Escuela de Ingenierías Industriales

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO DE FIN DE GRADO

Análisis de los mensajes del bus CAN del rover J8

Autor: Álvaro Cardona Marina

Tutor: Jesús Morales Rodríguez

Cotutor: Jorge L. Martinez Rodríguez

Departamento de Ingeniería de Sistemas y Automática

Área de conocimiento: Ingeniería de Sistemas y Automática

Málaga, 12 de Enero de 2024

Dedicado a mi abuelo, José María.

Abreviaturas

ACK	Reconocimiento (*Acknowledgment*)
BMS	Sistema de Gestión de Baterías (*Battery Management System)*
CAN	Red de Área del Controlador (*Controller Area Network)*
CRC	Comprobación de Redundancia Ciclica (*Cyclic Redundancy Check*)
CTOR	Contactores
DBC	Base de Datos para CAN (*DataBase for CAN*)
ECU	Unidad de Control Electrónica (*Electronic Control Unit*)
EMI	Interferencias electromagnéticas (*Electromagnetic Interference*)
EOF	Final de Trama (*End Of Frame*)
GND	Tierra (*Ground*)
ID	Identificador
ISO	Organización Internacional de Normalización (*International Organization for Standardization*)
LCD	Pantalla de Cristal Líquido (*Liquid Crystal Display)*
LED	Diodo Emisor de Luz (*Light Emitting Diode*)
LiDAR	Detección y Medición por Luz (*Light Detection and Ranging*)
LLC	Control de Enlace Lógico (*Logic Link Control*)

MAC	Control de Acceso al Medio (*Medium Acces Control*)
MDI	Interfaz Dependiente del Medio (*Medium Dependent Interface*)
NC	Normalmente Cerrado (*Normally Closed*)
NO	Normalmente Abierto (*Normally Opened*)
NRZ	Sin Retorno a Cero (*Non Return to Zero*)
OSI	Interconexión de Sistemas Abiertos (*Open Systems Interconnection*)
PLS	Señalización Física (*Physical Signalling*)
PMA	Conexión al Medio Físico (*Physical Medium Attachment*)
ROS	Sistema Operativo de Robot (*Robot Operating System*)
SOC	Estado de Carga (*State Of Charge*)
SOF	Inicio de Trama (*Start of Frame*)
SPD	Velocidad (*Speed*)
TFG	Trabajo de Fin de Grado
USB	Bus Serial Universal (*Universal Serial Bus*)
VLTG	Voltaje (*Voltage*)

Índice

Índice de figuras

Capítulo 1

Introducción

Contenido

Sinopsis

Este es el primer capítulo introductorio sobre el desarrollo del proyecto, donde se establece un marco general sobre el presente TFG, sus objetivos y cómo se ha organizado esta memoria.

1.1. Antecedentes

El rover J8 de Argo es un vehículo robótico terrestre, alimentado por baterías, con una gran capacidad de movimiento sobre superficies irregulares (ver Figura 1.1). Dispone de tracción *skid-steer* con ocho neumáticos de baja presión de 24 pulgadas, una velocidad máxima de 27 km/h y radio de giro nulo.

Internamente, J8 utiliza el bus CAN para comunicar los distintos dispositivos a bordo. Este bus se emplea comunmente en automoción para enlazar los distintos ECUs (*Electronic Control Unit*) de un vehículo (ver Figura 1.2).

Sin embargo, J8 es un sistema cerrado, es decir, no se pueden conocer datos relevantes como la velocidad de las ruedas o el porcentaje de batería restante, por ejemplo. Pero, a largo plazo se pretende convertirlo en un robot autónomo. Para ello, primero necesitamos una interfaz con el rover para conocer el estado del mismo y para poder moverlo.

Figura 1.1: El rover J8 de ARGO.

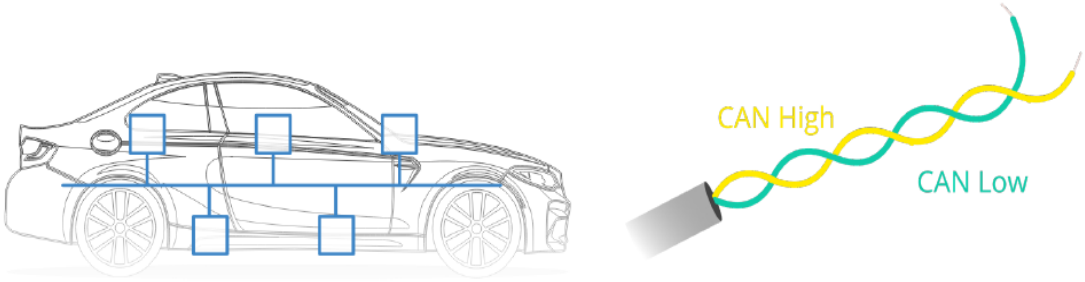


Figura 1.2: Bus CAN. Fuente: CAN bus - The Ultimate Guide (2023).

1.2. Objetivos

En este trabajo de fin de grado se estudiarán los mensajes enviados a través del bus CAN del rover J8 de los motores de tracción y de dirección, los enviados por las baterías y sus cargadores y por el receptor de radio. También se explicará cómo se ha llevado a cabo la obtención de dichos mensajes.

La principal dificultad reside en que no hay información acerca de muchos de estos mensajes, además de que no disponemos de diccionarios de decodificación para los mensajes de los motores ni del receptor de radio. Por tanto, es necesario un trabajo de ingeniería inversa para descifrarlos.

1.3. Estructura del documento

Este proyecto está estructurado en seis capítulos y dos apéndices. El primero es un capítulo introductorio. El segundo detalla el rover J8, sus partes y características principales. En el tercero se pondrá en contexto sobre qué es y para que sirve el bus CAN. El capítulo cuarto trata sobre las redes bus CAN en J8, sobre cómo se ha conectado la computadora externa a esta red y sobre cómo se ha llevado a cabo la extracción de los mensajes que nos conciernen. El quinto capítulo consiste en la decodificación de los mensajes enviados a traves del bus CAN por las baterías, cargador, motor de tracción, motor de dirección y el receptor de radio. Por ultimo, en el capítulo sexto de conclusión se hará una breve recapitulación sobre el proyecto, se pondrá en manifiesto lo aprendido y se comentará para qué se podrá utilizar lo estudiado en un futuro.

En los apéndices se desglosarán los diccionarios de decodificación de los mensajes de los motores, baterías, cargadores y receptor de radio desarrollados en este proyecto, así como los códigos utilizados en Python para descifrar dichos mensajes.

Capítulo 2

El rover J8 de Argo

Contenido

Sinopsis

En este capítulo se describe el rover J8, sus partes más importantes y sus características principales.

2.1. Introducción

El rover J8 es un vehículo todoterreno, equipado con ocho ruedas de baja presión con transmisión por cadenas, diseñado para operar en condiciones extremas y terrenos difíciles (ver Fígura 1.1). La empresa canadiense ARGO, especializada en la fabricación de vehículos robóticos para diferentes aplicaciones, es la encargada de su producción.

El diseño todoterreno de J8 le permite trabajar en lugares remotos y de complicado acceso. Además, está equipado con una variedad de sensores que le permiten detectar obstáculos y evitar colisiones.

El vehículo mide 3.2 metros de largo y 1.6 metros de ancho. Pesa 1090 kg y tiene una capacidad de remolque de 680 kg. Las baterías duran entre cinco y seis horas, dependiendo de las condiciones de funcionamiento. Es un vehículo anfibio (puede desplazarse por agua) con centro de gravedad bajo.

Cuenta con dos motores eléctricos (motor principal de tracción y motor secundario de dirección) conectados a la transmisión. Todas estas cualidades lo convierten en un rover sumamente idóneo para el transporte de material pesado por terrenos desfavorables en expediciones de larga duración.

2.2. Componentes principales

2.2.1. Computadora

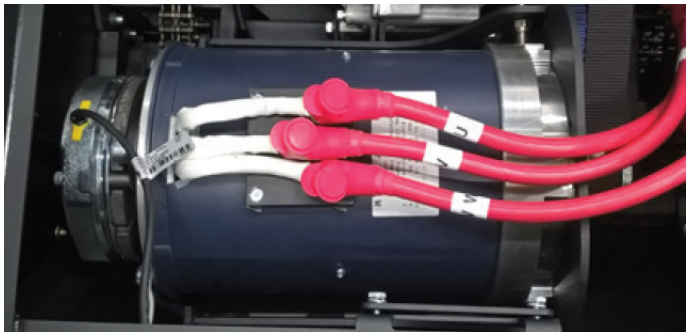
La computadora que utiliza J8 es un Advantech UNO-2483G-474AE embebido en una caja i7-4650U (ver Figura 2.1). Tiene un diseño modular y sin ventilador. Incluye tecnología *iDoor* que admite extensiones de funciones de automatización, como comunicación de bus de campo industrial, Wi-Fi/3G y E/S digital.

Figura 2.1: Ordenador Advantech UNO-2483G-474AE. Fuente: Atlas J8 Operator´s Manual (2017).

2.2.2. Motores

El rover J8 dispone de dos motores eléctricos: el de tracción y el de dirección. El motor principal de tracción es un motor eléctrico AC34-28.11 (ver Figura 2.2a). Este motor se encarga del movimiento en línea recta tanto en avance como en retroceso. Consta de un freno electromagnético en el extremo del eje, el cual es usado para mantener parado el vehículo. Este freno está normalmente cerrado, por lo cual, cuando el vehículo pierda energía o señales de control, se activará.

Por otro lado, el motor secundario, de dirección, se encarga de dirigir el movimiento propiciado por el motor principal. Este motor secundario es un motor eléctrico AC9-03.27.8 (ver Figura 2.2b) que cuenta con un engranaje reductor (ver Figura 2.3).



(a) Motor de tracción.

(b) Motor de dirección.

Figura 2.2: Motores de J8. Fuente: Atlas J8 Operator´s Manual (2017).

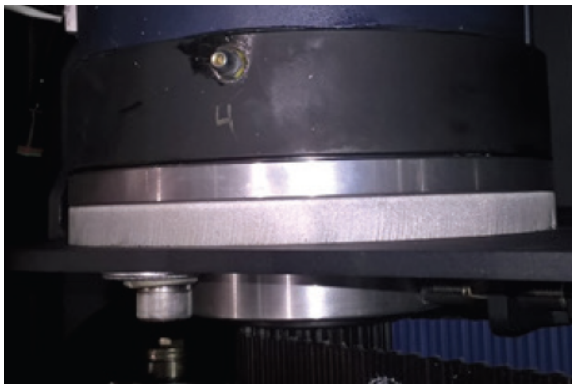


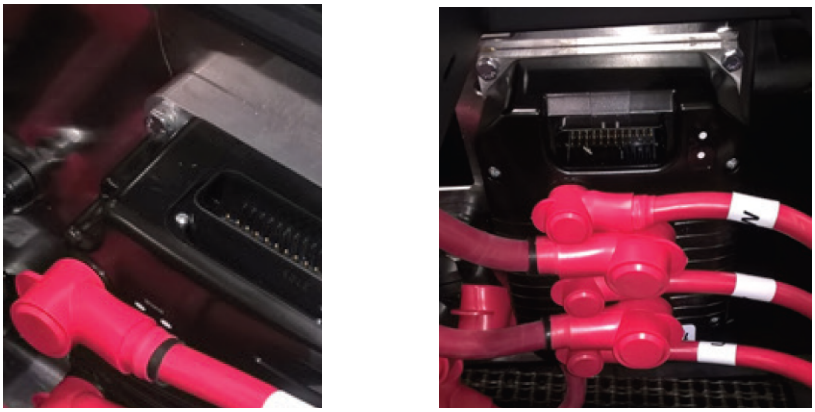
Figura 2.3: Engranajes de reducción. Fuente: Atlas J8 Operator´s Manual (2017).

