El sistema completo se compone de los siguientes elementos principales:

- **Adaptador OBD-II a DB9:** conectado directamente al puerto OBD-II del Nissan Leaf, ubicado bajo el volante, en la zona inferior del salpicadero. Este conector proporciona acceso directo al bus CAN de alta veloci**dad del vehículo.**

  Adaptador Robotell USB-CAN: conectado al extremo DB9 del adaptador OBD-II. Este módulo queda alojado en la parte inferior del puesto de conducción, sujeto con fijaciones no permanentes (bridas o cinta de doble cara) para evitar movimientos bruscos durante la conducción.

- **Equipo host (portátil con Linux):** colocado en el asiento del copiloto durante las pruebas, alimentado mediante el puerto USB del propio vehículo o una batería externa. La conexión al adaptador USB-CAN se realiza mediante cable USB estándar.
- **Router o módem 5G:** cuando se ha requerido conexión remota en tiempo real, se ha utilizado un router industrial 5G ubicado en el maletero, proporcionando conectividad estable para la publicación de datos vía MQTT.

3.5.2. Consideraciones de integración

Durante la instalación se han tenido en cuenta las siguientes consideraciones:

- **No interferencia en la conducción**: todos los componentes han sido colocados de manera que no interfieran con los mandos, pedales o visibilidad del conductor.
- **Seguridad eléctrica**: se ha evitado cualquier derivación o conexión directa al cableado del vehículo. Todo el sistema es completamente pasivo respecto a la red CAN, operando únicamente como lector.
- **Facilidad de montaje y desmontaje**: el sistema puede retirarse completamente sin dejar rastro físico, en menos de dos minutos, lo que permite reutilizarlo o transferirlo a otro vehículo de forma sencilla.

Esta disposición ha sido especialmente útil para validar el sistema durante pruebas reales en carretera y en estacionamiento, permitiendo una adquisición fiable y estable de datos en situaciones de operación real.

CAPÍTULO 4

Desarrollo del sistema software

Contenido

En este capítulo se describe el proceso de implementación del software encargado de la adquisición, decodificación y transmisión de los datos obtenidos del vehículo. Se detallan las herramientas y entornos utilizados, la estructura del script principal en Python y su integración con el adaptador Robotell USB-CAN. Además, se aborda la configuración del protocolo MQTT para el envío de datos, así como el diseño de la interfaz gráfica de usuario mediante Node-RED, encargada de representar en tiempo real las señales relevantes del vehículo. Este capítulo constituye el núcleo funcional del sistema, en el que convergen la captación física de datos y su visualización remota.

4.1. Entorno de desarrollo

El desarrollo del sistema de adquisición y publicación de datos se ha llevado a cabo en un entorno basado en Linux, concretamente utilizando la distribución Ubuntu 22.04 LTS, por su estabilidad, compatibilidad con herramientas de código abierto y amplio soporte comunitario. Este sistema operativo se ha ejecutado en una máquina virtual (VM) creada con Oracle VM VirtualBox, instalada sobre un equipo con sistema operativo nativo Windows 10.

La decisión de utilizar una máquina virtual responde a criterios de flexibilidad y portabilidad: permite mantener el entorno de desarrollo aislado del sistema principal del usuario, replicarlo fácilmente en otros equipos y realizar pruebas sin afectar al sistema operativo anfitrión. Además, simplifica la gestión de dependencias y versiones de software.

Dentro de esta VM se ha instalado el conjunto de herramientas necesarias para implementar el sistema de adquisición y publicación:

- **Python 3.10** como lenguaje de desarrollo, por su simplicidad, legibilidad y la disponibilidad de bibliotecas específicas para el manejo de buses CAN y protocolos de mensajería.
- Biblioteca **python-can**, utilizada para la comunicación con el adaptador Robotell USB-CAN.
- **Mosquitto**, como broker MQTT local para pruebas y como servicio intermedio entre el script de adquisición y la interfaz gráfica.
- **Node-RED**, herramienta de desarrollo visual para construir dashboards que muestren los datos recibidos mediante MQTT.

4.2. Estructura del script de adquisición CAN

El núcleo funcional del sistema desarrollado es un script en Python encargado de leer, interpretar y publicar las tramas CAN extraídas del vehículo a través del adaptador Robotell USB-CAN. Este script ha sido diseñado con un enfoque modular, priorizando la claridad, la eficiencia y la escalabilidad del código para facilitar su mantenimiento y futura ampliación.

4.2.1. Arquitectura general

El script sigue una arquitectura en la que se distinguen claramente tres bloques funcionales:

- **Inicialización del bus CAN**: se establece la conexión con el adaptador Robotell utilizando la biblioteca python-can, configurando el canal, velocidad de comunicación (500 kbps) y los parámetros del puerto serie.

- **Lectura y filtrado de tramas**: el script se mantiene en bucle continuo capturando las tramas CAN que circulan por el bus. Para mejorar el rendimiento y evitar el procesamiento innecesario de datos, se emplea un filtro interno que solo selecciona aquellas tramas cuyos IDs coinciden con los definidos previamente en un diccionario de señales permitidas (SENIALES_PERMITIDAS).

- **Decodificación y publicación**: una vez identificada una trama relevante, se extrae la señal específica mediante operaciones de desplazamiento de bits, máscaras y escalado (si procede). El valor obtenido se empaqueta en formato JSON y se publica a través de MQTT con la biblioteca paho-mqtt.

En la Figura 4.1 se puede ver un diagrama de flujo de la estructura general del script Python desarrollado.

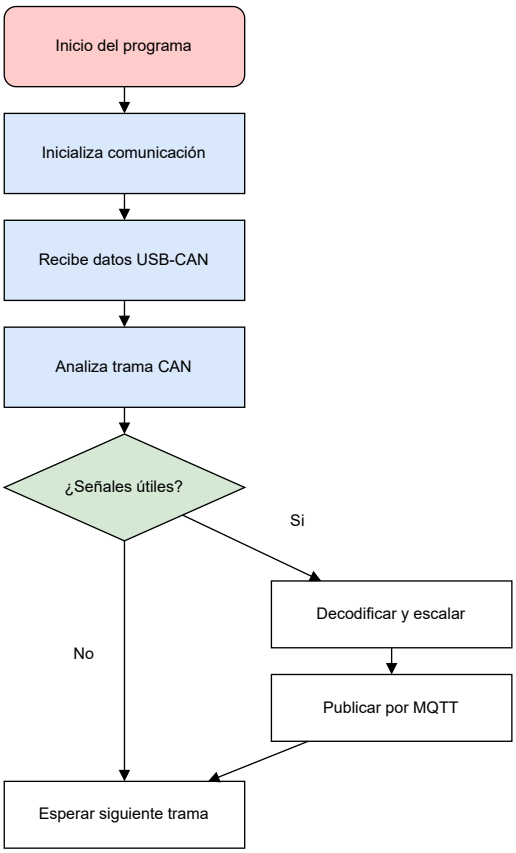

Figura 4.1: Diagrama de flujo del funcionamiento del script general.

4.2.2. Bibliotecas utilizadas en el script

El script desarrollado emplea una serie de bibliotecas estándar y de terceros, cada una con un propósito específico en el procesamiento y transmisión de datos CAN del vehículo:

- **sys**: permite salir del programa cuando se detectan errores de configuración mediante sys.exit().
- **time**: gestiona esperas y temporizaciones (por ejemplo, sleep() en los hilos).
- **serial**: proporciona acceso al puerto serie para la comunicación con el dispositivo USB-CAN.
- **struct**: facilita la conversión entre bytes y tipos de datos (empaquetado/desempaquetado de tramas).
- **logging**: gestiona la salida de mensajes de depuración y advertencias, permitiendo el registro de eventos.
- **argparse**: permite definir y procesar argumentos de línea de comandos como –port o –speed, mejorando la flexibilidad del script.
- **socket** de (de socket): se emplea para crear un socket UDP que permite la recepción remota de comandos desde otra máquina.
- **threading**: utilizada para lanzar procesos concurrentes como el hilo de temporizador y el servidor UDP.
- **json**: permite interpretar los mensajes UDP, que llegan codificados en formato JSON.
- **cantools**: biblioteca especializada para trabajar con archivos DBC y decodificar tramas CAN de acuerdo con su definición.
- **paho.mqtt.client**: permite establecer la conexión con un broker MQTT y publicar las señales CAN procesadas.

4.2.3. Señales CAN seleccionadas para la adquisición

El conjunto de señales disponibles se ha obtenido a partir del análisis del archivo DBC, creado por un tercero, con todas las señales interpretadas del vehículo, debido a que Nissan no proporciona estas bases de datos. EL archivo *CAR-can_AZE0.dbc*, correspondiente al modelo Nissan Leaf 2014. Este archivo describe la estructura de las tramas CAN utilizadas por las distintas unidades de control electrónico del vehículo (ECUs), detallando los identificadores de mensaje (CAN ID), los nombres de las señales contenidas en cada trama, su posición en bits, longitud, tipo de dato, factor de escala y offset, entre otros datos relevantes.

Para explorar y validar las señales contenidas en el DBC, se ha utilizado la biblioteca *cantools* en Python, que permite cargar y decodificar los mensajes según las definiciones del archivo. Una vez

cargado, el script puede interpretar automáticamente los bytes de cada trama en sus correspondientes señales, aplicando los factores de escalado y offset definidos por el creador del archivo DBC. Es por ello que el código Python incluye una función para corregir los fallos de escalado y offset que puedan encontrarse en el archivo DBC al no ser propio del fabricante.

El subconjunto de señales que se ha decidido adquirir y publicar mediante MQTT se ha seleccionado con un criterio orientado a la monitorización del comportamiento de un vehículo autónomo durante la circulación. Las señales elegidas permiten cubrir los siguientes aspectos clave:

- **Cinemática del vehículo**: velocidad real (*VehicleSpeedCluster*), velocidades de cada rueda (*Wheel_Speed_**), distancia recorrida.
- **Actuadores principales**: posición del acelerador (*ThrottlePosition*), presión de freno (*BrakePressure**), fuerza de frenado (*BrakingForce*), sentido de giro del volante (*SteeringAngle*) y tasa de cambio (*SteeringAngleChangeRate*).
- **Estado energético** potencia disponible del motor (*AvailableMotorPower*), consumo (*PowerConsumptMotor_m100_div0p05*), voltaje de batería auxiliar (*LeadAcidBatteryVoltage*), estado de salud del paquete de baterías (*BatteryStateOfHealth*) y barras de carga visibles (*BatteryAvailableChargeBars*).
- **Diagnóstico del vehículo**: código de error en la batería de litio (*LB_Diagnosis_Trouble_Code*, estado general del vehículo (*BCM_VehicleState*) y tiempo estimado de carga completa (*FullChargeTime*).

Este conjunto se ha considerado suficiente para evaluar tanto el estado general del vehículo como el comportamiento de sus sistemas de tracción y frenado, información crítica para tareas de control, registro o diagnóstico de vehículos autónomos.

4.2.3.1. Ampliación futura

El sistema ha sido diseñado de forma modular, por lo que es posible ampliar el número de señales en cualquier momento. Para ello, basta con:

- Añadir el identificador CAN y los nombres de señales deseadas en la estructura *SENIALES_PERMITIDAS*.
- (Opcional) Incluir el factor de escalado y offset correspondiente en el diccionario *ESCALADO*, si se desea ajustar la magnitud a unidades físicas.

Esta capacidad de ampliación permite adaptar el sistema a nuevas necesidades de monitorización, sin necesidad de modificar el núcleo del script de adquisición.

A continuación, en la Tabla 4.1 se detalla el subconjunto de tramas y señales permitidas.

ID CAN	Señal	Unidad	Escalado
0x002	SteeringAngle	grados	0.1
	SteeringAngleChangeRate	-	1
	SteeringSensorHearbeat	-	1
0x176	AscdSpeedRequest	km/h	1
0x260	AvailableMotorPower	kW	1
	PowerConsumptMotor_m100_div0p05	kW	0.05 (offset -100)
0x280	VehicleSpeedCluster	km/h	0.01
0x284	Wheel_Speed_FR	km/h	0.005
	Wheel_Speed_FL	km/h	0.005
	VehicleSpeedFromABS	km/h	0.01
	DistanceTraveled1	-	1
	DistanceTraveled2	-	1
0x285	Wheel_Speed_RR	km/h	0.005
	Wheel_Speed_RL	km/h	0.005
0x292	LeadAcidBatteryVoltage	V	0.1
0x5B3	BatteryPackTemperature	ºC	0.25
	BatteryStateOfHealth	%	1
	FullChargeTime	min	1
	FullChargeTimeMux	-	1
	BatteryAvailableChargeBars	barras (0–12)	1
0x60D	BCM_VehicleState	-	1
0x174	ShifterPosition	-	1
0x1CB	BrakingForce	N	1
0x180	ThrottlePosition	%	0.5
0x1CA	BrakePressure1	-	1
	BrakePressure2	-	1
	BrakePressure3	-	1
	BrakePressure4	-	1
0x5C0	LB_Diagnosis_Trouble_Code	DTC	1

Tabla 4.1: Señales CAN seleccionadas para la adquisición.

4.2.4. Descripción de funciones y bloques lógicos

4.2.4.1. Configuración del cliente MQTT y señales de interés

El script desarrollado en este trabajo comienza inicializando la conexión con el broker MQTT, que será el encargado de recibir las señales CAN ya decodificadas y escaladas. La conexión se establece a través de la biblioteca paho.mqtt.client. En caso de que la conexión falle, se captura la excepción y se notifica el error por consola. Esta función ha sido creada en su totalidad desde cero.