2.2.3. Controladoras de motor

Como ya hemos mencionado, el vehículo cuenta con dos motores, el de tracción y el de dirección. Ambos motores cuentan con su propia controladora que

se encargan de alimentarlos para ejecutar los giros que se les comandan.

La controladora del motor de tracción es una Curtis 1236SE-5621 (ver Figura 2.4a) y se sitúa en la sección frontal de la unidad, junto al motor de tracción en la parte izquierda de dicha sección. Por otro lado, la controladora del motor de dirección es una Curtis 1234SE-5421 (ver Figura 2.4b), situado junto al motor de dirección en la parte derecha de la unidad.



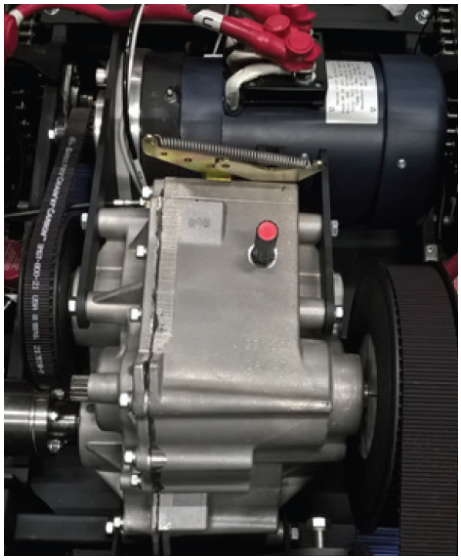
(a) Controladora del motor de tracción.

(b) Controladora del motor de dirección.

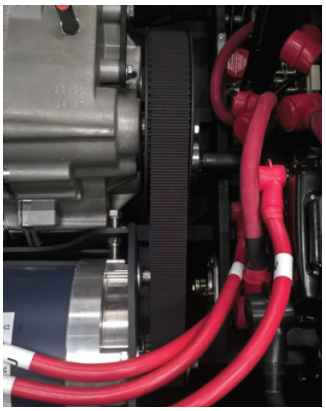
Figura 2.4: Controladoras de los motores de J8. Fuente: Atlas J8 Operator´s Manual (2017).

2.2.4. Caja de transmisión

El rover J8 cuenta con correas (ver Figura 2.5) entre cada motor y la transmisión 34-300 (ver Figura 2.5a), la cual, a su vez, mueve las ruedas de cada lado mediante cadenas.



(a) Caja de transmisión.



(b) Correa del motor de tracción.

(c) Correa del motor dirección.

Figura 2.5: Caja de transmisión y correas de los motores de J8. Fuente: Atlas J8 Operator´s Manual (2017).

2.2.5. Parada de emergencia

J8 dispone de un botón cableado de parada de emergencia situado en el mastil (ver Figura 2.6). Al pulsar dicho botón el vehículo se detendrá abruptamente.

Figura 2.6: Botón de parada de emergencia.

Además del botón de parada de emergencia del vehículo, existen otros dos. Uno en el control remoto (ver Figura 2.8) y otro en un mando inalámbrico que puede ser usado por otro operario que no esté controlando el rover (ver Figura 2.7).

Figura 2.7: Botón de parada de emergencia en el mando inalámbrico.

2.2.6. Control remoto de *Hetronic*

El vehículo cuenta con un control remoto via radio del fabricante *Hetronic* usado para manejar el rover J8 (ver Figura 2.8). Este control tiene una banda para la cintura con el objetivo de mantener las manos del usuario libres. Además, usa un sistema de doble *joystick* en el que cada uno controla una función diferente del vehículo:

- *Joystick* izquierdo: avance y retroceso.
- *Joystick* derecho: giro a izquierda y derecha.

El grado en el que cada *joystick* es movido se traduce en la correspondiente velocidad del vehículo, es decir, cuanto más se mueva el *joystick* hacia el extremo, más rápido avanzará o más agresivo será el giro que realice. Al soltar los *joystick*, el vehículo regresará automáticamente a velocidad nula y activará los frenos.

Figura 2.8: Control remoto de *Hetronic*.

Además, cuenta con una pequeña pantalla LCD (*Liquid Crystal Display*) de información y de un pulsador para regular la velocidad máxima de funcionamiento.

2.2.7. Sensores LiDAR y cámaras

Los sensores LiDAR (Light Detection and Ranging), son dispositivos que emplean pulsos láser para medir distancias y crear mapas tridimensionales o bidimensionales de alta resolución del entorno. El rover J8 cuenta con un LiDAR tridimensional *Velodyne 16 PUCK* y otro bidimensional *SICK TIM 561* (ver Figura 2.9).

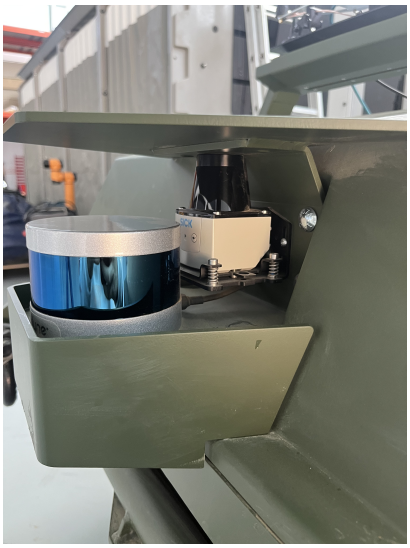
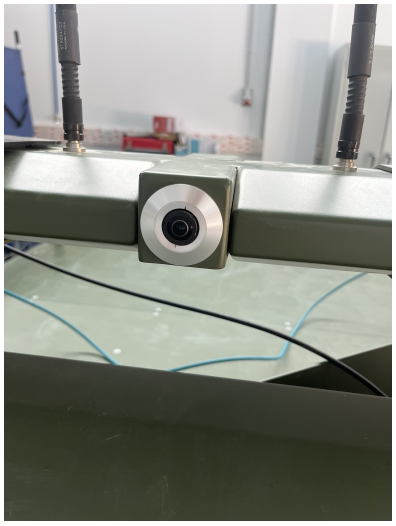


Figura 2.9: LiDARS en la parte delantera del vehículo.

Además de los LiDAR, el vehículo cuenta con una camara delantera y otra trasera situadas en la parte superior del mástil (ver Figura 2.10).



(a) Cámara delantera.

(b) Cámara trasera.

Figura 2.10: Cámaras del rover J8.

2.2.8. Baterías

La energía principal es suministrada por cuatro baterias individuales de 5 kW/h cada una, del fabricante *Brigs & Stratton* (ver Figura 2.11). Cada una de ellas tiene un sistema interno de gestión BMS (Battery Management System) que monitoriza su estado, comunica diversas mediciones y cálculos, y se asegura de que la descarga de la batería no baje a un límite peligroso. Son capaces de operar independientemente en caso de que otra falle.

Figura 2.11: Baterías. Fuente: Atlas J8 Operator´s Manual (2017).

Las baterías constan de cuatros estados: *Sleep, Discharge, Charge y Hybrid*. El modo *Sleep* deberá emplearse cuando no esté en uso. Este modo ayuda tanto a preservar el estado de carga como a agregar otro nivel de seguridad al desconectar los terminales de la propia batería.

El modo *Discharge* es el estado principal, la batería espera que se extraiga corriente de ella, así como recibir corriente de regeneración. Una vez que se pone la batería en modo *Discharge*, hace rápidamente una serie de pruebas para determinar si es seguro, comprobando que no haya una fuente de alimentación inesperada conectada a la batería, además de comprobar que todas las baterías en paralelo tienen la misma tensión.

El modo *Charge* se utiliza para cargar las baterías de forma segura y apropiada mediante el cargador de *Vanguard*. Este último pone automáticamente la bateria en modo *Charge* cuando es conectado a esta y a una fuente de corriente alterna. Para ello, se requiere comunicación a través del bus CAN (de la cual se hablará más adelante) entre la batería y el cargador.

El modo *Hybrid* es, como su propio nombre indica, un hibrido entre el modo *Discharge* y el modo *Charge*. Está pensado para operar mientras se carga la batería vía generador u otra fuente de alimentación. Para entrar en este modo, la batería debe estar previamente en modo *Discharge* y posteriormente conectarle el cargador de *Vanguard*.

2.2.9. Cargador de baterías

El encargado de recargar las batería es un cargador industrial *Switch Mode* de *Vanguard* (ver Figura 2.12). Este cargador cuenta con algorítmos avanzados de carga diseñados para optimizar tanto la capacidad de la batería como su vida útil general. No tiene piezas móviles y está sellado al vehículo. La entrada de corriente alterna universal que posee le permite usar una gran variedad de tensiones y frecuencias de corriente alterna. Las funciones de la interfaz del cargador incluyen cuatro LEDs y un puerto externo para la comunicación con el bus CAN (*Controller Area Network*) y un LED montado en panel remoto. El modelo de 1050W tambien es capaz de carga multitensión automática de corriente continua, lo que permite la detección y el ajuste automáticos de la tensión de salida de corriente continua en función del paquete de baterías conectado.

El cargador cuenta con comunicación Bluetooth, que habilita a un teléfono móvil o tableta a usar la aplicación *ChargerConnect*™, que puede ser usada para:

- Ver el estado del ciclo de carga en tiempo real.
- Descargar el historial de los ciclos de carga del cargador.
- Subir el historial de los ciclos de carga del cargador a la nube.
- Seleccionar el perfil activo de batería.
- Descargar perfiles nuevos de batería desde la nube.
- Subir perfiles de batería al cargador.

Figura 2.12: Cargador *Vanguard* "Switch Mode". Fuente: Atlas J8 Operator´s Manual (2017).

2.3. Sistema de locomoción

El rover J8 cuenta con tracción *skid-steer*, lo cual es un diseño de vehículo que utiliza un método particular para controlar la dirección y velocidad. Este sistema es comúnmente utilizado en vehículos como minicargadores (nuestro caso), miniexcavadoras y otros equipos de construcción compactos.