A continuación, se define el diccionario SENIALES_PERMITIDAS, que establece un filtrado sobre las señales CAN que realmente se desean procesar y enviar. Este filtrado permite reducir el volumen de datos y centrarse únicamente en aquellas señales relevantes para el análisis o visualización. Está

estructurado como un diccionario donde las claves son los identificadores (IDs) CAN y los valores son listas con los nombres de señales específicas extraídas de cada trama.

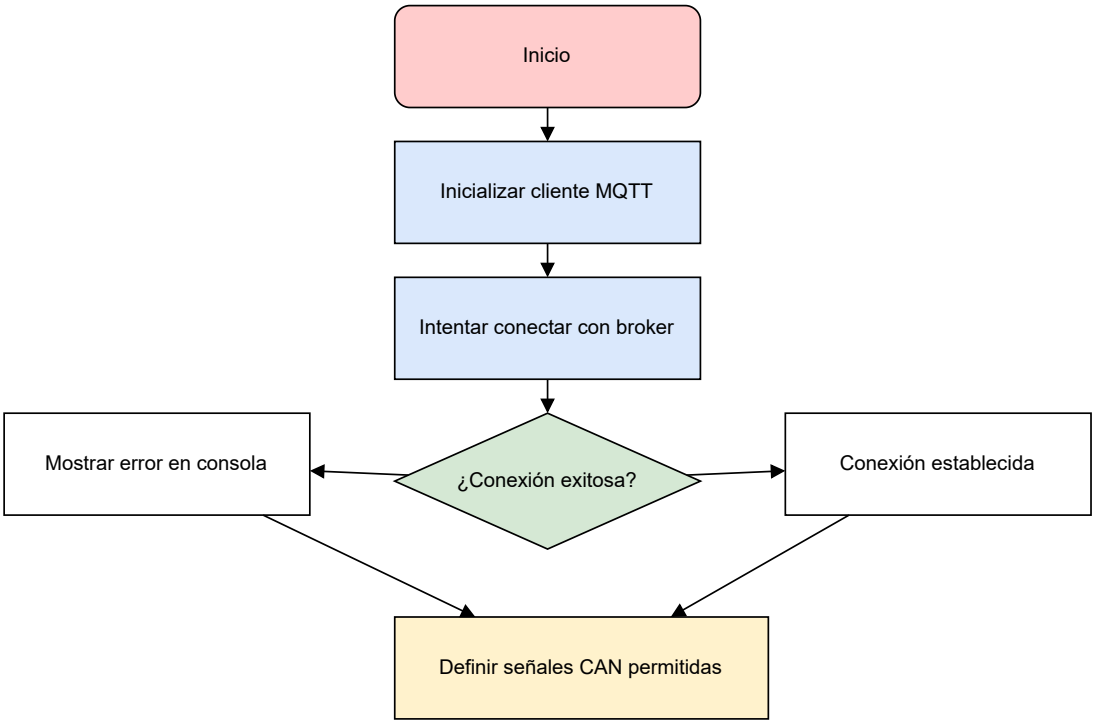

Figura 4.2: Diagrama de flujo del código encargado de la comunicación IoT y el filtrado.

Con este bloque, cuyo diagrama de flujo es el mostrado en la Figura 4.2 se realizan, por tanto, las siguientes funciones:

- Establece el canal de comunicación IoT (MQTT).
- Filtra y selecciona las señales CAN útiles según IDs.
- Permite desacoplar la lógica de adquisición de la lógica de visualización o análisis.

4.2.4.2. Carga del archivo DBC

El archivo DBC es un fichero de base de datos que describe la estructura de los mensajes CAN. En este caso, se utiliza para decodificar las tramas recibidas del vehículo Nissan Leaf. El script utiliza la biblioteca cantools, ampliamente empleada para el análisis de redes CAN.

La carga se realiza sin modo estricto (strict=False) para evitar errores en caso de que el archivo contenga definiciones irregulares o no completamente compatibles. El diagrama de flujo de este proceso se muestra en la Figura 4.3.

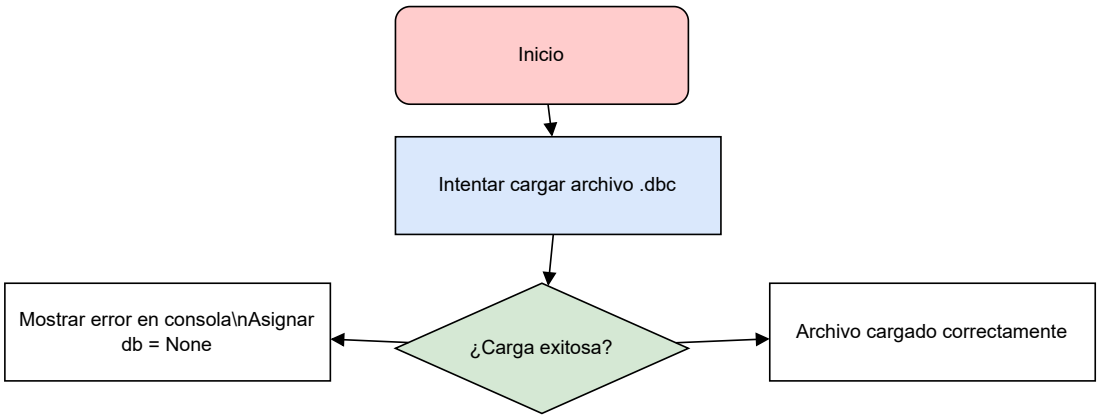

Figura 4.3: Diagrama de flujo del código encargado de la carga del archivo .dbc.

El formato DBC (Database CAN) es un estándar ampliamente utilizado en la industria de la automoción para describir el significado de los mensajes que circulan por el bus CAN de un vehículo. Estos archivos contienen definiciones estructuradas de cada mensaje, incluyendo su ID, las señales que contiene, el rango de bits, tipo de dato, escalado, offset, unidad y condiciones de validez.

El uso de un archivo DBC se justifica dado que, al recibir los mensajes en crudo, únicamente contienen información binaria, por lo que debe ser decodificada para obtener valores comprensibles y útiles.

Un archivo DBC describe cada mensaje como una estructura de datos, con las siguientes secciones:

- **BO_ (message definition)**: define el ID del mensaje, su tamaño y nombre.
- **SG_ (signal definition)**: define el nombre de cada señal, su posición en bits, longitud, tipo (signed/unsigned), factor de escala, offset, unidad y rango válido.

Para utilizar archivos DBC dentro del script Python, se ha empleado la biblioteca cantools, que permite cargar el archivo, decodificar automáticamente los mensajes y extraer los valores de las señales sin necesidad de aplicar manualmente máscaras o desplazamientos. Esta función al igual que la anterior ha tenido que ser implementada ya que el script original de adquisición de datos crudos a través del protocolo CAN no contaba con esta función.

Dado que no todas las señales definidas en el DBC son necesarias para la aplicación desarrollada, se ha implementado un sistema de filtrado que selecciona únicamente aquellas incluidas en el diccionario SENIALES_PERMITIDAS. Esta estrategia permite reducir la carga de procesamiento y enfocar el sistema en señales críticas como velocidad, SOC o temperatura del sistema de baterías.

4.2.4.3. Hilo de temporización (TimerThread)

Este hilo auxiliar permite establecer una bandera (timerFlag) en intervalos de tiempo definidos por el parámetro INTERVAL. El objetivo es que el bucle principal del programa pueda detectar cuándo

ha pasado un cierto tiempo desde el último ciclo, y ejecutar acciones cíclicas si es necesario.

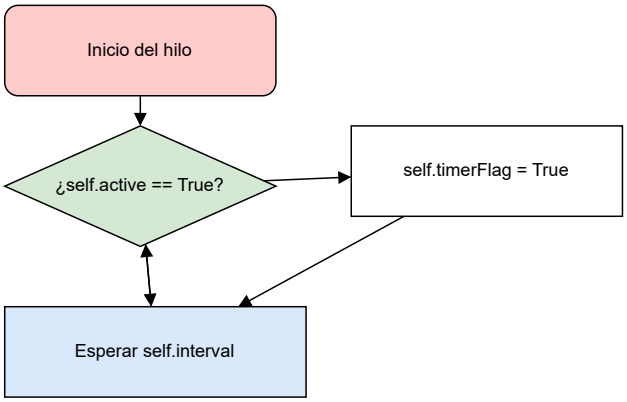

Figura 4.4: Diagrama de flujo del código encargado de ejecutar acciones cíclicas.

El funcionamiento de esta parte del código, cuyo diagrama de flujo se muestra en la Figura 4.4, se basa en ejecutarse en paralelo al resto del programa (es un hilo), activar una bandera cada INTERVAL segundos y permitir que el hilo principal detecte este evento y ejecute las tareas programadas.

4.2.4.4. Hilo receptor de mensajes UDP (ServerThread)

Este hilo paralelo se encarga de escuchar en un puerto UDP para recibir instrucciones externas. Al recibir un paquete, lo interpreta como un mensaje JSON con una estructura determinada. Si el contenido del mensaje es válido, activa la bandera self.interrupt, lo que permite al bucle principal actuar en consecuencia.

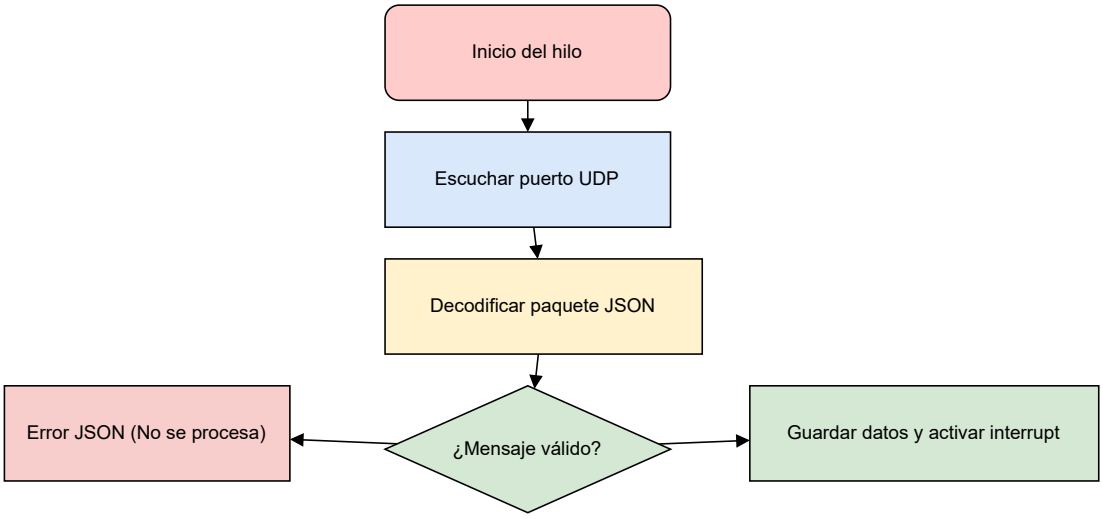

Figura 4.5: Diagrama de flujo del código encargado del hilo receptor de mensajes UDP.

El diagrama de flujo de la Figura 4.5 explica la función de este hilo, el cual escucha continuamente en un puerto UDP, intenta interpretar los mensajes como estructuras JSON válidas y si el mensaje tiene el formato correcto (transmit, filter, etc.) activa una bandera para que el programa principal lo procese.

Esta función no ha sido desarrollada, se trata de una función ya creada para el correcto funcionamiento del adaptador Robotell USB-CAN, y en caso de que no se mencione lo contrario, todas las funciones detalladas en esta sección cumplen esta misma condición.

4.2.4.5. Analizador de tramas serie (ParseClass)

Esta clase actúa como un parser de bajo nivel que recibe bytes secuencialmente y los interpreta según un protocolo específico basado en encabezados (0xAA), control de tramas (0xA5), y terminadores (0x55). Cada cambio en la variable status representa un estado dentro del autómata de parsing. El proceso se inicia al detectar una secuencia de arranque compuesta por dos bytes 0xAA, tras lo cual comienza la acumulación de datos y el cálculo del CRC. Dependiendo del contenido y la longitud de la trama, el parser puede derivar hacia diferentes estados, como la verificación del CRC, el manejo de bytes especiales de control (0xA5) o la validación del final de la trama con dos bytes 0x55. Si en cualquier punto se detecta una anomalía, como un CRC incorrecto o un delimitador inválido, el parser se reinicia automáticamente para evitar interpretar datos corruptos. Cuando una trama completa es reconocida y validada, la función parseData devuelve el buffer con la trama; en caso contrario, devuelve una lista vacía y espera nuevos datos. Su funcionamiento se muestra en el diagrama de flujo de la Figura 4.6.