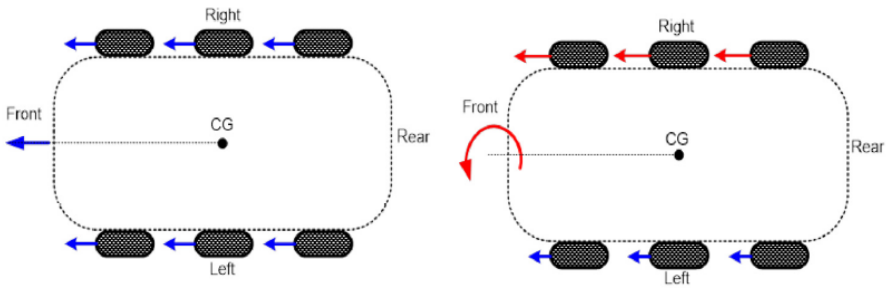
En un vehículo *skid-steer*, las ruedas de cada lado pueden girar de forma independiente. La dirección y velocidad se controlan ajustando la velocidad de rotación de las ruedas en cada lado del vehículo. Por ejemplo, para girar a la izquierda, se reduce la velocidad de las ruedas del lado izquierdo o se aumenta la velocidad de las ruedas del lado derecho (ver Figura 2.13b). Este sistema proporciona gran maniobrabilidad, ya que el vehículo puede girar sobre su propio eje central, esto se consigue manteniendo la velocidad de las ruedas de cada lado igual, pero con distinto signo.

Comunmente, este sistema de locomoción se implementa con dos motores, uno para cada lateral. Pero, en el rover J8, como se ha comentado, el motor principal se encarga de la tracción hacia delante o hacia atrás y el motor de dirección suma movimiento a un lateral y se lo resta al otro. Todo ello, a través de las cajas de transmisión (ver Figura 2.14).

En concreto, siendo V_t la velocidad de tracción, V_d la velocidad de dirección, V_l la velocidad de las ruedas del lado izquierdo y V_r la velocidad de las ruedas del lado derecho, todas en rad/s, entonces:

$$V_l = V_t + V_d \tag{2.1}$$

$$V_l = V_t - V_d \tag{2.2}$$



(a) Avance en línea recta. (b) Giro hacia la izquierda.

Figura 2.13: Mecanismo de dirección *skid-steer*. Fuente: Comprehensive study of skid-steer wheeled mobile robots: development and challenges (2020).

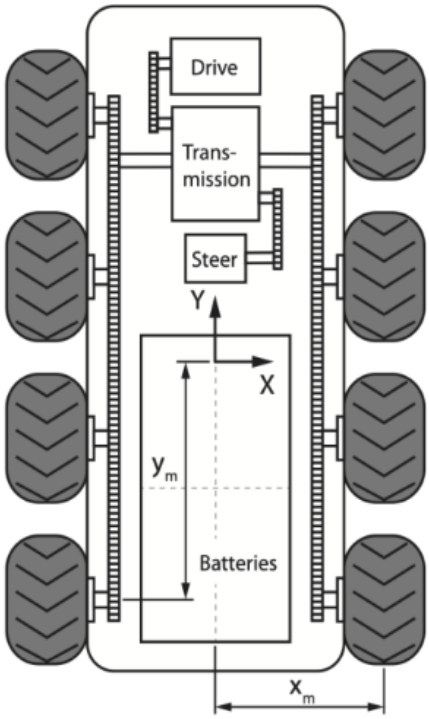


Figura 2.14: Esquema de locomoción de J8. Fuente: Elaboración propia.

2.4. Modos de funcionamiento

En la actualidad, J8 tiene dos modos de funcionamiento: por radio control (con el control remoto mencionado anteriormente) y otro modo llamado *follow me* (sígueme). Este último se activa presionando un botón en la parte delantera del mástil (ver Figura 2.15). El modo *follow me* hace uso de los LiDARS del vehículo, para detectar una persona delante de éste y seguirla a una cierta distancia de seguridad.

Figura 2.15: Botón *follow me* en la parte delantera del vehículo.

Capítulo 3

El Bus CAN

Contenido

Sinopsis

En este capítulo se hará una introducción detallada sobre qué es, cómo funciona y para qué se usa el bus CAN. Se abordará su origen y su utilidad. También se comentará el modelo de comunicaciones OSI.

3.1. Origen histórico

El surgimiento del bus CAN (*Controller Area Network*) tuvo origen en la necesidad imperante de las empresas fabricantes de automóviles durante la década de los 80. En este periodo, se experimentaba un incremento sustancial en el número de unidades electrónicas incorporadas a los vehículos. El propósito era agilizar la conexión entre estos dispositivos electrónicos, en lugar de depender de las complejas y pesadas conexiones punto a punto que se utilizaban hasta entonces. El objetivo principal consistía en reemplazar los extensos metros y kilogramos de cables utilizados en las conexiones individuales entre dispositivos electrónicos, optando por una red digital de comunicaciones que permitiera la interconexión eficiente de dichos dispositivos (ver Figura 3.1). La propuesta y desarrollo del bus CAN fueron llevados a cabo por Robert Bosch GmbH, fabricante de componentes para automóviles. La especificación de Bosch se estandarizó posteriormente en la norma ISO 11898-1.

La robustez, facilidad de implementación y gran nivel de capacidad en tiempo real, junto con una serie de atributos que se detallarán a continuación, propiciaron la rápida adopción del bus CAN en aplicaciones industriales. Este uso se extendió tanto en sistemas embebidos, como en entornos abiertos destinados a la automatización, donde sirve como base para diversos buses de campo tales como CANOpen, DeviceNet, entre otros.

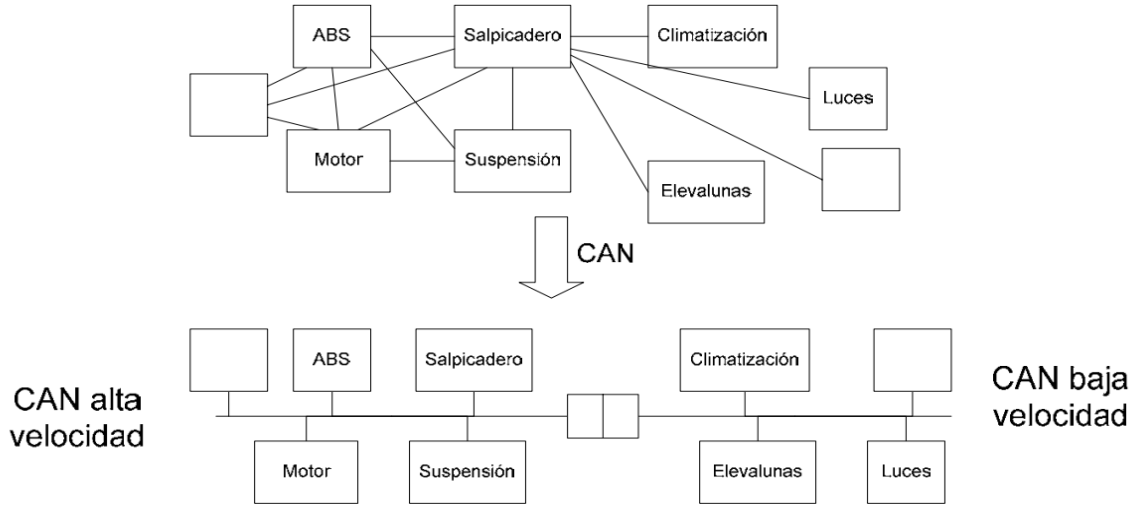


Figura 3.1: Necesidad y origen del bus CAN. Fuente: Comunicaciones industriales: Sistemas distribuidos y aplicaciones (2007).

3.2. El modelo OSI

El Modelo OSI (*Open Systems Interconnection*), es un marco conceptual estandard que describe las funciones de un sistema de red o telecomunicaciones en siete capas (ver Figura 3.2). Fue desarrollado por la Organización Internacional de Normalización (ISO) para estandarizar y facilitar la comunicación entre máquinas.

Las siete capas del Modelo OSI son:

- Capa Física (*Physical Layer*): Esta capa se encarga de la transmisión física de bits a través de un medio de comunicación, como cables, fibras ópticas o señales inalámbricas. Define características eléctricas, mecánicas y de procedimiento para activar, mantener y desactivar conexiones físicas.
- Capa de Enlace de Datos (*Data Link Layer*): Aquí se gestionan los enlaces de comunicación en la capa física. Su función incluye la detección y corrección de errores, control de flujo y organización de los datos en tramas para su transmisión eficiente.
- Capa de Red (*Network Layer*): Se ocupa del enrutamiento de datos a través de la red. Esta capa determina la ruta que los datos deben seguir para llegar desde el origen hasta el destino, independientemente de la topología física de la red.
- Capa de Transporte (*Transport Layer*): Proporciona la transferencia de datos confiable y eficiente entre sistemas finales. Se encarga de la segmentación y reensamblaje de datos, así como del control de flujo y de la recuperación de errores.
- Capa de Sesión (*Session Layer*): Establece, mantiene y finaliza sesiones de comunicación entre aplicaciones en diferentes dispositivos. Facilita el diálogo y controla el intercambio de datos, proporcionando servicios de inicio, cierre y sincronización de sesiones.
- Capa de Presentación (*Presentation Layer*): Se encarga de la traducción, compresión y codificación de los datos para garantizar que los sistemas finales puedan entender la información intercambiada. También maneja la sintaxis y semántica de los datos.
- Capa de Aplicación (*Application Layer*): Proporciona servicios de red directamente a las aplicaciones del usuario final. Incluye protocolos para funcio-

nes como correo electrónico, transferencia de archivos, acceso a bases de datos y navegación web.

El Modelo OSI es un marco teórico y conceptual que ayuda a establecer las funciones de las distintas capas de una red. Cada capa realiza funciones específicas y se comunica con las capas adyacentes para lograr la transmisión de datos de manera eficiente y estandarizada. Este modelo no prescribe protocolos específicos, pero proporciona una base común para el diseño y la implementación de sistemas de red.

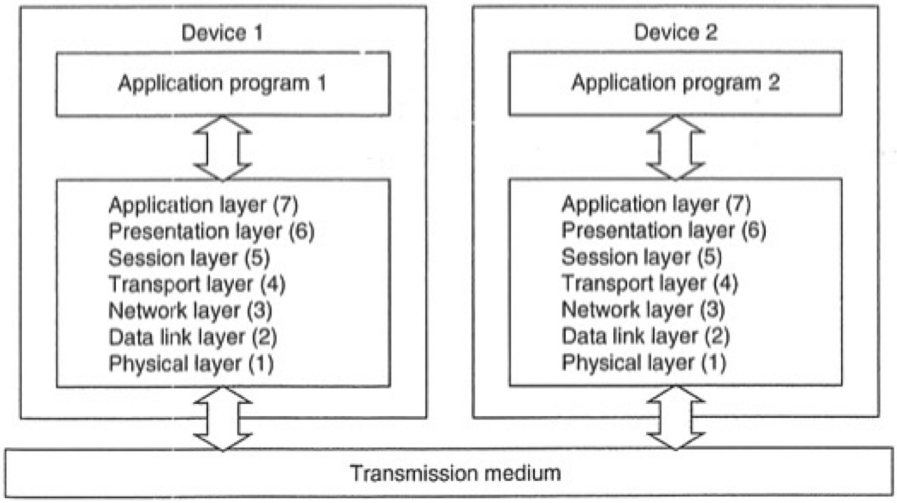


Figura 3.2: Modelo OSI de 7 capas. Fuente: Building Automation. Communication Systems with EIB/KNX, LON, anf Bacnet (2009).