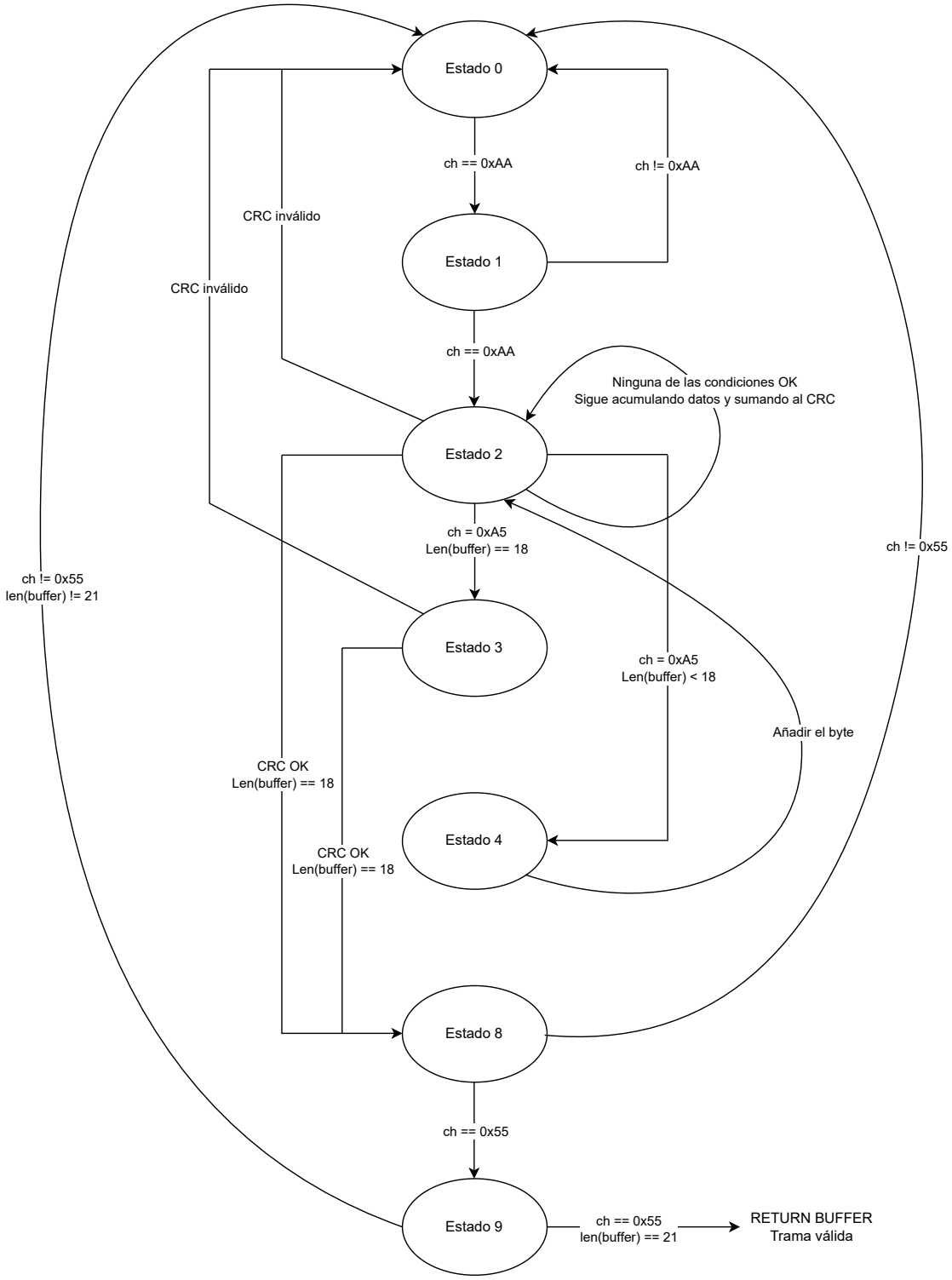

Figura 4.6: Diagrama de flujo del código encargado de analizar las tramas serie (ParseClass).

4.2.4.6. Construcción manual de tramas USART

Esta función se encarga de construir manualmente una trama de datos en formato binario para ser transmitida a través de la interfaz serie, conforme al protocolo definido por el dispositivo USB-CAN utilizado. Aunque no se emplea en la versión final del programa, sirve como ejemplo del formato de trama esperado.

La estructura general de este bloque se basa en la función _setTransmitMsg(id, rtr, ext, len, buf) genera un array de bytes sendData con el siguiente propósito:

- **Encabezado**: los dos primeros bytes (0xAA 0xAA) indican el inicio de la trama.
- **Identificador del mensaje (ID)**: los bytes sendData[2] a sendData[5] representan el identificador del mensaje CAN, formateado como ID estándar (11 bits) o extendido (29 bits), dependiendo del parámetro ext.
- **Datos**: los siguientes bytes (sendData[6] a sendData[13]) son los datos del mensaje (hasta 8 bytes).
- **Longitud del mensaje**: se asigna en sendData[14], especificando el número de bytes válidos en buf.
- **Tipo de trama**: sendData[16] indica si es una trama extendida (ext=1) o estándar (ext=0).
- **Tipo de mensaje**: sendData[17] define si es una trama de datos (rtr=0) o de solicitud remota (rtr=1).
- **Checksum (CRC)**: se calcula como la suma de los bytes desde la posición 2 hasta la 17, y se guarda en sendData[18] como control de integridad.
- **Final**: los bytes finales (0x55 0x55) actúan como delimitador de fin de trama.

Cuyo funcionamiento interno se basa en generar una plantilla base sendData con valores predefinidos. El identificador id se convierte a una cadena hexadecimal de 8 dígitos (idStr), y se trocea para rellenar los campos del identificador CAN:

- Si la trama es **estándar**, solo se usan los dos últimos bytes (idStr[4:8]), con una máscara de 11 bits.
- Si la trama es **extendida**, se usan los 4 bytes completos, ajustados con una máscara de 29 bits.

Por consiguiente, los datos del buffer buf se insertan byte a byte a partir de la posición 6, se calcula la longitud (len), el tipo de frame (ext) y el tipo de mensaje (rtr). Se recalcula el CRC con sum(sendData[2:18]) & 0xff, y se inserta. Finalmente, se imprime el contenido para depuración y se retorna la trama resultante.

Se hace uso de unas constantes auxiliares que forman parte del protocolo propietario de comunicación serie entre el host y el dispositivo USB-CAN. En tramas válidas, los bytes 0xAA marcan el inicio, y los dobles 0x55 indican el cierre.

- USART_FRAMECTRL = 0xA5: byte de control especial (usado para escape o indicación de CRC).
- USART_FRAMECTRL = 0xAA: byte de inicio de trama.
- USART_FRAMECTRL = 0x55: byte de fin de trama.

En la Figura 4.7 se puede observar el diagrama de flujo correspondiente a esta función.

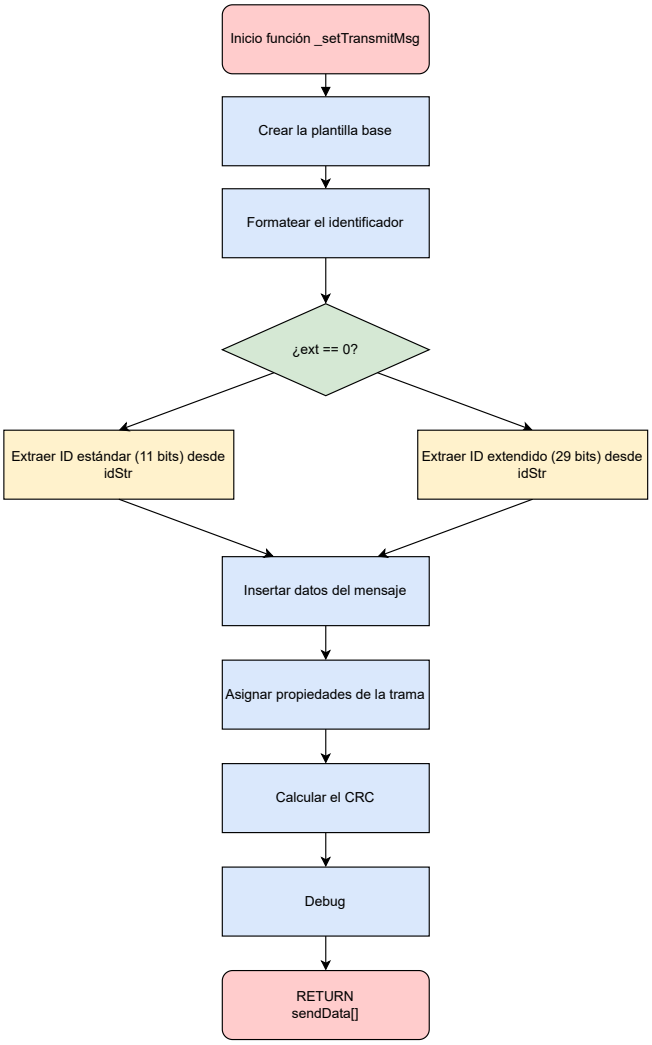

Figura 4.7: Diagrama de flujo de la función _setTransmitMsg.

4.2.4.7. Función insertCtrl()

Esta función implementa un mecanismo de 'escape' para los bytes especiales del protocolo serie. Si el byte ch coincide con alguno de los valores de control (0xA5, 0xAA, 0x55), se inserta una secuencia extra para que el receptor pueda distinguir los datos reales de los bytes reservados del protocolo.

El funcionamiento se basa en la entrada de un buffer (lista de bytes) y un byte ch. Si ch es igual a alguna de las constantes auxiliares (bit de control, inicio de trama o final de trama) se inserta un byte extra 0xA5 antes de ch para indicar que ese byte no es un delimitador real sino un dato, permitiendo al receptor deshacer el 'escape' y recuperar la intención original de la trama. En la Figura 4.8 se puede ver de forma gráfica el funcionamiento de esta función.

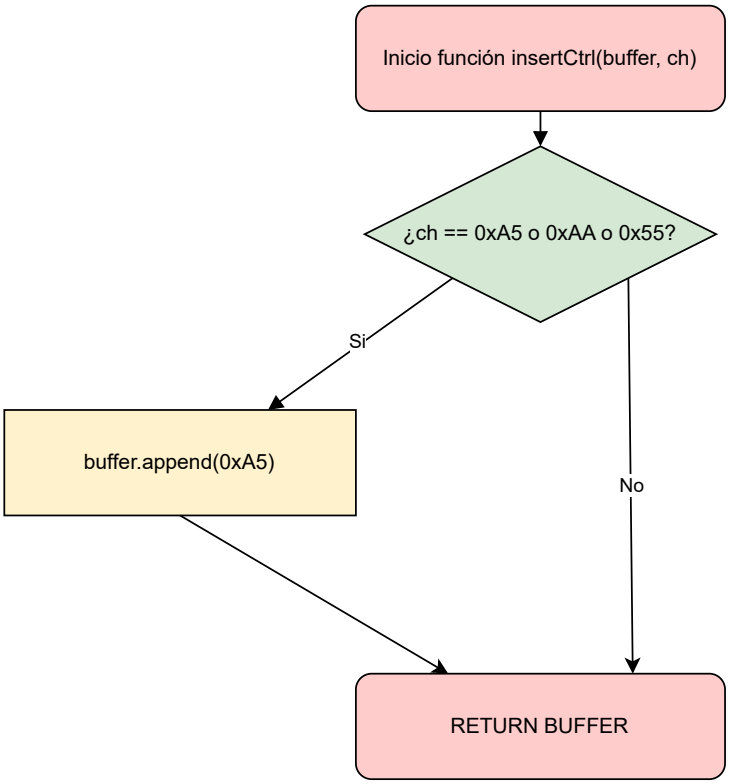

Figura 4.8: Diagrama de flujo de la función insertCtrl.

4.2.4.8. Construcción de trama CAN para configuración de filtros mediante la función setFilterMsg()

Esta función construye una trama CAN de configuración de filtros, que permite al dispositivo USB-CAN recibir solo ciertos mensajes de interés, según ID, máscara y tipo de trama.

Se basa en la validación de parámetros, construye un bloque inicial con bytes fijos, convierte el ID a hexadecimal e inserta los bytes en el orden adecuado, haciendo uso de insertCtrl() para escapar caracteres especiales. Codifica la máscara a hexadecimal byte a byte y configura el tipo de trama, la petición (set) y el bit RTR. Calcula el CRC y se añade, añadiendo además dos bytes 0x55 como delimitadores finales. Para facilitar la comprensión de esta función se hace uso del diagrama de la Figura 4.9.

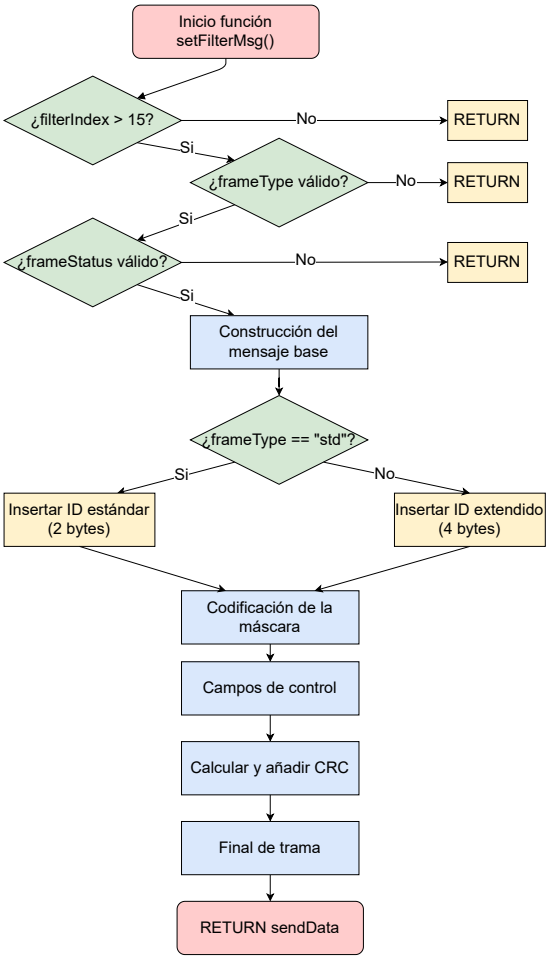

Figura 4.9: Diagrama de flujo de la función setFilterMsg().

4.2.4.9. Construcción de trama CAN completa para transmisión de datos setTransmitMsg()

Esta función se encarga de construir manualmente una trama CAN válida para su envío a través del bus, respetando el protocolo definido por el conversor USB-CAN. A partir de los parámetros de entrada (el identificador del mensaje, los datos a transmitir, y el tipo de trama estándar o extendida,

de datos o remota) se genera un bloque de bytes sendData que incluye encabezados, ID CAN formateado, datos (con padding si es necesario), y campos de control como la longitud, tipo de frame y un CRC calculado. Además, se utiliza la función auxiliar insertCtrl para evitar conflictos con los bytes reservados del protocolo serie, escapándolos si es necesario. Esta función, que se puede ver en el diagrama de flujo de la Figura 4.10, garantiza que la trama construida siga el formato correcto y pueda ser transmitida sin ambigüedad por el bus CAN, con delimitadores claros de inicio y fin (0xAA y 0x55, respectivamente).

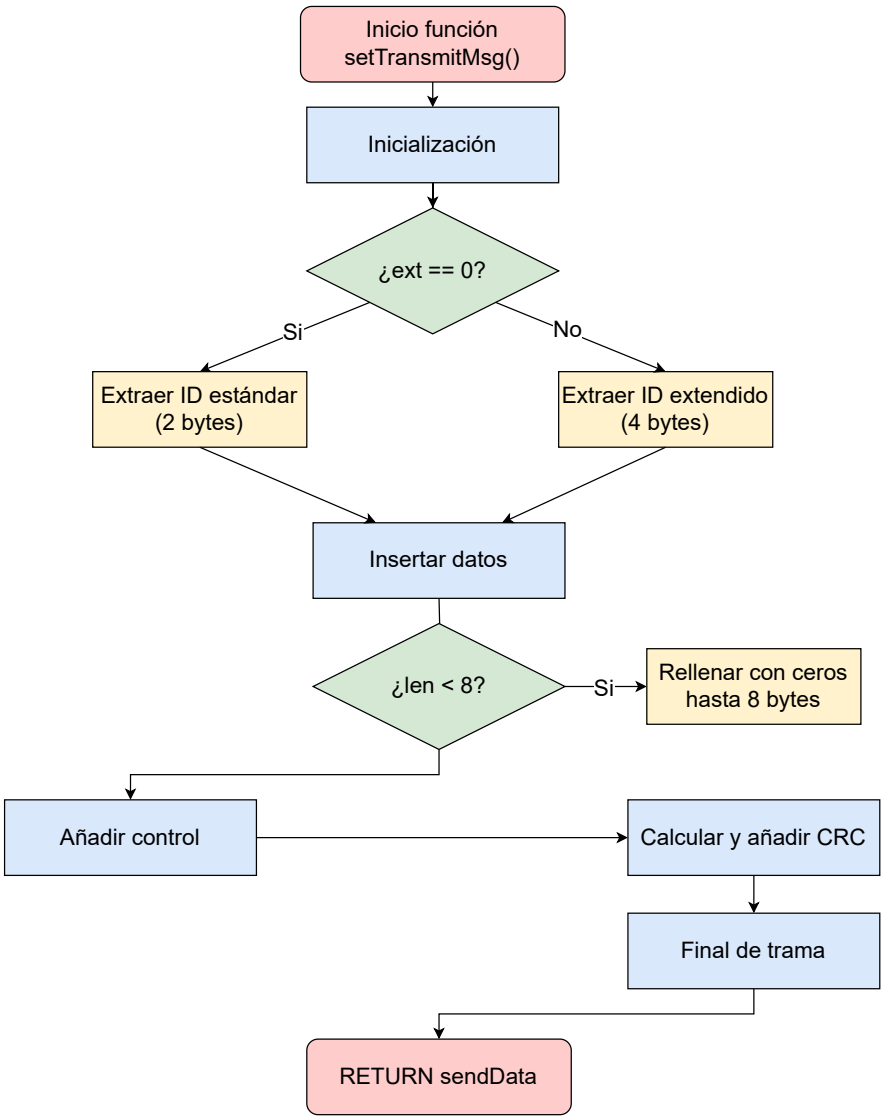

Figura 4.10: Diagrama de flujo de la función encargada de la construcción de la trama CAN completa setTransmitMsg().

4.2.4.10. Envío de trama de datos sendMsg()

La función sendMsg es la encargada de transmitir los mensajes construidos previamente a través del puerto serie. Antes de iniciar la escritura, implementa una espera activa que garantiza que el buffer de salida del puerto se encuentre vacío (ser.out_waiting == 0), evitando así solapamientos en la transmisión. A continuación, recorre secuencialmente el contenido del búfer de entrada buf, empaqueta cada byte utilizando struct.pack("B", b) y lo envía mediante ser.write(). Una vez finalizado el envío, se ejecuta ser.flush() para asegurarse de que todos los bytes han sido despachados completamente desde el búfer del sistema al hardware físico. Esta rutina constituye el mecanismo base de transmisión serial en el sistema propuesto.

4.2.4.11. Inicialización del módulo USB-CAN initId()

La función initId se encarga de generar una trama de inicialización fija que se transmite al dispositivo USB-CAN como parte del arranque o la configuración inicial del sistema. Esta trama contiene un conjunto predefinido de valores en formato byte, incluyendo campos de control e identificación. Antes de ser enviada, se calcula su código de verificación (CRC) sumando los bytes comprendidos entre las posiciones 2 y 17 y aplicando una máscara de 8 bits. Esta suma se almacena en la posición data[18]. Finalmente, la trama completa se transmite usando la función sendMsg, asegurando así que el sistema queda correctamente sincronizado o autenticado antes de iniciar la recepción o transmisión de datos.

4.2.4.12. Construcción de trama de consulta readInfo(id)

La función readInfo permite realizar una consulta a un identificador específico del sistema CAN, construyendo una trama con formato predefinido que se adapta al protocolo del conversor USB-CAN. A partir del parámetro de entrada id, este se convierte a una representación hexadecimal de 8 dígitos, y se insertan sus bytes constituyentes en los campos correspondientes de la trama. Posteriormente, se calcula el CRC para validar la integridad de los datos y se añade en la posición correspondiente. Una vez completada, la trama se envía utilizando sendMsg, iniciando así el proceso de lectura o solicitud de información hacia el dispositivo destino en el bus.

4.2.4.13. Envío de trama de lectura para consulta de configuración de filtro específico readFilter(index)

La función readFilter permite consultar la configuración de un filtro específico dentro del módulo USB-CAN, accediendo a él por su índice. Para ello, se define una trama de consulta con formato fijo, en la que se sobrescribe el campo data[2] con el valor 0xE0 + index, que identifica unívocamente el filtro objetivo. Posteriormente, se calcula el CRC de la trama para garantizar su integridad, sumando los bytes del cuerpo principal y aplicando una máscara de 8 bits. Una vez finalizada la construcción,

la trama se transmite mediante la función sendMsg, lo que provoca que el dispositivo responda con la información del filtro solicitado.

4.2.4.14. Configuración de la velocidad del bus CAN setSpeed()

La función setSpeed permite seleccionar la velocidad de comunicación del bus CAN entre varios valores estándar, como 1 Mbps, 500 kbps, 250 kbps, entre otros. Para cada una de estas velocidades, se define una trama específica de configuración que el módulo USB-CAN reconoce como una instrucción válida para cambiar el bitrate del bus. La función compara el valor del parámetro speed con cada una de las opciones disponibles y, si hay coincidencia, transmite la trama correspondiente mediante sendMsg, retornando True para indicar éxito. En caso contrario, retorna False, señalando que la velocidad solicitada no está soportada.

4.2.4.15. Registro de contenido en trama de bytes logginFrame()

La función loggingFrame tiene como objetivo mostrar de forma legible el contenido de un buffer de bytes, precedido por una etiqueta que indica su contexto, como 'Transmit=' o 'Receive='. Para ello, recorre cada byte del buffer y lo convierte a su representación hexadecimal en mayúsculas, separándolos por espacios y encapsulándolos entre corchetes. Esta cadena resultante se registra en el log mediante logging.info, permitiendo un seguimiento claro y estructurado de las tramas CAN que entran y salen del sistema.

4.2.4.16. Interpretación y decodificación de la trama CAN printFrame()

La función printFrame constituye el núcleo del procesamiento de tramas CAN en el script, esta función ha sido una modificación de una ya existente para poder realizar las funciones que requieren el actual trabajo. Recibe como entrada un buffer con una trama decodificada a bajo nivel y determina si se trata de una trama estándar o extendida, extrayendo correctamente su identificador (receiveId) y su contenido. Si el identificador es uno de los definidos como relevantes en SENIALES_PERMITIDAS y existe un archivo DBC cargado, se decodifican las señales contenidas en la trama. Posteriormente, las señales se escalan según corresponda y se publican individualmente mediante MQTT en sus respectivos tópicos. Además, si se detecta un cambio con respecto a los valores previamente recibidos, se imprime por consola. Este enfoque permite una monitorización eficiente y en tiempo real del estado del vehículo, con un control claro sobre las señales transmitidas y recibidas. Su funcionamiento se explica de forma gráfica mediante el diagrama de flujo de la Figura 4.11.

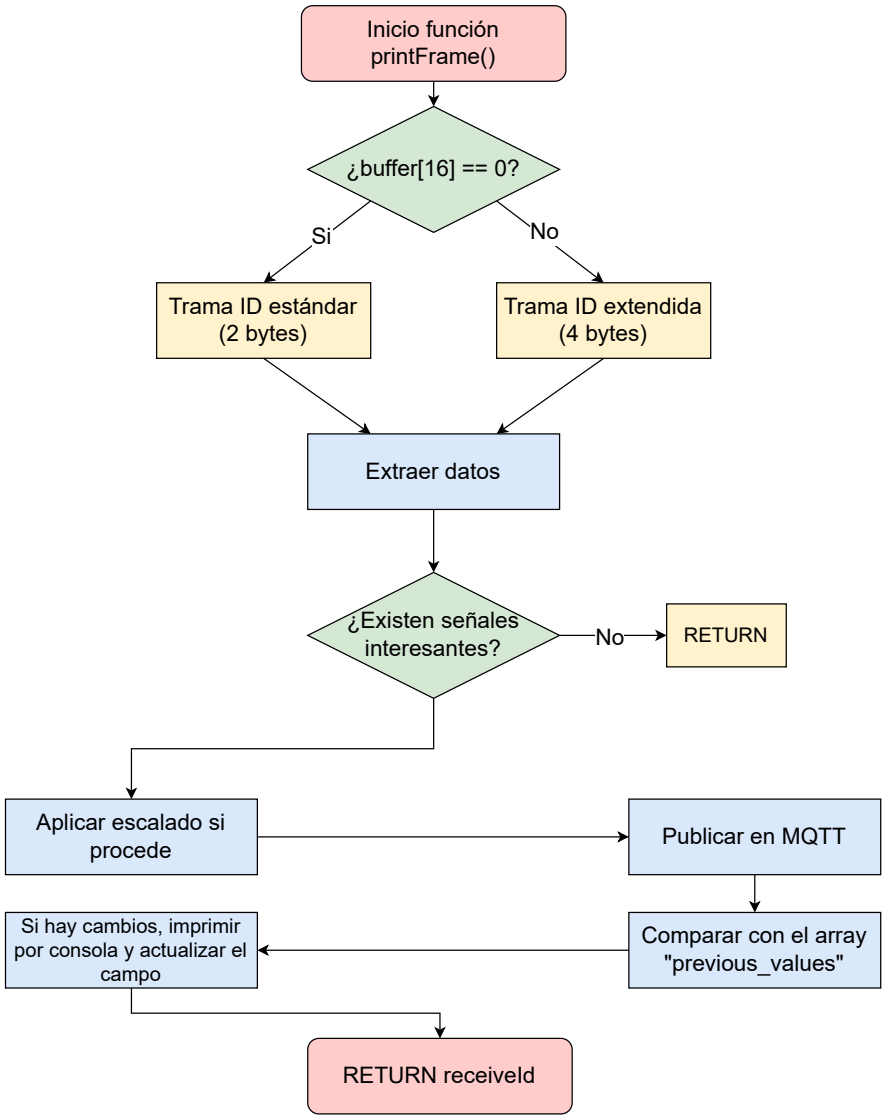

Figura 4.11: Diagrama de flujo de la función printFrame().

El análisis detallado de cada una de las funciones que componen el script de adquisición CAN permite entender el flujo completo de captura, decodificación, tratamiento y envío de señales del vehículo. Desde la configuración inicial del puerto y del broker MQTT, hasta el procesamiento de tramas y la publicación de señales seleccionadas.

4.2.5. Ejemplo de evolución de un mensaje CAN desde formato hexadecimal hasta el dato final

Como ejemplo, se supone que se recibe la siguiente trama CAN:

```
ID: 0x280
Datos (hex): 58 02 00 00 00 00 00 00
```

Este mensaje contiene la señal *VehicleSpeedCluster*, definida en el archivo .dbc y listada en *SENIALES_PERMITIDAS[0x280]*. La señal tiene un factor de escala de 0.01 y offset 0, como aparece en el diccionario *ESCALADO*.

- Paso 1: recepción y decodificación.