3.3. El bus CAN

El bus CAN, como su propio nombre indica, implementa un sistema de comunicación en bus, posibilitando la interacción de varios dispositivos conectados al mismo medio de comunicación. Fundamentado en el modelo cliente/servidor, opera como un protocolo orientado a mensajes, donde la información destinada al intercambio se divide en mensajes. Estos mensajes se caracterizan por poseer un identificador específico y son encapsulados en tramas para su posterior transmisión.

El protocolo CAN es un sistema de comunicación serie asíncrono y multimaestro diseñado para conectar unidades de control electrónicas, sensores y

actuadores en aplicaciones automotrices e industriales. La especificación de Bosch, estandarizada en ISO-11898-1, define la capa de enlace de datos y aspectos de la capa física independientes del medio de transmisión, con la capa física completada en diversas especificaciones.

El bus CAN opera como una red de libre difusión, permitiendo que cualquier nodo escuche las tramas transmitidas. La norma ISO-11898-1 establece dos estados complementarios en el medio de transmisión: dominante y recesivo. La transmisión simultánea de bits dominantes y recesivos resulta en un estado dominante en el medio.

La norma describe dos versiones del protocolo CAN: 2.0A, con formato de mensaje estándar de 11 bits, y 2.0B, que define el formato de mensaje extendido con identificador de 28 bits. Se denomina controlador CAN a un dispositivo electrónico con implementación del protocolo, siendo 2.0B pasivo capaz de recibir y transmitir mensajes estándar y admitir mensajes extendidos sin almacenarlos. Un controlador CAN 2.0B activo puede almacenar y enviar mensajes en formato extendido, respaldando ambas versiones del protocolo.

En la Figura 3.3 se puede apreciar el formato estandar de una trama, que consta de:

- SOF (*Start of Frame*): El principio de la trama es 'dominante 0' para informarle a los demas nodos que un nodo CAN intenta comunicarse.
- ID: Identificador de la trama. Valores bajos tienen mayor prioridad.
- RTR (*Remote Transmission Request*): Indica si un nodo envía datos o si los solicita de otro nodo.
- Control: Contiene el IDE (*Identifier Extension Bit*) el cual es 'dominante 0' para 11 bits. También tiene el DLC (*Data Length Code*) que especifica la longitud de los bytes de datos a transmitir (de 0 a 8 bytes).
- Datos: Contiene los bytes de datos, conocidos como carga útil, que incluyen señales CAN que pueden ser extraídas y decodificadas para adquirir información.
- CRC (*Cyclic Redundancy Check*): Es usado para asegurar la integridad de los datos.
- ACK (*acknowledgment*): El espacio ACK indica si el nodo ha reconocido y recibido los datos correctamente.

- EOF (*End Of Frame*): Marca el fin de la trama CAN.

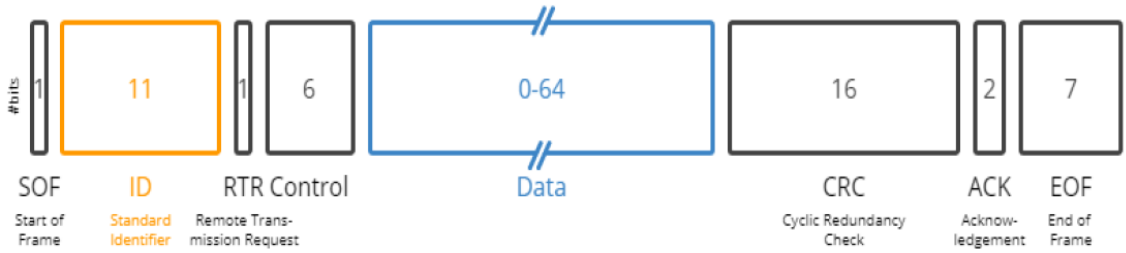


Figura 3.3: Trama CAN estandard. Fuente: CAN bus - The Ultimate Guide (2023).

El bus CAN desempeña un papel fundamental en el ámbito automotriz al servir como un medio de comunicación para las conocidas como unidades de control electrónicas (ECUs) integradas en los sistemas que supervisan (ver Figura 3.4). Los vehículos típicamente cuentan con múltiples redes CAN, siendo común la presencia de dos: una de alta velocidad, vinculada a ECUs en tiempo real como las de inyección de combustible y frenado ABS, y otra de baja velocidad, destinada a dispositivos electrónicos como elevalunas, sistemas de iluminación y climatización. Algunos automóviles incluyen también una red CAN de 125 kbit/s con transceptores tolerantes a fallos para unir las unidades controladoras de *airbags*.

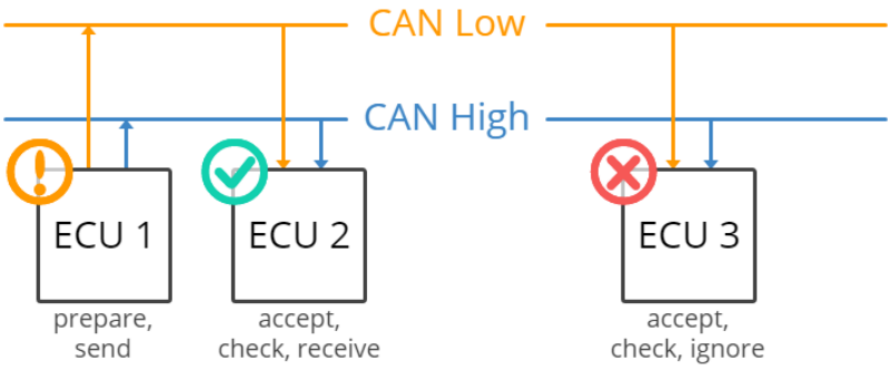


Figura 3.4: Conexión de unidades de control electrónicas (ECU) mediante redes CAN de alta y baja velocidad. Fuente: CAN bus - The Ultimate Guide (2023).

Otras características de este protocolo de comunicación son:

- Prioridad de mensajes.

- Garantía de tiempos de latencia.
- Flexibilidad en la configuración.
- Recepción por multidifusión con sincronización de tiempos.
- Sistema robusto en cuanto a consistencia de datos.
- Sistema multimaestro.
- Detección y señalización de errores.
- Retransmisión automática de tramas erróneas.
- Distinción entre errores temporales y fallos permanentes de los nodos de la red, y desconexión automática de los nodos defectuosos.

3.3.1. Nodos CAN

En el contexto de este capítulo, se define un nodo CAN como una unidad de control electrónica que contiene un microprocesador o microcontrolador con el respectivo programa de aplicación y el software para las capas superiores del protocolo. Además, debe incluir un controlador CAN, al menos 2.0B pasivo, y un transceptor CAN de alta velocidad sobre un par de hilos con retorno según la norma ISO-11898-2, utilizada en aplicaciones industriales y de automatización.

Estos componentes suelen implementarse de dos maneras comunes: cada uno en un circuito integrado separado o integrando el controlador CAN y el microcontrolador en el mismo encapsulado (ver Figura 3.5). Las opciones tienen ventajas e inconvenientes relacionados con el precio, el espacio ocupado, la fiabilidad, la carga en la CPU para el acceso al controlador CAN y la reusabilidad del software.

Incluso en el sector automotriz, se han desarrollado productos que integran los tres elementos en el mismo chip para superar la dificultad de combinar diferentes tecnologías en un solo sustrato. En la actualidad, muchos microcontroladores de bajo costo incluyen un controlador CAN de comunicaciones.

La conexión entre el controlador CAN y el transceptor se realiza a través de una salida del controlador CAN, TX, que indica el nivel dominante o recesivo que el transceptor debe colocar en el bus, y una entrada RX, donde el transceptor indica al controlador el estado actual del bus. Por último, el software para las capas 3 a 7 de ISO/OSI puede ajustarse a la especificación correspondiente de

la capa de aplicación de un bus de campo como DeviceNet o CANOpen, o puede ser una solución propietaria.

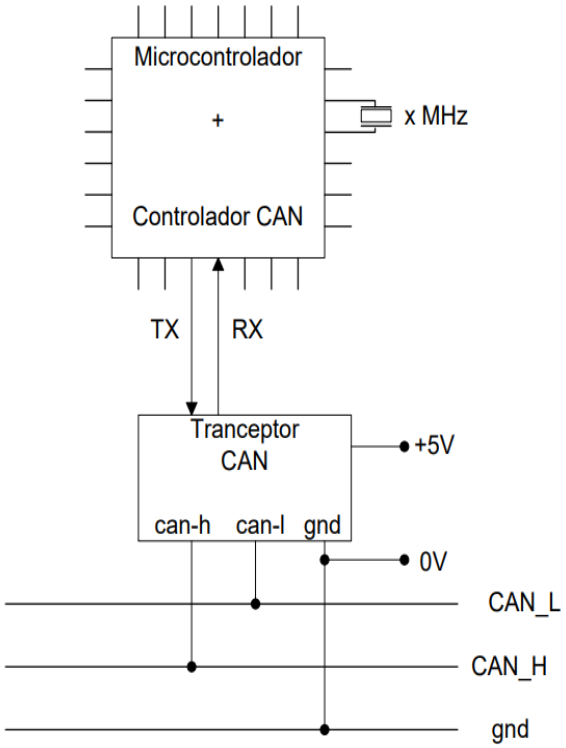


Figura 3.5: Nodo CAN con controlador CAN integrado en microcontrolador y transceptor. Fuente: Comunicaciones industriales: Sistemas distribuidos y aplicaciones (2007).

3.3.2. Capa de enlace de datos

En el marco del modelo ISO/OSI, los servicios proporcionados por la capa de enlace de datos en el protocolo CAN se materializan en las subcapas de Control de Enlace Lógico (LLC) y Control de Acceso al Medio (MAC). En el contexto de CAN, se emplean mensajes con una estructura predefinida, también conocidos como tramas, para facilitar la gestión de las comunicaciones. La subcapa MAC desempeña diversas funciones, siendo una de sus responsabilidades clave la definición del formato de las tramas utilizadas en los mensajes.

Existen cuatro tipos de tramas distintas:

- Trama de datos (*Data Frame*), por la que el nodo transmisor envía datos a

los receptores.

- Trama remota (*Remote Frame*), por la que un nodo solicita la transmisión de una trama de datos con el mismo identificador.
- Trama de error (*Error Frame*), que se transmite por un nodo cuando detecta un error en el bus.
- Trama de sobrecarga (*Overload Frame*), que transmite un nodo cuando el nodo requiere un retardo extra antes de recibir la próxima trama de datos/-remota.