  La función *printFrame(buffer)* recoge la trama a través del puerto serie.

  - Extrae el ID (0x280) a partir de los bytes del buffer.
  - Extrae los bytes de datos [0x58, 0x02, ...].
  - Si el ID está en *SENIALES_PERMITIDAS*, se intenta decodificar usando la libreria *cantools*.
    `decoded = db.decode_message(receiveId, data)`
    `decoded = {'VehicleSpeedCluster': 600}`

- Paso 2: escalado.

  La señal *VehicleSpeedCluster* aparece en *ESCALADO* con los parámetros (0.01, 0).

  ```
  escala, offset = ESCALADO['VehicleSpeedCluster']
  valor = decoded['VehicleSpeedCluster'] * escala + offset
  = 600 * 0.01 + 0
  = 6.00 km/h
  ```

- Paso 3: publicación.

  Se construye el *topic* de la siguiente forma.
  `topic = "nissan/leaf/VehicleSpeedCluster"`
  Y se publica en el broker MQTT.
  `mqtt_client.publish(topic, "6.0")`

Este proceso se realiza automáticamente para todas las señales presentes en SENIALES_PERMITIDAS, siempre que el archivo .dbc esté bien cargado y el ID recibido corresponda con una definición válida.

CAPÍTULO 5

Sistema a bordo

Contenido

Este capítulo presenta la arquitectura y configuración del sistema a bordo encargado de transmitir la telemetría capturada del vehículo hacia la nube.

5.1. Interfaz de visualización en Node-RED: configuración y lógica interna

Se ha desarrollado una interfaz haciendo uso de Node-RED, una herramienta de desarrollo basada en flujos visuales, diseñada para facilitar la integración de dispositivos, servicios y datos mediante una interfaz gráfica intuitiva. Gracias a su estructura de nodos conectables y su compatibilidad con protocolos como MQTT, resulta especialmente útil en entornos IoT donde se requiere procesar y visualizar datos en tiempo real. En este sistema, Node-RED se ha utilizado como plataforma para representar gráficamente la telemetría extraída del vehículo eléctrico, permitiendo tanto su monitorización como su análisis inmediato desde cualquier navegador. Esta sección describe cómo se ha configurado la interfaz de usuario, detalla los bloques funcionales empleados y explica la lógica que sigue cada flujo para transformar los datos CAN en información útil y accesible.

5.1.1. Estructura general de flujo

La arquitectura lógica del sistema en Node-RED sigue un esquema simple pero robusto de tres etapas: entrada, procesamiento y visualización. Este enfoque modular permite escalar el flujo con facilidad a medida que se añaden nuevas señales CAN o se requieren transformaciones adicionales.

- **Entrada**: la etapa de entrada está compuesta por nodos *MQTT in*, que se suscriben al tópico jerárquico *nissan/leaf/(señal en concreto)*. Esto permite capturar todas las señales publicadas por el sistema de adquisición sin necesidad de configurar cada una por separado. Cada mensaje recibido contiene una única señal identificada por su nombre en el *topic* y su valor como carga útil.

- **Procesamiento**: tras la recepción, los mensajes se dirigen a nodos de tipo *function*, que permiten aplicar transformaciones si es necesario (ej: aplicar un umbral, cambiar el formato, etc.) y filtrar mensajes según la señal.

- **Visualización**: finalmente, los datos son enviados a nodos de *dashboard* (ui_text, ui_gauge, ui_chart, etc.), que construyen en tiempo real la interfaz gráfica accesible vía navegador. Esta visualización permite al usuario ver el estado del vehículo eléctrico en todo momento: velocidad, temperatura de batería, presión de freno, posición del cambio, entre otros..

Para organizar la visualización, se han estructurado diferentes secciones, cada una dedicada a un aspecto del vehículo:

- **Estado del vehículo:** se muestran variables como el estado general (*BCM_VehicleState*), la posición de la palanca de cambios (*ShifterPosition*) y la presencia de códigos de error en la batería (*LB_Diagnosis_Trouble_Code*). Véase la Figura 5.1.

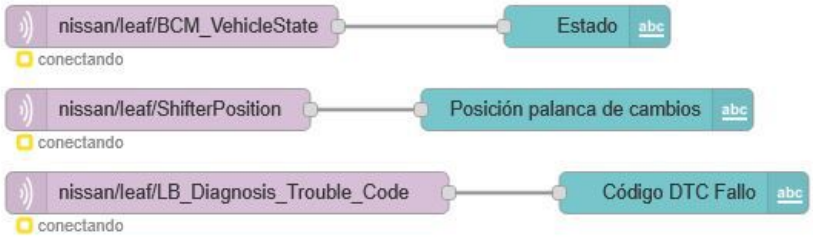

Figura 5.1: Diagrama de flujo Node-RED de la sección dedicada al estado del vehículo.

- **Control del volante:** incluye la visualización del ángulo del volante (*SteeringAngle*), su velocidad de cambio (*SteeringAngleChangeRate*) y el latido del sensor como señal de diagnóstico (*SteeringSensorHearbeat*). Véase la Figura 5.2.

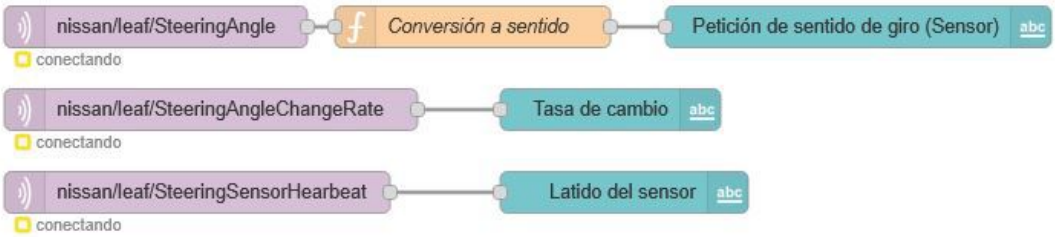

Figura 5.2: Diagrama de flujo Node-RED de la sección dedicada al control del movimiento del volante.

- **Estado de las baterías:** se representan variables energéticas como el tiempo estimado de carga (*FullChargeTime*), el estado de salud del pack de baterías, la temperatura del conjunto, el voltaje de la batería auxiliar y el número de barras de carga visibles. Véase la Figura 5.3.

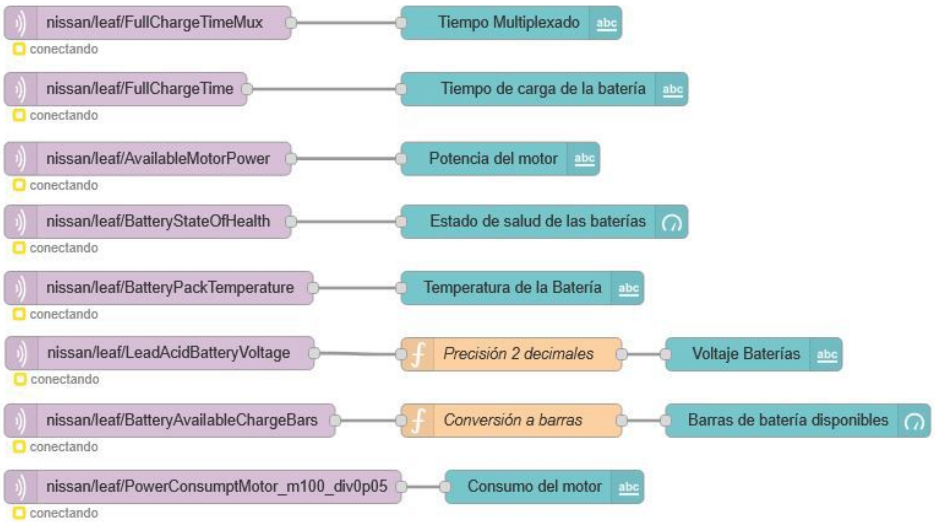

Figura 5.3: Diagrama de flujo Node-RED de la sección dedicada al estado de las baterias del vehículo.

- **Movimiento del vehículo:** se incluyen señales que reflejan la cinemática y la interacción con los actuadores: velocidad solicitada, velocidad real (general y por ruedas), fuerza de frenado, presiones de freno, posición del acelerador y distancia recorrida. Véase la Figura 5.4.

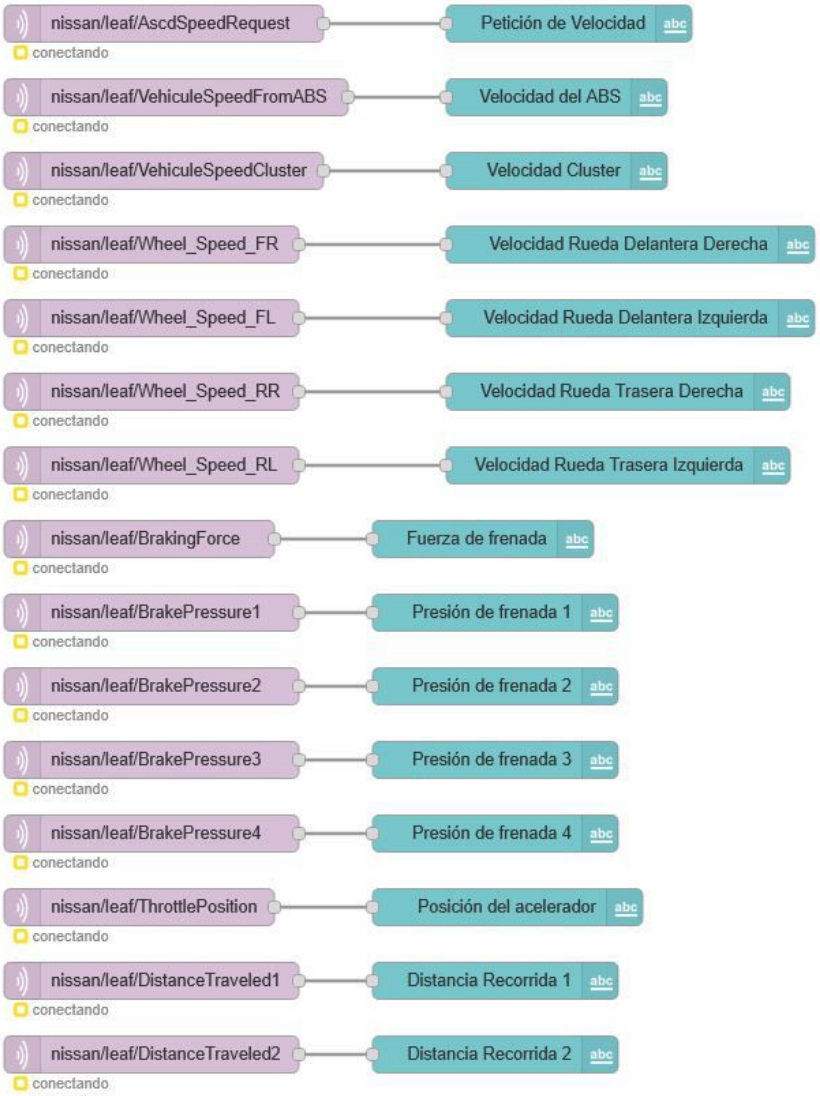

Figura 5.4: Diagrama de flujo Node-RED de la sección dedicada al movimiento del vehículo.

5.1.2. Lógica de los nodos function aplicados a las señales

Dentro del flujo de procesamiento de datos en Node-RED, los nodos function permiten escribir funciones personalizadas en JavaScript, adaptando el comportamiento de cada señal según sus características o requerimientos. Esta capa de inteligencia intermedia permite asegurar que los datos mostrados en la interfaz gráfica sean útiles para su interpretación en tiempo real.

5.1.2.1. Función conversión a sentido

Uno de los nodos function implementados en Node-RED se encarga de determinar la dirección de giro del volante a partir del análisis de su valor actual y el inmediatamente anterior. Este bloque compara ambos valores y, si la diferencia supera un umbral determinado, clasifica el movimiento como 'Derecha', 'Izquierda' o 'Quieto'. El uso de una memoria de contexto (context) permite conservar el valor anterior entre ejecuciones, y el umbral actúa como filtro frente al ruido. A continuación se muestra el diagrama de flujo en la Figura 5.5 que representa esta lógica de decisión.