Codificación de la trama

La subcapa MAC en el protocolo CAN se encarga de codificar las tramas. Dado que el bus CAN opera de manera asíncrona, no cuenta con una señal de reloj adicional para sincronizar la recepción entre los diferentes nodos y el nodo transmisor. Para abordar esto, los controladores CAN utilizan una señal de reloj local con una frecuencia más alta que la de la transmisión en el bus, sincronizándose con los flancos de transición entre estados en la línea de bus.

En el protocolo CAN, la secuencia de bits en un mensaje se codifica mediante el método NRZ (*Non-Return-to-Zero*). Esto implica que durante todo el tiempo de bit, el nivel de bit generado es dominante o recesivo. Sin embargo, para evitar problemas de sincronización cuando se transmiten muchos bits consecutivos con la misma polaridad, se implementa la técnica de *bit stuffing* en las tramas CAN. Durante la transmisión, el transmisor introduce un bit de relleno de polaridad contraria cuando detecta hasta 5 bits consecutivos con la misma polaridad. Los receptores eliminan este bit de relleno al recibir el mensaje.

Detección y gestión de errores

La capa MAC en el protocolo CAN desempeña funciones críticas de chequeo y señalización de errores en las tramas. Se identifican varios tipos de errores:

- Error de bit: Cada nodo transmisor verifica si el bit transmitido coincide con el bit detectado en el bus. Se considera un error de bit si se recibe un bit con polaridad opuesta a la transmitida, excepto en el campo de arbitraje o en el bit de reconocimiento.

- Error de relleno (*Stuff Error*): Se detecta un error de relleno cuando hay 6 bits consecutivos del mismo signo en cualquier campo sujeto a la regla de relleno.
- Error de forma (*Form Error*): Se produce cuando un campo de formato fijo llega alterado en algún bit.
- Error de reconocimiento (*Acknowledgment Error*): Sucede cuando ningún nodo receptor transmite un bit dominante durante el bit de reconocimiento.

Si un nodo detecta un error y está en estado de error activo, transmite una trama de error consistente en 6 bits dominantes sucesivos (violando la regla de relleno de bits), invalidando así el mensaje para todos los demás nodos. El nodo transmisor intentará reenviar el mensaje cuando las condiciones del bus lo permitan.

Para un transmisor, un mensaje es válido si no se detecta ningún error hasta el final del campo de fin de trama. Para un receptor, un mensaje es válido si no se detecta ningún error hasta el antepenúltimo bit del campo de fin de trama. Esto asegura una consistencia total de la información en el sistema distribuido: todos los nodos reciben la misma información al mismo tiempo o no reciben información válida.

Aislamiento de nodos con fallo

El estado de un nodo en relación con el bus puede clasificarse como activo (capaz de transmitir y recibir mensajes), pasivo (requiere esperar una secuencia adicional de bits recesivos para transmitir y no puede señalar errores mediante una trama de error activa) o anulado (debe desactivar su transceptor y queda excluido de la comunicación, en un estado denominado *bus off*).

Cada nodo sigue un proceso de autodiagnóstico sofisticado. Cuando un nodo acumula errores, cambia de un estado activo a uno pasivo. Si la degradación persiste, el nodo pasa al estado anulado, autoexcluyéndose de la comunicación para evitar perturbar a los demás nodos de la red. Este mecanismo garantiza la integridad y estabilidad del sistema, permitiendo a los nodos adaptarse según su salud y contribuir a un entorno de comunicación más fiable.

3.3.3. Capa física

Cualquier capa física CAN tiene que soportar la representación de los estados recesivo y dominante en el medio de transmisión. El medio de transmisión estará

en el estado recesivo si ningún nodo del bus transmite un bit dominante y estará en estado dominante si uno o más nodos transmiten un bit dominante. Además, la capa física debe ser capaz de transmitir y recibir señales al mismo tiempo.

La capa física del protocolo CAN se divide en tres subcapas: PLS (señalización física), PMA (conexión al medio físico), y MDI (interfaz dependiente del medio). La PLS implementada en los controladores CAN, aborda la codificación, sincronización y temporización a nivel de bit.

La capa PMA describe las características del transceptor, que conecta el controlador CAN al bus mediante salidas y entradas serie. El transceptor, a través de las líneas CAN_H y CAN_L (ver Figura 3.5), proporciona transmisión y recepción diferencial. La transmisión diferencial es robusta ante interferencias electromagnéticas (EMI).

La capa MDI especifica las características del conector y del medio de transmisión. La norma ISO 11898-2 define una línea de dos cables con retorno común, terminada en ambos extremos por resistencias de igual valor a la impedancia característica de la línea. La topología del cableado debe ser similar a una estructura de línea simple para evitar reflexiones. La relación entre la velocidad de transmisión y la longitud máxima del bus se establece en la norma, según la Tabla 3.1, aunque estos valores son orientativos y varían dependiendo de la tolerancia de los osciladores de los nodos, las resistencias serie del cable del bus y de entrada de los nodos, y de los retardos en los nodos y la línea de bus. No obstante la norma recomienda una longitud máxima del bus sin repetidores de 1 km.

Además, CAN *in Automation* agrega características específicas para aplicaciones industriales, como el uso de conectores de tipo D de 9 pines y recomendaciones para el uso de dispositivos puente o repetidores en buses de mayor longitud. También sugiere el uso de optoacopladores para longitudes de buses superiores a 200 m.

Velocidad	Tiempo de bit	Longitud troncal máxima
1 Mb/s	1 μs	30 m
800 kb/s	1,25 μs	50 m
500 kb/s	2 μs	100 m
250 kb/s	4 μs	250 m
125 kb/s	8 μs	500 m
50 kb/s	20 μs	1000 m
20 kb/s	50 μs	2500 m
10 kb/s	100 μs	5000 m

Tabla 3.1: Velocidad de transmisión - longitud máxima en el bus CAN. Fuente: Comunicaciones industriales: Sistemas distribuidos y aplicaciones (2007).

Capítulo 4

Registro de datos en J8 del bus CAN

Contenido

Sinopsis

En este capítulo del proyecto se explicará en detalle cómo se ha realizado la conexión entre una computadora externa y el bus CAN del vehículo para la extracción de los mensajes enviados por las baterías, cargadores, motores y receptor de radio. También se analizará el protocolo utilizado para la extracción de dichos mensajes.

4.1. Tipos de bus CAN en el rover J8

Para entender la conexión entre la computadora de abordo y la red bus CAN del vehículo, y como se han extraido los mensajes, primero hay que explicar las redes CAN que se encuentran en el vehículo objeto de estudio.

El rover J8 cuenta con dos redes CAN conectadas a los puertos CAN0 y CAN1 de la computadora de abordo (ver Figura 4.1).

El receptor de radio está conectado al puerto CAN0, y trabaja a 250 Kbit/s, mientras que al puerto CAN1, configurado a 500 kbit/s, están conectados los dos motores, las baterías y los cargadores.

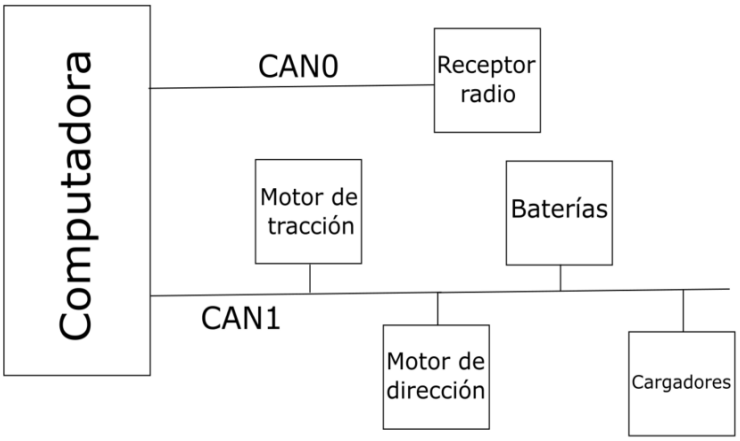


Figura 4.1: Esquema de las dos redes CAN de J8.

4.2. Conexión portatil-bus CAN

Para capturar los mensajes publicados en el bus con un portátil externo se han utilizado dos conversores USB2CAN (ver Figura 4.2), uno para cada red bus CAN de J8, con terminal USB (conectado al portátil externo) y el otro un conector DB9 macho.

Figura 4.2: Conversor USB2CAN Fuente: 8 devices (2023).

Este conversor, a su vez, esta conectado a un cable de tres hilos de elaboración propia con un terminal DB9 macho en un extremo. Por el otro extremo se conecta a la computadora de J8 con un DB9 hembra.

El proceso de conexión se realizará dos veces (una con cada conversor USB2CAN), uno para la conexión del portatil con la red CAN0 y otro para la conexión entre el portatil y la red CAN1 (ver Figura 4.3).

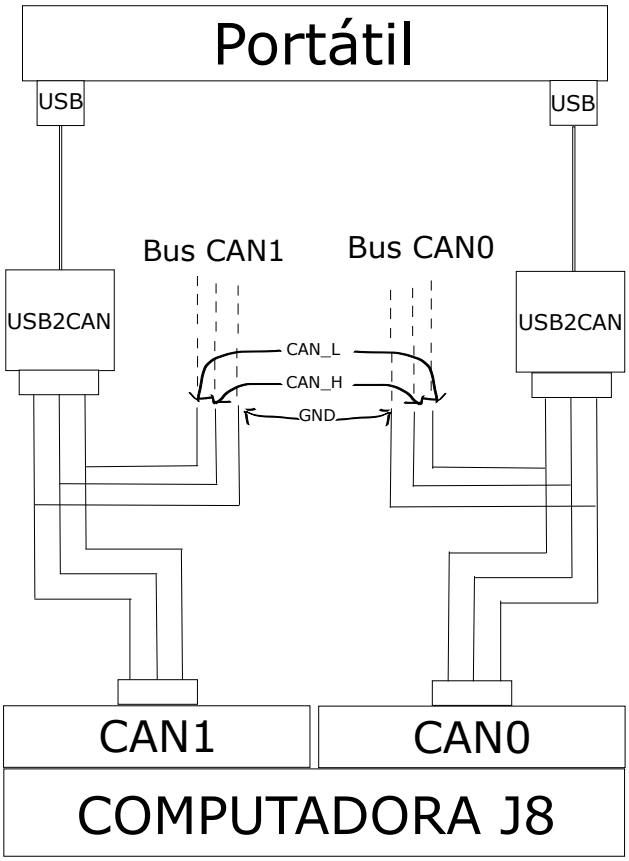


Figura 4.3: Esquema de la conexión del portátil externo con la computadora de J8 y los buses CAN.

De esta forma, se pueden interceptar los mensajes del bus sin llegar a interferir en su funcionamiento.

4.3. Recepción de los mensajes

En primer lugar, los mensajes publicados en el bus CAN se capturan mediante la conexión que hemos mencionado en el apartado 4.2.