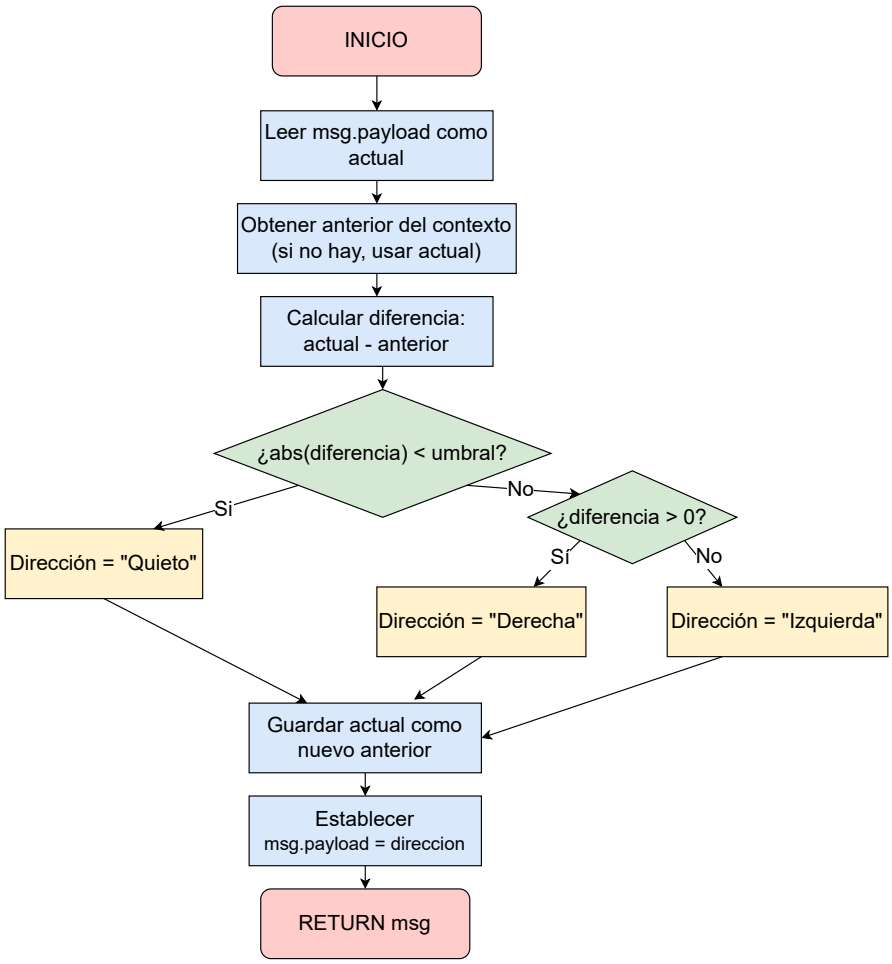

Figura 5.5: Diagrama de flujo de la funcion implementada en Node-RED para la estimación del sentido de giro del volante.

5.1.2.2. Función de extracción de los 4 bits menos significativos para muestreo de las barras de batería disponibles

Esta función extrae únicamente los 4 bits menos significativos (bits 0 a 3) de un valor recibido en msg.payload, que llega como un número entero sin procesar. Esto se realiza mediante una operación lógica AND con la máscara 0x0F (equivalente a 00001111 en binario), permitiendo obtener el número de barras disponibles en el indicador de carga de la batería. Esta operación es común en protocolos CAN, donde varias señales están empaquetadas en un único byte. La lógica se explica gráficamente mediante el diagrama de la Figura 5.6.

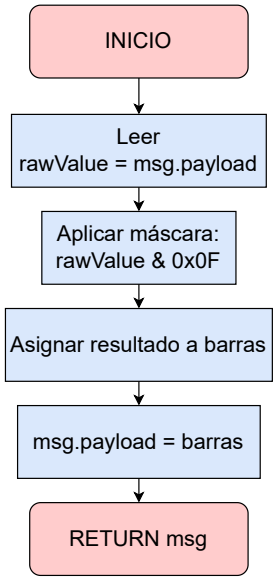

Figura 5.6: Diagrama de flujo de la funcion implementada en Node-RED para la conversión a barras de batería.

5.1.2.3. Función de conversión a dos decimales para la señal de voltaje de baterias auxiliares (Lead Acid)

Otra de las funciones implementadas en los nodos function tiene como objetivo ajustar el formato de los valores numéricos antes de su visualización. En este caso, se utiliza la conversión explícita a tipo Number y el método .toFixed(2) para redondear cada valor a dos cifras decimales.

5.1.3. Representación visual de señales

Con el objetivo de facilitar la interpretación en tiempo real de las señales CAN adquiridas, se ha implementado un panel gráfico en Node-RED organizado por bloques temáticos. En este panel se utilizan diferentes tipos de widgets de la librería node-red-dashboard, tales como:

- **Indicadores visuales**: empleados para representar variables con escalado intuitivo y valor numérico centrado.
- **Elementos de texto estático y dinámico**: utilizados para visualizar directamente valores o etiquetas textuales.

El panel de visualización mostrado en la Figura 5.7 representa la interfaz gráfica final diseñada en Node-RED para la monitorización del Nissan Leaf. Esta interfaz recoge y organiza de forma estructurada las señales CAN previamente seleccionadas, agrupadas por bloques funcionales: estado del vehículo, baterías, ángulo del volante y movimiento.

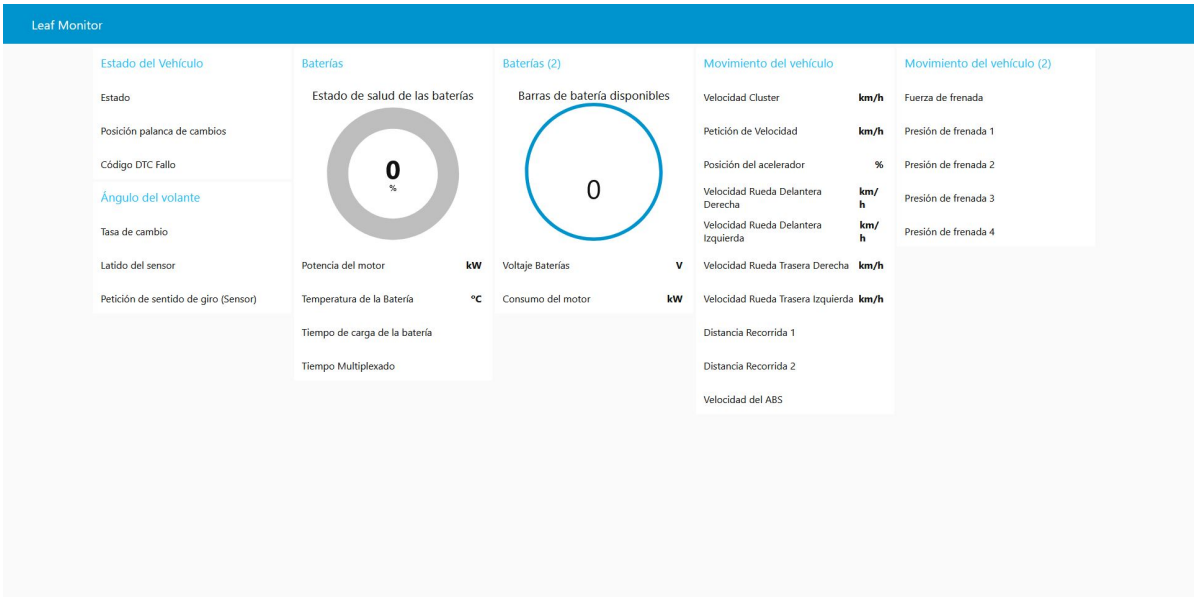

Figura 5.7: Dashboard implementado en Node-RED.

Cada bloque permite supervisar, en tiempo real, el comportamiento de subsistemas específicos. Por ejemplo, en la sección “Movimiento del vehículo” se observa cómo se representan señales críticas para el análisis dinámico, como la velocidad por cada rueda, la posición del acelerador y la fuerza de frenado. Estas variables son clave para evaluar el comportamiento de un vehículo autónomo durante la circulación.

En el bloque “Baterías”, se han incorporado indicadores visuales circulares (gauges), como el que muestra el estado de salud del pack de baterías y el número de barras de carga disponibles.

Estos indicadores permiten identificar rápidamente el nivel de carga o el estado de los componentes eléctricos del vehículo.

Asimismo, en la parte dedicada al "Estado del vehículo", se reflejan valores como la posición de la palanca de cambios o los códigos de diagnóstico (DTC), mientras que el bloque "Ángulo del volante" incluye tanto el ángulo como su tasa de cambio, permitiendo observar maniobras en tiempo real.

Cada uno de estos datos procede del sistema de adquisición, es procesado por Node-RED tras su recepción vía MQTT, y finalmente representado mediante nodos visuales interactivos. Este flujo garantiza que cada señal recibida en bruto pueda ser entendida y supervisada por el usuario final de forma accesible.

El recorrido típico de una señal desde que es publicada en el broker MQTT hasta su representación en la interfaz gráfica se estructura en tres etapas consecutivas dentro del entorno Node-RED. Este flujo, representado en la Figura 5.8, ilustra cómo se gestionan los datos de telemetría en tiempo real:

- **Inicio del mensaje MQTT:** el proceso comienza cuando una señal es publicada en el broker MQTT bajo un tópico específico del tipo *nissan/leaf/(nombre_señal)*. Esta publicación se realiza automáticamente por el sistema de adquisición que lee y decodifica las tramas CAN.
- **Nodo *MQTT IN*:** Node-RED dispone de nodos suscritos a dichos tópicos. Al recibir un mensaje, este nodo actúa como punto de entrada del flujo, almacenando el valor de la señal y asociándolo con su tópico de origen.
- **Nodo *function* (si aplica):** en esta etapa intermedia opcional, es posible aplicar transformaciones al dato recibido. Por ejemplo, se pueden ajustar escalas, establecer etiquetas semánticas o filtrar señales específicas si se requiere una lógica condicional.
- **Visualización:** finalmente, el valor procesado se envía a un nodo de interfaz gráfica. Dependiendo de la naturaleza de la señal, puede representarse mediante un *gauge* (indicador circular) si es un valor continuo o de referencia rápida, o mediante nodos *text* si se desea mostrar el valor numérico de forma directa.

Este esquema modular permite una clara separación entre la adquisición, el tratamiento de datos y la visualización, lo cual facilita tanto la escalabilidad del sistema como su mantenimiento. Además, al estar basado en tópicos jerárquicos, resulta muy sencillo añadir nuevas señales a la interfaz simplemente replicando este flujo para cada una de ellas.

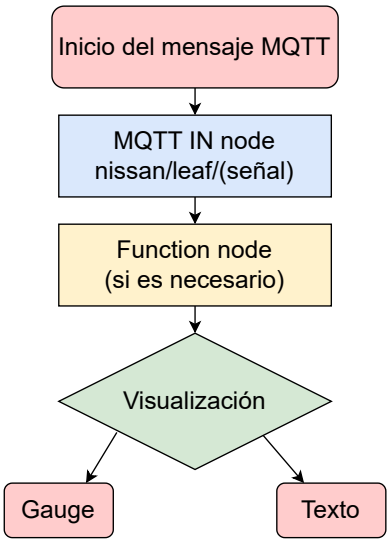

Figura 5.8: Diagrama de flujo del proceso completo hasta UI Dashboard en Node-RED.

5.1.4. Configuración de acceso remoto a Node-RED

Para permitir que el panel de control generado con Node-RED fuese accesible desde otros dispositivos (como un smartphone conectado a Internet mediante la red móvil), ha sido necesario realizar modificaciones en el archivo de configuración *settings.js*, ubicado en el directorio *C:User/(Usuario)/.node-red/*. Concretamente en la línea que menciona por defecto:

```
//uiHost: "127.0.0.1",
```

Y cambiándola por:

```
uiHost: "0.0.0.0",
```

Por defecto, Node-RED solo escucha en 127.0.0.1 (localhost), lo cual impide que otros equipos accedan al dashboard. Al establecer 0.0.0.0, Node-RED acepta conexiones desde cualquier interfaz de red, permitiendo así su exposición a través de Internet, tras abrir el puerto correspondiente en el router y en el firewall del sistema operativo. (También es válido poner la dirección IP fija del host si está abierto el puerto 1880 en esa IP en la configuración del router).

5.2. Broker Mosquitto: instalación y uso

El protocolo MQTT (Message Queuing Telemetry Transport) es un estándar de mensajería ligera ampliamente utilizado en entornos IoT debido a su bajo consumo de recursos, simplicidad de implementación y eficiencia en comunicaciones máquina a máquina (M2M). Diseñado inicialmente por IBM en 1999 para monitorizar oleoductos vía satélite, su arquitectura basada en el modelo publicador–suscriptor permite desacoplar emisores y receptores de mensajes, facilitando sistemas escalables y tolerantes a fallos [15].

5.2.1. Arquitectura y funcionamiento

El modelo de funcionamiento de MQTT se basa en tres elementos principales:

- **Publicador (publisher)**: entidad que envía datos a un tema (topic) específico.
- **Suscriptor (subscriber)**: entidad que recibe los datos publicados en uno o varios temas.
- **Broker**: servidor intermedio encargado de recibir los mensajes de los publicadores y reenviarlos a los suscriptores correspondientes.

Este modelo desacoplado permite que el vehículo envíe los datos sin conocer el destino final, mientras que los sistemas de visualización o almacenamiento solo necesitan suscribirse al tema adecuado.

El protocolo trabaja sobre TCP/IP y define distintos niveles de calidad de servicio (QoS) que controlan la garantía de entrega de los mensajes (desde “al menos una vez” hasta “exactamente una vez”), lo que permite adaptarse a diferentes requerimientos de fiabilidad según la aplicación.

En este proyecto se utiliza Eclipse Mosquitto, un broker MQTT de código abierto ampliamente adoptado por la comunidad y con soporte completo al estándar. Mosquitto permite una configuración sencilla y es compatible con herramientas como Node-RED, facilitando el despliegue de sistemas IoT completos en dispositivos embebidos o entornos cloud.

La elección de MQTT como protocolo de mensajería responde a la necesidad de un canal de comunicación ligero, eficiente y fácil de integrar en sistemas heterogéneos, logrando una arquitectura de vehículo conectado flexible, escalable y funcional en tiempo real.

5.2.2. Instalación en sistema Windows

A diferencia de entornos típicos en Linux, en este proyecto el broker Mosquitto se ha instalado directamente en el sistema operativo nativo del desarrollador (Windows 10). Esta configuración permite mantener separadas las funciones de adquisición (realizadas desde una máquina virtual Linux) y de distribución y visualización (desde el host Windows), comunicándose entre sí a través de red local o, cuando se requiere acceso desde el exterior, mediante la interfaz de red 5G.

El proceso de instalación del broker incluyó los siguientes pasos:

- Descarga del instalador oficial desde *mosquitto.org*.
- Instalación del paquete que incluye los ejecutables *mosquitto.exe* y *mosquitto_sub.exe*.
- Configuración del archivo *mosquitto.conf*, habilitando la recepción de mensajes externos con los siguientes parámetros:

```
listener 1883
allow_anonymous true
```

- Apertura del puerto TCP 1883 en el firewall de Windows para permitir conexiones desde dispositivos externos.

5.2.2.1. Configuración de red para acceso remoto

Para habilitar el acceso remoto desde Internet, el sistema se ha diseñado de forma que la máquina virtual Linux (encargada de la adquisición de datos CAN y publicación MQTT) y el sistema Windows (donde residen Mosquitto y Node-RED) pueden encontrarse en redes separadas o incluso en ubicaciones diferentes.

La configuración de red realizada es la siguiente:

- El host Windows dispone de una dirección IP local fija: `195.0.1.60`, asignada por el router 5G mediante reserva DHCP.
- La dirección IP pública del router 5G que proporciona acceso desde Internet es `81.60.220.151`.
- Se han redirigido los puertos necesarios desde el router al host Windows:
  - **1883** → para permitir conexiones MQTT al broker Mosquitto.
  - **1880** → para exponer la interfaz web de Node-RED al exterior.

Gracias a esta configuración, cualquier equipo remoto conectado a Internet puede acceder al sistema de telemetría:

- Visualización de la interfaz gráfica: `http://81.60.220.151:1880/ui`

Este diseño distribuido permite separar físicamente la adquisición y la monitorización del vehículo, una característica fundamental en entornos de pruebas con vehículos autónomos, donde la adquisición puede realizarse a bordo y la visualización desde un puesto de control remoto.

En caso de que la dirección IP pública cambiara debido a una reconexión del router o a las políticas del operador, sería recomendable incorporar un servicio de DNS dinámico (como DuckDNS o No-IP) para mantener un dominio persistente que apunte siempre a la IP actual.

5.2.3. Pruebas y validación

Tras iniciar el servicio Mosquitto, se realizaron pruebas de conectividad desde la máquina virtual Ubuntu (host de adquisición) utilizando los siguientes comandos:

- Suscripción desde Ubuntu:
  `mosquitto_sub -h 195.0.1.60 -t test/topic`
- Publicación desde el script Python:
  `client.connect("195.0.1.60", 1883, 60)`
  `client.publish("telemetria/nissan_leaf/VehicleSpeedCluster")`

Estas pruebas confirmaron la correcta comunicación entre el host Linux (cliente publicador) y el broker Mosquitto ejecutándose en el entorno Windows.

5.3. Flujo de datos en la nube (publisher/subscriber)

El sistema de telemetría desarrollado se basa en un modelo de comunicación desacoplada entre emisores y receptores de información, siguiendo la arquitectura publisher/subscriber del protocolo MQTT. Este enfoque permite que los distintos elementos del sistema funcionen de forma independiente, facilitando la escalabilidad, modularidad y tolerancia a fallos.

5.3.1. Recorrido de la información

El flujo de datos en el sistema sigue la siguiente secuencia:

- **Adquisición en el vehículo**: el script Python, ejecutado en el host Linux conectado al adaptador USB-CAN, intercepta en tiempo real las tramas CAN que circulan por el bus del Nissan Leaf.
- **Decodificación de señales**: a partir de las tramas recibidas, se extraen las señales definidas como relevantes (por ejemplo, VehicleSpeedCluster, BatteryPackTemperature, SOC, etc.) aplicando los correspondientes factores de escalado y offset, definidos manualmente o a través de un archivo DBC.
- **Publicación MQTT**: cada señal decodificada se empaqueta en un mensaje JSON y se publica mediante el cliente paho-mqtt en un topic específico del broker, ubicado en el sistema nativo Windows 10 con IP fija 195.0.1.60.
- **Distribución desde el broker Mosquitto**: el broker recibe los mensajes publicados y los reenvía automáticamente a todos los clientes suscritos a los temas correspondientes. Esta operación se realiza con baja latencia y sin necesidad de que el publicador conozca la identidad o cantidad de suscriptores.

- **Visualización en Node-RED**: la interfaz gráfica se suscribe a los topics de interés mediante nodos MQTT y representa los datos en tiempo real mediante instrumentos visuales como indicadores, gráficos o barras. Esto permite al usuario monitorizar el estado del vehículo desde cualquier dispositivo conectado a la misma red.

5.3.2. Ventajas del modelo publisher/subscriber

Existe un desacoplamiento total entre componentes: el publicador no necesita conocer a los suscriptores, lo que permite escalar el sistema sin modificar su lógica interna. Se tiene soporte para múltiples suscriptores simultáneos, que pueden recibir los mismos datos en paralelo para propósitos distintos (visualización, almacenamiento, análisis...). Además cuenta con una eficiencia y ligereza ideal para entornos con conectividad móvil o recursos limitados.

Este flujo de datos representa el núcleo funcional del sistema de telemetría BusCAN, y ha demostrado una alta fiabilidad durante las pruebas, permitiendo la adquisición y visualización remota de señales del vehículo en tiempo real, incluso en condiciones de movilidad.

5.4. Topología de red del sistema final

El sistema diseñado para la adquisición y visualización de datos del vehículo eléctrico se apoya en una topología en estrella distribuida, donde el nodo central es el broker MQTT, encargado de gestionar la comunicación entre todos los elementos, que se encuentran en redes distintas y se conectan a través de Internet utilizando una IP pública.

Los elementos principales del sistema son los siguientes:

- **Dispositivo de adquisición a bordo:** una máquina virtual con Ubuntu que actúa como nodo *publisher*. Ejecuta el script Python que lee señales del vehículo mediante un adaptador USB-CAN y publica los datos procesados (decodificados y escalados) en distintos tópicos del broker MQTT. Este dispositivo se encuentra en una red separada del broker, y se comunica con él a través de la IP pública.

- **Broker MQTT (Mosquitto):** instalado en el sistema operativo Windows del host físico, accesible desde el exterior mediante la dirección IP pública `81.60.220.151` y el puerto estándar `1883`. El broker recibe los mensajes publicados por el sistema de adquisición y los redistribuye a todos los clientes suscritos. La redirección de puertos en el router 5G permite exponer este servicio a través de Internet.

- **Interfaz de usuario Node-RED:** se ejecuta en la misma máquina que el broker MQTT (host Windows) y actúa como *subscriber*. Escucha los tópicos definidos y actualiza en tiempo real los indicadores del panel gráfico accesible vía navegador. Esta interfaz puede consultarse desde cualquier dispositivo remoto a través de la URL pública `http://81.60.220.151:1880/ui`.

- **Clientes adicionales (opcional):** cualquier dispositivo autorizado, incluso fuera de la red local, puede suscribirse a los tópicos del broker y visualizar los datos, siempre que se configure adecuadamente el acceso remoto y la seguridad. Estos clientes pueden usarse para monitoreo, almacenamiento externo o análisis avanzado.

La topología general del sistema se muestra de forma esquemática en la Figura 5.9.

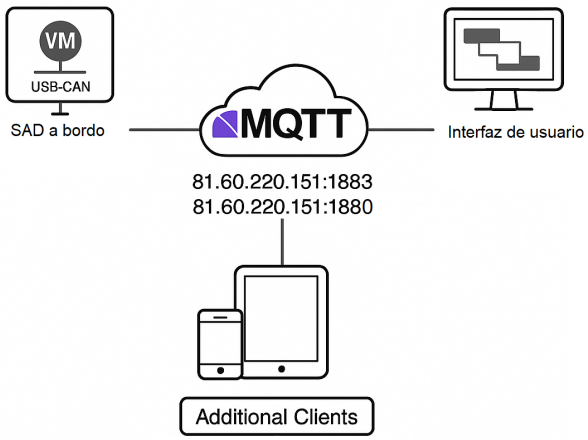

Figura 5.9: Esquemático de topología de red del sistema.

CAPÍTULO 6

Validación y resultados

Contenido

Este capítulo recoge el proceso de validación del sistema diseñado, así como una muestra representativa de los resultados obtenidos durante las pruebas realizadas sobre el vehículo Nissan Leaf. Se presentan las señales que se han conseguido extraer correctamente, su visualización en tiempo real mediante la interfaz de Node-RED y una recopilación de escenarios de uso típicos donde se evalúa el comportamiento del sistema.

6.1. Puesta en marcha del sistema

Para llevar a cabo una sesión de adquisición y visualización de datos del vehículo, se requiere la inicialización ordenada de los distintos componentes del sistema, los cuales se encuentran distribuidos entre una máquina virtual (Ubuntu) y el sistema operativo host (Windows). A continuación se detalla el proceso paso a paso:

1. **Conexión del lector USB-CAN al host Windows:** se debe conectar físicamente el adaptador USB-CAN al equipo. Este adaptador será gestionado por la máquina virtual Ubuntu mediante la configuración de redirección de dispositivos USB en el hipervisor (por ejemplo, VirtualBox o VMware).

2. **Arranque de la máquina virtual Ubuntu:** se inicia la VM configurada previamente con las herramientas necesarias para la adquisición, incluyendo Python, el archivo `.dbc`, y el script de lectura y publicación de señales CAN.

3. **Ejecución del script de adquisición:** una vez iniciada la VM, se accede al directorio del proyecto y se lanza el script principal mediante terminal. El comando típico es:

   ```
   python3 CAN_FILTER.py -p /dev/ttyUSB0 -s 500000
   ```

   Esto activa la lectura de tramas CAN y la publicación de las señales decodificadas a través del broker MQTT.

4. **Inicio del broker Mosquitto en Windows:** el broker MQTT debe estar configurado para aceptar conexiones en el puerto 1883. Puede ejecutarse como servicio o lanzarse manualmente desde consola, asegurándose de que el archivo `mosquitto.conf` permite conexiones externas si se desea.

5. **Ejecución de Node-RED:** en el sistema Windows se inicia Node-RED, que cargará automáticamente el flujo definido para visualizar las señales. La interfaz puede abrirse en el navegador accediendo a:

   ```
   http://(IP_PUBLICA):1880/ui
   ```

6. **Verificación del flujo de datos:** una vez ejecutado el script y con el vehículo encendido, las señales decodificadas comenzarán a mostrarse tanto en consola (terminal de Ubuntu) como en la interfaz gráfica de Node-RED. En esta etapa puede comprobarse que los datos llegan correctamente y se actualizan en tiempo real.

Este procedimiento puede repetirse para cada nueva sesión de pruebas, y permite validar tanto la conectividad entre los módulos del sistema como el funcionamiento conjunto del software desarrollado.

6.2. Validación en entorno real y observaciones en pista

Para validar el sistema de adquisición y visualización de datos se realizaron pruebas en las instalaciones de una empresa privada, donde fue posible poner en funcionamiento el vehículo y activar el sistema completo en condiciones controladas. Las restricciones de disponibilidad impuestas por el entorno de pruebas limitaron el tiempo y las acciones que se pudieron realizar, aunque la sesión resultó útil para comprobar el funcionamiento general del sistema.

Durante la prueba, el sistema de adquisición, ejecutado en una máquina virtual Ubuntu, capturó en tiempo real las tramas CAN, decodificándolas mediante el archivo `.dbc` e imprimiendo las señales interpretadas en consola. Esta actividad se muestra en la Figura 6.1, donde puede observarse el flujo continuo de mensajes ya procesados.

Simultáneamente, el dashboard desarrollado en Node-RED, alojado en el sistema host Windows, recibió los datos a través del broker MQTT, actualizando los distintos indicadores visuales de forma dinámica. La Figura 6.2 recoge una captura del panel en funcionamiento, donde pueden observarse algunas señales relevantes como voltaje de batería, velocidad del vehículo, presión de frenado o estado general.

Es importante destacar que todas las pruebas realizadas durante esta jornada de validación se llevaron a cabo en un entorno de red local (LAN). Tanto la máquina virtual de adquisición (Ubuntu) como el host Windows que ejecuta el broker MQTT y la interfaz Node-RED estaban conectados a la misma red interna, sin exponer los servicios a través de la IP pública ni utilizar la conexión 5G. La transmisión de datos por Internet y el acceso remoto quedaron reservados para futuras fases de prueba debido al tiempo disponible.

También mencionar que el archivo DBC utilizado en esta etapa no es oficial de Nissan, sino una reconstrucción elaborada por terceros. Como resultado, no todas las señales fueron interpretadas correctamente durante la prueba. A continuación, se resumen las principales observaciones:

- Las señales de **velocidad de cada rueda** y **consumo de motor** no presentaban el escalado correcto, lo que resultó en magnitudes incorrectas y exceso de decimales. Este problema se solventará mediante funciones de redondeo dentro del flujo de Node-RED, como ya se aplicó en la visualización del voltaje de la batería auxiliar.

- Algunas señales, como **VehicleSpeedCluster**, **ThrottlePosition**, **Wheel_Speed_ABS**, **BrakePressure1-4**, **BCM_VehicleState** y **ShifterPosition**, no mostraron actualizaciones durante la sesión, por falta de actividad durante la prueba, codificación incorrecta en el DBC o error en la suscripción al *topic* correspondiente (En el caso de *VehicleSpeedCluster* y *Wheel_Speed_ABS* se ha compro-

bado que la causa fue a raíz de un error a la hora de escribir el nombre de ambas señales para realizar la suscripción al *topic*).