A continuación se activan las conexiones CAN en el portátil, el cual tiene instalada la distribución de Linux Ubuntu 20.02 mediante los siguientes comandos:

```
sudo ip link set can0 type can bitrate 250000
```

```
sudo ip link set up can0
sudo ip link set can1 type can bitrate 500000
sudo ip link set up can1
```

Posteriormente se instala el paquete *CANutils* con el comando:

```
sudo apt install can-utils
```

Y se capturan los mensajes con:

```
candump -l can0
candump -l can1
```

CANdump es una herramienta dentro del conjunto de herramientas *CANutils* en Linux que se utiliza para capturar y mostrar tramas CAN (mensajes) que se transmiten en un bus CAN. Esta herramienta es útil para el análisis y la depuración de la comunicación en tiempo real en redes CAN.

La salida de *CANdump* proporciona información detallada sobre cada trama capturada. Incluye la identificación del mensaje (ID), la longitud de los datos, y los propios datos. Esta información se presenta en un formato legible que facilita el análisis (habrá que decodificarla posteriormente). La interfaz de usuario de *CANdump* es de línea de comandos, y su uso puede ser simple, con opciones para especificar el bus CAN y aplicar filtros.

Por otro lado, *CANutils* es un conjunto de herramientas que incluye, además de *CANdump*, otras utilidades para trabajar con redes CAN en Linux. Algunas de las herramientas más comunes incluyen:

- *CANgen*: Utilizado para generar tramas CAN de prueba para propósitos de prueba y desarrollo.
- *CANplayer*: Reproduce tramas CAN desde un archivo de registro, lo que facilita la reproducción de situaciones específicas para pruebas.
- *CANsend*: Permite enviar tramas CAN manualmente desde la línea de comandos.

Los mensajes de cada bus CAN se guardaran en ficheros distintos (*logs*), los de los motores, baterías y cargadores por un lado, y los del receptor de radio por otro. Estos ficheros se utilizarán posteriormente para la decodificación mediante Python a lenguaje humano (ver Capítulo 5).

Capítulo 5

Mensajes decodificados

Contenido

Sinopsis

En este capítulo se decodificarán y analizarán los mensajes enviados por las baterías, cargadores, el receptor radio y los motores de tracción y dirección, a través del bus CAN.

5.1. Decodificación de los mensajes

Los mensajes brutos (sin decodificar) enviados a través del bus CAN se dividen en ID CAN y bytes de datos, en hexadecimal. El ID CAN sirve para identificar de dónde viene el mensaje, ya sea del motor de dirección, de tracción, baterías, etc. Por otro lado, los bytes de datos contienen la información de dicho mensaje, que posteriormente tendrán que ser decodificados.

En el ejemplo de la Figura 5.1, extraído del archivo en el que estan guardados todos los mensajes enviados a traves del bus CAN del rover J8, se pueden discernir el ID CAN (antes de la almohadilla) y los bytes de datos (despues de la almohadilla).

```
      TIEMPO          BUS          MENSAJE

(1675939620.136822) can0 0CF10AF3#07FF0EFF340F0300
(1675939620.137045) can0 1A6#00009C9FFFFF6400
(1675939620.137192) can0 325#01
(1675939620.137445) can0 0CF10AF4#04FF0EFF340F0300
```

Figura 5.1: Mensajes brutos sin decodificar.

5.1.1. *Big endian y little endian*

Antes de entrar en materia sobre los archivos DBC y las reglas de decodificación, se explicará brevemente en qué consisten las formas *big endian* y *little endian* de organizar bytes en la memoria de una computadora.

Big endian y *little endian* son dos esquemas de ordenación de bytes en la memoria de una computadora que difieren en la disposición de bytes para datos multibyte, tales como palabras de 16 bits, enteros de 32 bits o números de punto flotante de 64 bits. La distinción fundamental reside en la secuencia en que estos bytes son almacenados en memoria (ver Figura 5.2).

En el caso de *big endian*, el byte más significativo se ubica en la dirección de memoria más baja, mientras que el byte menos significativo se sitúa en la dirección de memoria más alta. Este orden asemeja la escritura de números de izquierda a derecha.

Por otro lado, *little endian* dispone el byte menos significativo en la dirección de memoria más baja, y el byte más significativo en la dirección de memoria más alta. Este esquema se asemeja a la escritura de números de derecha a izquierda.

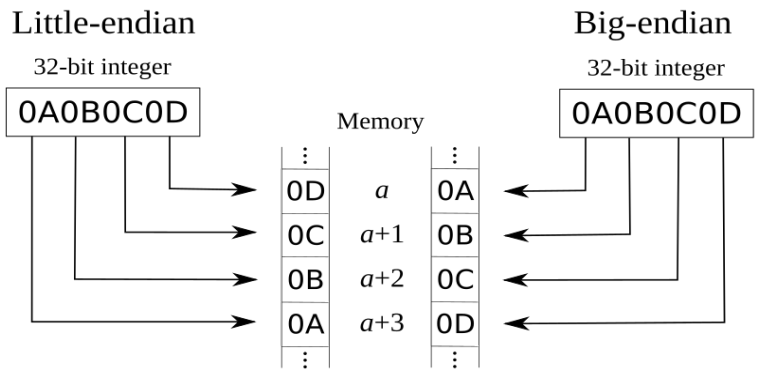


Figura 5.2: Ejemplo de ordenación de bytes *big endian y little endian*. Fuente: Agile Scientific (2017).

5.1.2. Archivo DBC y reglas de decodificación

Un archivo DBC (*Database for CAN*) es un archivo de texto que contiene la información necesaria para decodificar los mensajes brutos del bus CAN en valores físicos.

En dicho archivo viene especificado el ID CAN (en decimal) junto con el nombre de cada mensaje, por ejemplo, DRIVE_FB (*Driver controller feedback*). También, viene detallada información para decodificar los bytes de datos, como: el bit en el que empieza la información relevante, la longitud de esta (en bits), si es *little endian* o *big endian*, la escala, el *offset* y las unidades.

En la Figura 5.3 se muestra la especificación para la decodificación de los mensajes enviados por el motor de tracción, extraído del archivo DBC.

```
BO_ 421 DRIVE_FB: 8 Vector__XXX
        SG_ speed : 0|16@1- (1,0) [0|0] "" Vector__XXX
        SG_ position : 16|32@1- (1,0) [0|0] "" Vector__XXX
        SG_ FB3 : 48|16@1- (1,0) [0|0] "" Vector__XXX
```

Figura 5.3: Especificación del mensaje DRIVE_FB en el archivo DBC.

De la primera linea de texto podemos extraer información acerca del mensaje (BO_ hace referencia a que se trata de un mensaje), en ella podemos observar el ID CAN ("421", en decimal), el nombre del mensaje (DRIVE_FB) y la longitud del mensaje en bytes (8).

Cada mensaje puede tener una o mas señales (SG_), en el ejemplo de la Figura 5.3 tiene tres, una para velocidad, otra para posición y una ultima que en este estudio no será relevante. En estas señales es donde encontramos información sobre la decodificación de los bytes de datos. En el caso de la señal de velocidad, podemos encontrar el bit en el que empieza dicha señal (0), su longitud en bits (16) y si es *little endian* o *big endian* (@1 significa que es *little endian*), también se encuentra su escala y *offset* (1,0). En este caso, no se detalla ni el máximo y mínimo, ni las unidades.

Para los mensajes de los motores y del receptor de radio careciamos de archivo DBC, por lo que, con una serie de manuales de operador, en los que explicaban la estructura de estos mensajes, y con la ayuda de un editor DBC *online* de *CSS Electronics*, se crearon dichos archivos DBC.

Para los mensajes de las baterías y cargadores sí contabamos con un archivo DBC (el mismo para los dos), proporcionado por la marca fabricante *Brigs & Stratton*.

Editor DBC *online*

El funcionamiento del editor *online* de *CSS Electronics* es simple. En primer lugar se añaden tantos mensajes CAN como los que se quieran decodificar (ver Figura 5.4), a cada uno se le añade el identificador (ID) y se especifica si es hexadecimal o decimal, se selecciona si dicho mensaje es estandar o extendido, su longitud en bytes y se le asigna un nombre.

Dentro de cada mensaje se añaden tantas señales como las que se quieran decodificar (ver Figura 5.4). A estas señales habrá que indicarle su bit de inicio y su longitud en bits. Una vez está todo especificado, se puede ver una vista previa de cada mensaje para comprobar que todo es correcto.

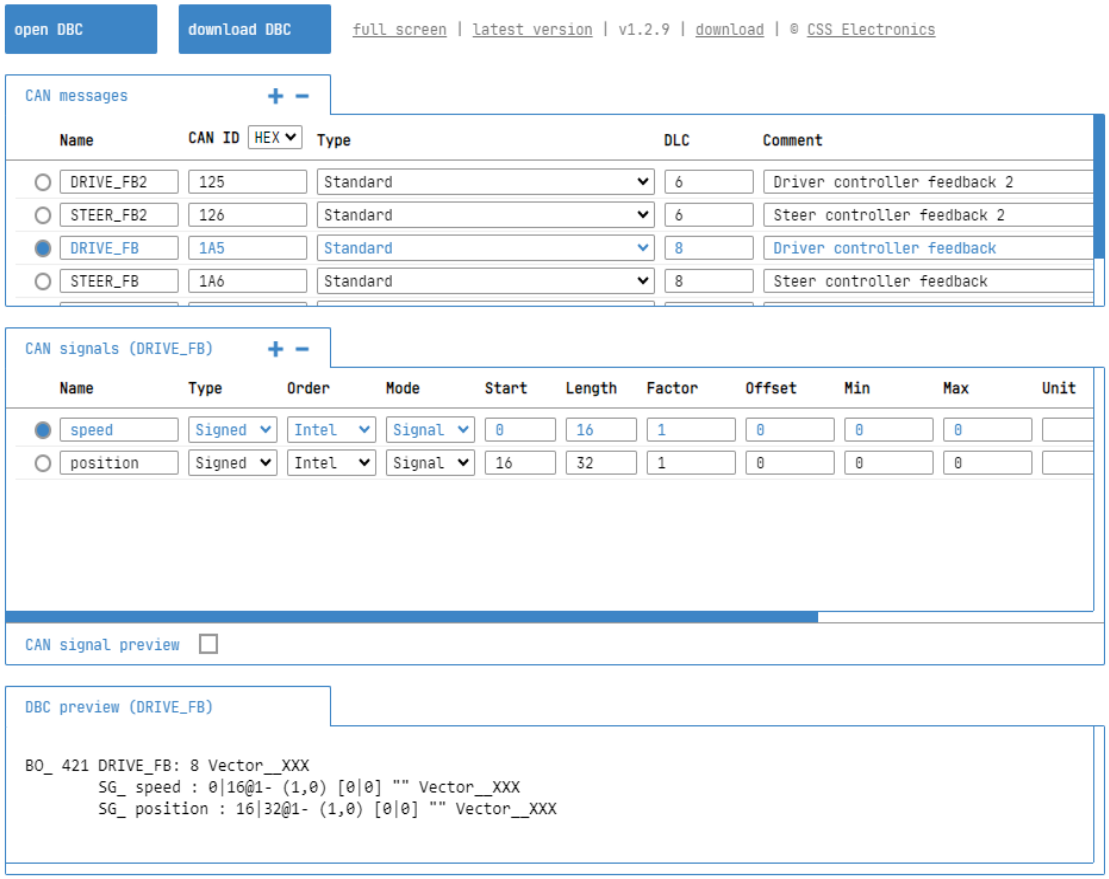


Figura 5.4: Editor DBC *online*. Fuente: *CSS Electronics* (2023).

5.1.3. Método de decodificación

Sabiendo todo lo anterior, ya podemos decodificar los mensajes brutos del bus CAN a valores físicos. El primer paso es buscar las señales que se van a decodificar en la batería de señales recogidas de J8, en nuestro caso, las de los motores de tracción y dirección, de las baterías, de los cargadores y del receptor de radio. Una vez identificada cada señal, habrá que fijarse en el archivo DBC para averiguar los bits que nos van a interesar de cada señal, y, si esta es *little endian* o *big endian*. En caso de ser *little endian*, habrá que reordenar la secuencia de los bytes que contengan la información. El resultado de esta reorganización (en hexadecimal) habrá que convertirlo a decimal (valor decimal bruto).

Conocida toda la información que nos concierne, se aplica la siguiente ecua-

ción para convertir la señal a valor físico:

$$Valor\_fisico = offset + escala * valor\_decimal\_bruto \tag{5.1}$$

Con este método se decodificaría una señal, pero, para muchas señales sería muy tedioso, por lo que para una gran cantidad de señales se utiliza software específico, que se comentará a continuación.

5.1.4. Software empleado

Para la decodificación de los mensajes se ha desarrollado un código en Python (ver Apéndice B), el cual lee el *log* en el que están almacenados todos los mensajes y el archivo DBC con las reglas de decodificación de cada una de los componentes que nos interesan (motores, baterías, cargadores y receptor de radio).

Posteriormente se hace un bucle que lea todos los mensajes del *log* de uno en uno. Como para cada componente solo nos interesan sus respectivos mensajes habrá que descartar todo aquel que no le pertenezca. Esto se hace comparando el ID (en hexadecimal) del mensaje que se esté leyendo en el momento con todos los ID (en hexadecimal) de los mensajes de dicho componente. En caso de coincidir con alguno, se lee el archivo DBC y se busca en este el ID (en decimal) para encontrar las reglas de decodificación de dicho mensaje (esto ultimo lo hace la herramienta *decode_message* del paquete *cantools*).

Los datos relevantes de cada mensaje, obtenidos con la herramienta *decode_message*, se guardaran en unas variables (inicializadas antes del bucle) para su posterior representación en gráficas (ver secciones 5.2 y 5.3).

La estructura de código mencionada sirve tanto para los motores como para las baterías, cargadores y receptor de radio. Tan solo habrá que cambiar en cada caso el *log*, el archivo DBC (baterías y cargadores usan el mismo diccionario de decodificación), los ID a comparar con el que se esté leyendo y las variables donde se guardara la información relevante para la posterior graficación.

5.1.5. Lista de mensajes descifrados

Además de DRIVE_FB (ver Figura 5.3), de los mensajes de los motores se han decodificado:

- DRIVE_SP (*Driver controller set point*).

- STEER_SP (*Steering controller set point*).
- STEER_FB (*Steering controller feedback*).

En cuanto a las baterías:

- F3_SOC (Estado de carga de la batería 1).
- F4_SOC (Estado de carga de la batería 2).
- F5_SOC (Estado de carga de la batería 3).
- F6_SOC (Estado de carga de la batería 4).
- F3_HVESSD1: En concreto las señales F3_HVESSD1_Current (intensidad de la batería 1) y F3_HVESSD1_Voltage (tensión de la batería 1).
- F4_HVESSD1: En concreto las señales F4_HVESSD1_Current (intensidad de la batería 2) y F4_HVESSD1_Voltage (tensión de la batería 2).
- F5_HVESSD1: En concreto las señales F5_HVESSD1_Current (intensidad de la batería 3) y F5_HVESSD1_Voltage (tensión de la batería 3).
- F6_HVESSD1: En concreto las señales F6_HVESSD1_Current (intensidad de la batería 4) y F6_HVESSD1_Voltage (tensión de la batería 4).

De los cargadores:

- F3_STATUS: En concreto la señal F3_STATUS_RunMode (modo de ejecución del cargador de la batería 1).
- F4_STATUS: En concreto la señal F4_STATUS_RunMode (modo de ejecución del cargador de la batería 2).
- F5_STATUS: En concreto la señal F5_STATUS_RunMode (modo de ejecución del cargador de la batería 3).
- F6_STATUS: En concreto la señal F6_STATUS_RunMode (modo de ejecución del cargador de la batería 4).

Y por último, del receptor de radio:

- ANALOG_CTRL (control analógico): En concreto las señales FWD_REV (hacia delante o hacia atrás) y LEFT_RIGHT (hacia la izquierda o hacia la derecha).
- DIGITAL_CTRL (control digital): En concreto las señales DK11 y DK12 (activar o desactivar los contactores), DK13 y DK14 (aumentar o disminuir la marcha del vehículo) y DK30 y DK31 (activar o desactivar el EStop).
- TXT_FB (*Text feedback*).

5.2. Mensajes del CAN1

5.2.1. Motores de tracción y dirección

Para la decodificación de los mensajes enviados por ambos motores se ha desarrollado el código Python mencionado en el apartado 5.1.4, el cual muestra una gráfica con la evolución de las velocidades de ambos motores con el tiempo (ver Figura 5.5).

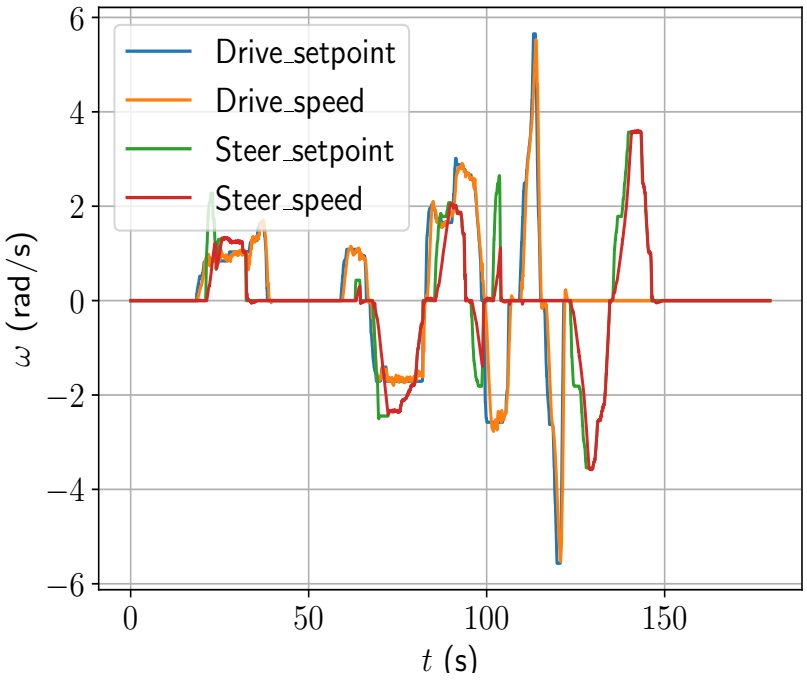


Figura 5.5: Evolución de la velocidad de los motores con el tiempo.

En la Figura 5.5 podemos observar cuatro señales. Dos de ellas son *Drive_setpoint* y *Steer_setpoint*, ambas son las consignas dadas al vehículo a traves del control remoto tanto para avanzar o retroceder como para girar, respectivamente. Las otras dos, *Drive_speed* y *Steer_speed*, muestran la velocidad real del motor de tracción y la del motor de dirección.

Se puede observar que las señales de velocidad de ambos motores van retrasadas con respecto a las señales *setpoint*, debido a que cada controladora implanta un perfil de velocidad trapeziodal con una aceleración máxima. Cuando las señales tienen valores positivos, significa que está desplazandose hacia detrás (*Drive_speed*) y hacia la izquierda (*Steer_speed*), si tienen valores negativos, hacia delante y derecha, respectivamente, y si tienen valor nulo, no se está moviendo.

Más adelante, cuando veamos la decodificación de los mensajes del receptor de radio, comprobaremos si las señales de las consignas (*Steer_setpoint* y *Drive_setpoint*) coinciden con los mensaje de la radio, ya que esta lee las consignas de tracción y dirección de los potenciómetros de los mandos.

5.2.2. Baterías

Con el mismo *log* utilizado para la sección anterior, el archivo DBC con las reglas de decodificación de los mensajes de las baterías y modificando el fichero Python utilizado se llevará a cabo la decodificación de los mensajes enviados por las baterías. En este apartado el enfoque será distinto, lo que nos interesará de las baterías será:

- El estado de carga de cada una de las baterías
- La tensión de cada una de las baterías.
- La corriente que consumen o es aportada a cada una de las baterías.

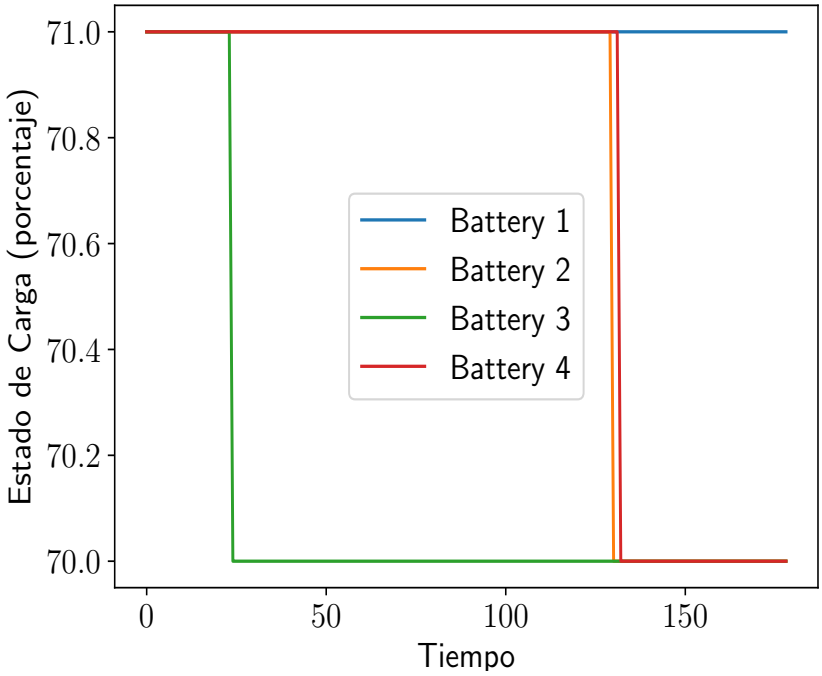


Figura 5.6: Evolución del estado de carga de las baterías con el tiempo.