- Las señales **DistanceTraveled1** y **DistanceTraveled2** mostraron un comportamiento de contador cíclico, incrementándose hasta 255 y reiniciando, lo cual no parece corresponderse con una distancia física recorrida.
- No fue posible inducir un fallo controlado en la **batería auxiliar**, por lo que no se pudo validar el comportamiento del campo **LB_Diagnosis_Trouble_Code**.

A pesar de estas limitaciones, la prueba confirmó el correcto flujo de adquisición, publicación y visualización de datos CAN en tiempo real. Las capturas incluidas evidencian que el sistema es funcional, y que con futuras sesiones de validación más prolongadas y el uso de archivos DBC más fiables, será posible ajustar los parámetros y ampliar el conjunto de señales útiles.

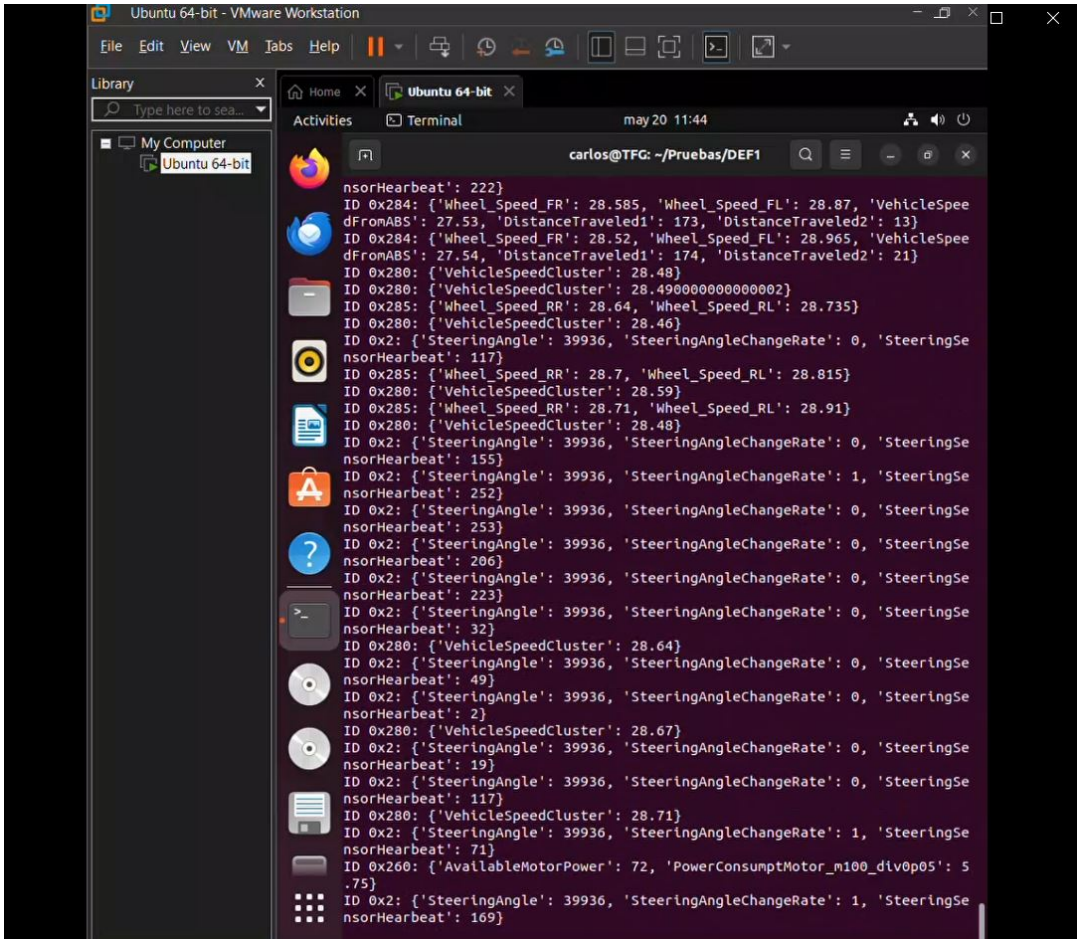

Figura 6.1: Flujo de datos CAN decodificados en la máquina virtual Ubuntu.

Figura 6.2: Captura del dashboard de Node-RED recibiendo y representando datos en tiempo real.

CAPÍTULO 7

Conclusiones y líneas futuras

Contenido

El presente capítulo recoge las conclusiones obtenidas tras el desarrollo e implementación del sistema de adquisición y visualización de datos del vehículo eléctrico. A lo largo del proyecto se ha abordado una solución completa que abarca desde la lectura de señales del bus CAN hasta su publicación mediante MQTT y su representación en tiempo real a través de una interfaz desarrollada en Node-RED.

Este apartado tiene como objetivo realizar una reflexión final sobre los resultados alcanzados, el valor técnico de las soluciones adoptadas y el aprendizaje obtenido durante el proceso. Además, se proponen posibles mejoras que permitirían aumentar la eficiencia, escalabilidad o versatilidad del sistema, así como futuras líneas de desarrollo que abren la puerta a nuevas aplicaciones e integraciones.

Con este análisis se cierra el ciclo de diseño, implementación y validación del sistema, contextualizando su utilidad dentro del ámbito de los sistemas embebidos, la conectividad en automoción y la monitorización remota de datos.

7.1. Conclusiones técnicas

El desarrollo de este proyecto ha permitido implementar con éxito un sistema completo de adquisición, procesamiento y visualización de señales del bus CAN de un vehículo eléctrico *Nissan Leaf* de primera generación (ZE0). A través del uso de herramientas *open source* y hardware accesible, se ha logrado:

- Capturar tramas CAN en tiempo real mediante un lector USB-CAN conectado por puerto serie.
- Filtrar y decodificar señales específicas a través de un archivo `.dbc`, utilizando la librería `cantools` para interpretar los datos en Python.
- Escalar las señales correctamente aplicando factores y *offsets* definidos por cada mensaje, adaptando la lectura bruta a unidades comprensibles (km/h, ºC, %, kW...).
- Transmitir las señales a un *broker* MQTT alojado localmente mediante el cliente `paho-mqtt`, separando completamente el plano de adquisición del de visualización.
- Desarrollar una interfaz para representar los datos en un panel en tiempo real utilizando Node-RED, que permite visualizar múltiples variables del vehículo en gráficos, medidores y textos, con organización temática.

Se ha seguido una arquitectura modular que desacopla la lógica de adquisición de la lógica de visualización, lo cual permite futuras extensiones del sistema sin necesidad de modificar su núcleo. Además, se ha verificado el correcto funcionamiento del sistema en condiciones reales mediante una fase de validación sobre el propio vehículo.

7.2. Valoración del aprendizaje adquirido

Este trabajo ha representado una experiencia integral que ha abarcado múltiples áreas técnicas, permitiendo aplicar y consolidar competencias transversales:

- **Electrónica y comunicaciones:** comprensión profunda del bus CAN como estándar de comunicación en automoción, análisis de tramas, uso de identificadores (IDs), modos de arbitraje y verificación mediante CRC. Se ha trabajado con un lector USB-CAN y se ha interpretado el flujo de datos binarios en bruto.
- **Interpretación de datos CAN mediante DBC:** uno de los aprendizajes clave ha sido el uso del archivo DBC como herramienta para descifrar las señales específicas del vehículo. Aunque todos los vehículos utilizan CAN, cada uno lo hace con una estructura de mensajes diferente. Entender cómo se definen las señales, en qué bits se sitúan, cómo aplicar factores de escala y offset, y cómo utilizar bibliotecas como *cantools* para su decodificación ha sido fundamental para poder obtener datos útiles del Nissan Leaf.

- **Programación en Python:** desarrollo de scripts robustos con manejo de excepciones, uso de múltiples librerías (*serial*, *cantools*, *paho-mqtt*, *argparse*, *threading*...), construcción de flujos modulares y diseño de funciones reutilizables orientadas a la adquisición y procesamiento de señales en tiempo real.
- **Redes e IoT:** despliegue de una red basada en MQTT sobre una infraestructura virtualizada con conectividad 5G, redirección de puertos y configuración de acceso remoto. Se ha comprendido la arquitectura cliente-servidor, el funcionamiento de brokers como Mosquitto y la distribución de datos a través de tópicos jerárquicos.
- **Interfaz gráfica y visualización:** uso de Node-RED como herramienta de desarrollo visual para representar y organizar las señales. Se ha trabajado con nodos de entrada, transformación y visualización (*ui_text*, *ui_gauge*, *function*, etc.) para construir un panel de control web accesible remotamente.

Además de los conocimientos técnicos, este proyecto ha favorecido el desarrollo de habilidades como la organización del trabajo, la documentación clara del código, la toma de decisiones sobre herramientas y arquitecturas, y la capacidad de resolución de problemas en entornos reales. Ha supuesto una oportunidad para enfrentarse a un caso de uso concreto y resolverlo de forma integral, desde el análisis de los datos CAN hasta su visualización remota en una plataforma IoT.

7.3. Posibles mejoras y futuras líneas de desarrollo

El sistema actual ofrece una base sólida para futuras ampliaciones tanto funcionales como tecnológicas. Algunas de las posibles líneas de evolución son:

- Incorporar **persistencia de datos** mediante bases de datos como InfluxDB, SQLite o PostgreSQL.
- Añadir **alertas y lógica avanzada** para notificar condiciones anómalas (sobretensiones, sobretemperaturas...).
- Mejorar la **interfaz gráfica** con paneles más detallados, controles remotos y exportación de datos.
- Exponer una **API web** que permita a aplicaciones externas consumir los datos del sistema.
- Enviar los datos a plataformas en la nube (*ThingsBoard*, *AWS IoT*, *Azure*, etc.) para análisis remoto o dashboards en línea.
- Añadir **compatibilidad multivehículo**, adaptando el sistema a distintos modelos mediante la carga dinámica de archivos *.dbc*.

- Mejorar la **ciberseguridad** del sistema mediante autenticación, cifrado de datos y control de accesos.
- Integrar el sistema de adquisición CAN en un **dispositivo embebido con Linux**, como una Raspberry Pi o Jetson Nano, o incluso explorar alternativas más ligeras como ESP32 con soporte CAN y MQTT, eliminando así la necesidad de una máquina virtual y mejorando la portabilidad del sistema.

Estas líneas abren el camino para convertir el sistema en una solución más completa y profesional, aplicable tanto a la monitorización particular como a flotas, investigación o integración en servicios de movilidad inteligente.

Bibliografía

[1] M. J. Pont, *Automotive Mechatronics: Operational and Practical Issues*. Springer Science & Business Media, 2012. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4614-0314-2

[2] A. Čolaković, A. Hasković Džubur, and B. Karahodža, "Wireless communication technologies for the internet of things," *Science, Engineering and Technology*, vol. 1, no. 1, pp. 1–14, 2021. [Online]. Available: https://www.researchgate.net/publication/355163956

[3] S. Fischer, "Nissan Leaf OBD-II manual," https://leaf-obd.readthedocs.io/en/latest/diagnostic-connector.html, consultado en febrero de 2025.

[4] nopnop2002, "Robotell-usb-can-python," https://github.com/nopnop2002/Robotell-USB-CAN-Python, repositorio GitHub, consultado en febrero de 2025.

[5] SonnePower, "¿Qué es el bus CAN?" https://www.spcontrolador.com/company-news/que-es-la-bus-can-como-funciona-y-sus-caracteristicas.html#:~:text=CAN_H%20y%20CAN_L%20a%202,0V%2C%20conocida%20como%20nivel%20recesivo, consultado en febrero de 2025.

[6] B. GmbH, "Can specification version 2.0," Robert Bosch GmbH, Stuttgart, Germany, Tech. Rep., 1991.

[7] K. Zeng, *Automotive Networking: Architecture, Applications, and Testing*. John Wiley & Sons, 2011.

[8] M. Farsi, K. Ratcliff, and M. Barbosa, *CANopen: A Practical Introduction to the Industrial Standard for CAN Based Networks*. London, UK: Springer, 1999.

[9] O. Pfeiffer, A. Ayre, and C. Keydel, *Embedded Networking with CAN and CANopen*. Copperhill Media, 2003.

[10] H. Niemann, "Automotive ethernet – the future of in-car networking," *IEEE Communications Magazine*, vol. 45, no. 11, pp. 166–173, 2007.

[11] SAE International, *SAE J1962: Diagnostic Connector and Associated Electrical Interface*. SAE International, 2001.

[12] S. Reif, *Automotive Mechatronics: Automotive Networking, Driving Stability Systems, Electronics*. Berlin, Germany: Springer, 2015.

[13] F. Quintero and L. Trujillo, "Vehicular telemetry systems: A review and future perspectives," *IEEE Latin America Transactions*, vol. 14, no. 2, pp. 1002–1010, 2016.

[14] U. of MyNissanLeaf Forum, "Leaf canbus decoding (open discussion)," https://mynissanleaf.com/threads/leaf-canbus-decoding-open-discussion.4131/, 2011, accessed: Apr. 26, 2025.

[15] A. Banks and R. Gupta, "Mqtt version 3.1.1," in *OASIS Standard*, Oct. 2014, https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

Anexo A: Acceso a repositorio GitHub

Enlace al repositorio GitHub con todo el código necesario para implementar lo desarrollado, junto con un archivo README.md con instrucciones de uso:
https://github.com/carlosdgg17/TFG-Nissan-Leaf-CAN-MQTT.