Como se puede comprobar en la Figura 5.6, al inicio del experimento las cuatro baterías se encontraban en un 71 % del total de carga de la batería. Según avanza el experimento apreciamos que las baterías tres, dos y cuatro, en ese orden, se descargan hasta un 70 % del total de la batería, mientras que la uno se mantiene en 71 %.

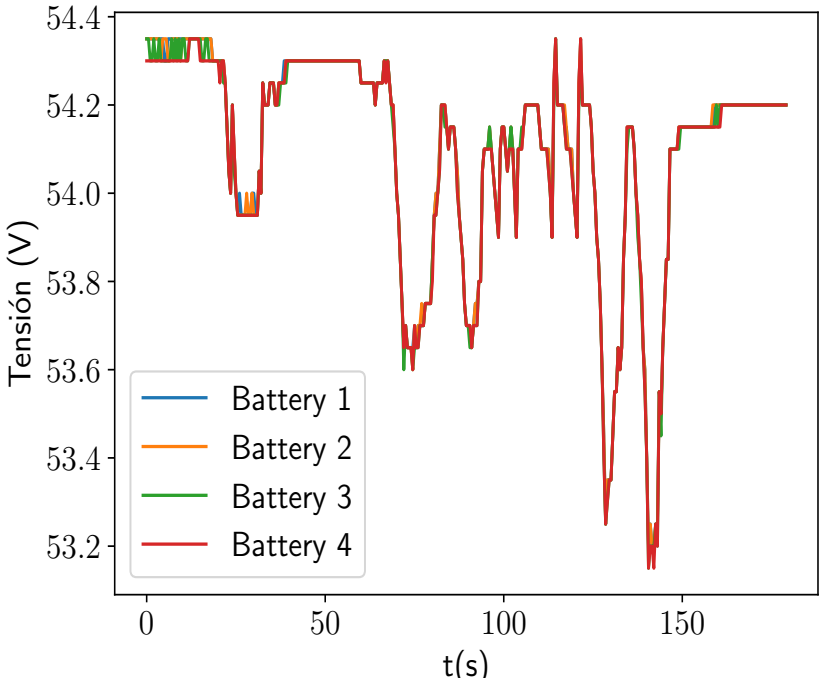


Figura 5.7: Evolución de la tensión de las baterías con el tiempo.

Se puede apreciar en la Figura 5.7 que la tensión en cada una de las baterías es practicamente la misma durante todo el experimento, y la variación es relativamente pequeña (54.3-53.2 V). Comparando esta gráfica con las de los motores, podemos observar que coinciden las variaciones de tensión con el desplazamiento del vehículo.

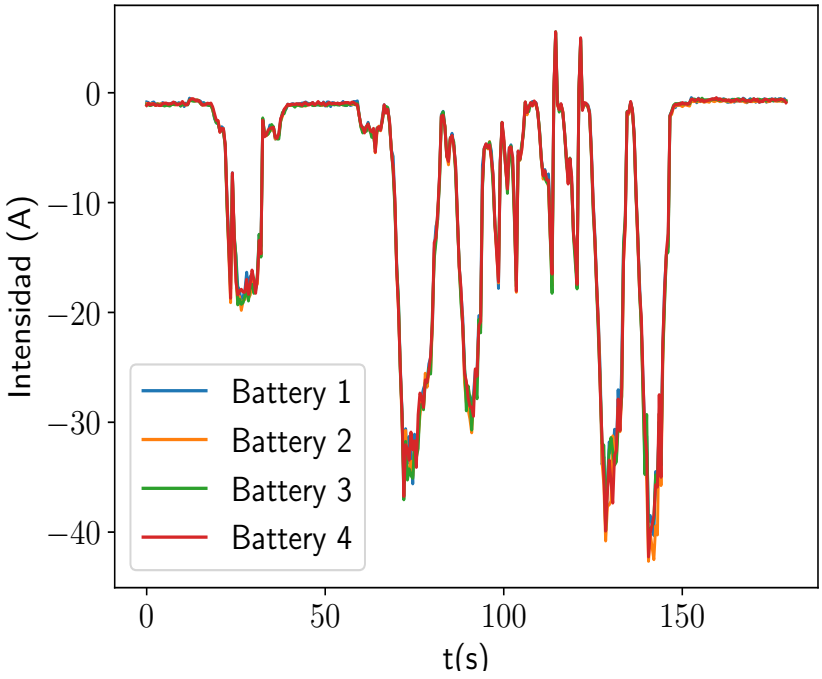


Figura 5.8: Evolución de la corriente que circula por cada batería con el tiempo.

Se observan valores de corriente negativos en la Figura 5.8, lo que nos indica que las baterías estan usando corriente durante el experimento, lo cual tiene sentido. De la misma forma que en la evolución de la tensión de las baterías, vemos que la variación de la corriente que consumen las baterías durante el experimento están directamente relacionado con el movimiento del vehículo.

5.2.3. Cargadores

Para la decodificación de los mensajes enviados por los cargadores se usará el mismo *log* que para los motores y las baterías. También, se utilizará el mismo archivo DBC que para las baterías, ya que las reglas de decodificación de los mensajes de los cargadores también estan incluidos en este.

En este caso estudiaremos los mensajes enviados por los cargadores los cuales nos indicarán en que estado se encuentran las baterías. En la sección 2.2.8 se explican los distintos estados que pueden experimentar las baterías.

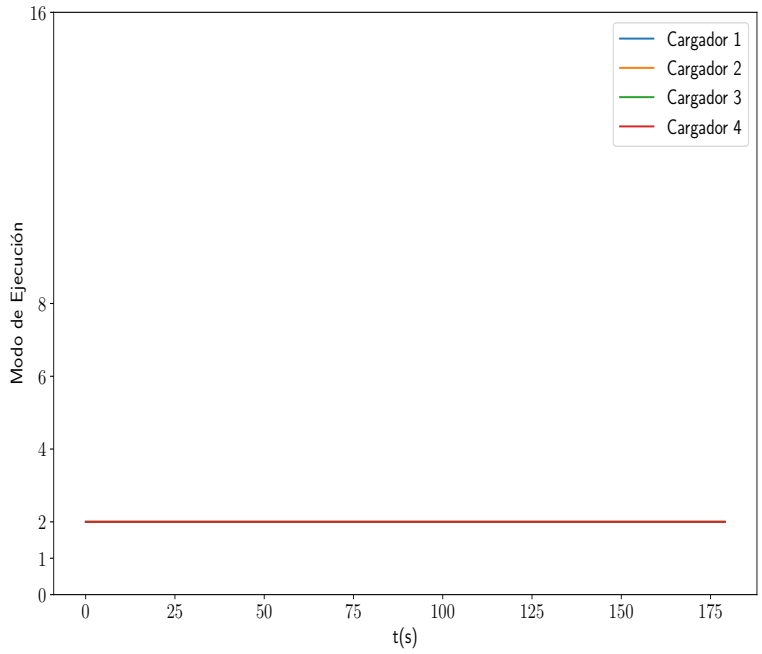


Figura 5.9: Estado de las baterías.

Los valores que aparecen en el eje Y (Modo de ejecución) de la Figura 5.9:

- **0:** *Sleep* (dormido)
- **1:** *Standby* (en espera)
- **2:** *Discharge* (descarga)
- **4:** *Charge* (carga)
- **6:** *Hybrid* (híbrido)
- **8:** *Briggs Command Mode*
- **16:** *Briggs Config. Mode*

Como se puede comprobar las baterías se mantienen en todo momento en estado *Discharge*, es decir, descargandose, lo cual es lógico, ya que durante todo el experimento se realizan acciones que consumen energía.

5.3. Mensajes del CAN0

El *log* de los mensajes enviados por el receptor de radio es distinto al de los motores, baterías y cargador, debido a que estos mensajes son enviados a traves de un bus CAN distinto (CAN1) al que envía los mensajes de la radio (CAN0).

Del receptor de radio llegan tres tipos de mensajes, unos con las señales de control analógicas, otros con las señales de control digitales y por último, otros con señales de *feedback* que aparecen en la pantalla LCD del control remoto *Hetronic*.

5.3.1. Señales de control analógicas

En estos mensajes lo único interensante para decodificar y representar son las señales de desplazamiento y giro enviados por el control remoto, las cuales se muestran en la Figura 5.10:

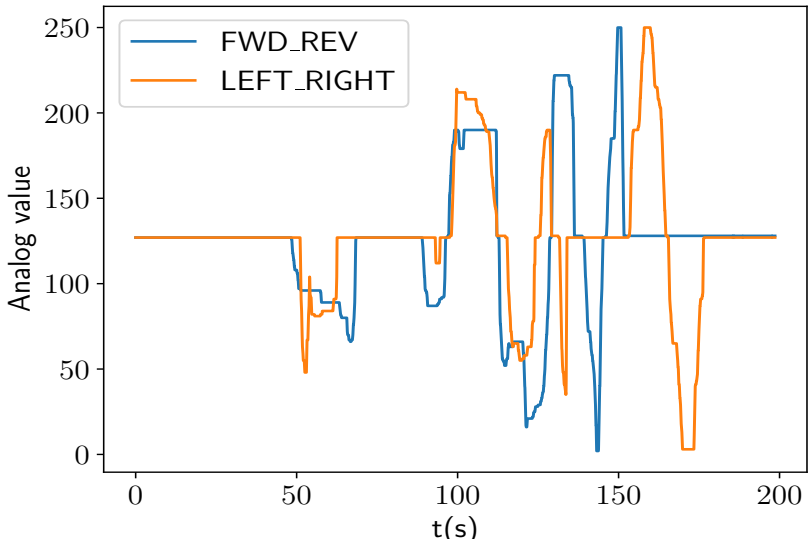


Figura 5.10: Señales de desplazamientos enviados por el receptor de radio.

La señal *FWD_REV* hace referencia al movimiento hacia delante (Forward) y hacia detras (*Reverse*), la señal *Left_Right*, al giro hacia la derecha (*Right*) o izquierda (*Left*). Si se comparan dichas señales con las de setpoint de am-

bos motores que encontramos en la Figura 5.5, comprobamos que coincide *Drive_setpoint* con *FWD_REV* y *Steer_setpoint* con *Left_Right*.

La única diferencia en esta última gráfica es que está invertida con respecto a la de los motores, lo que se debe a que, en la gráfica de los motores, valores negativos significan hacia delante o derecha, y valores positivos hacia detras o izquierda, y, en la gráfica del receptor radio, la dirección de la señal dependerá de hacia donde se esté moviendo el *joystick* encargado del desplazamiento y el *joystick* encargado del giro.

5.3.2. Señales de control digitales

Estos mensajes recogen las señales enviadas por el control remoto *Hetronic* al pulsar distintos botones en él. En la Figura 5.11 se muestra una representación del control remoto y sus botones junto a su nomenclatura:

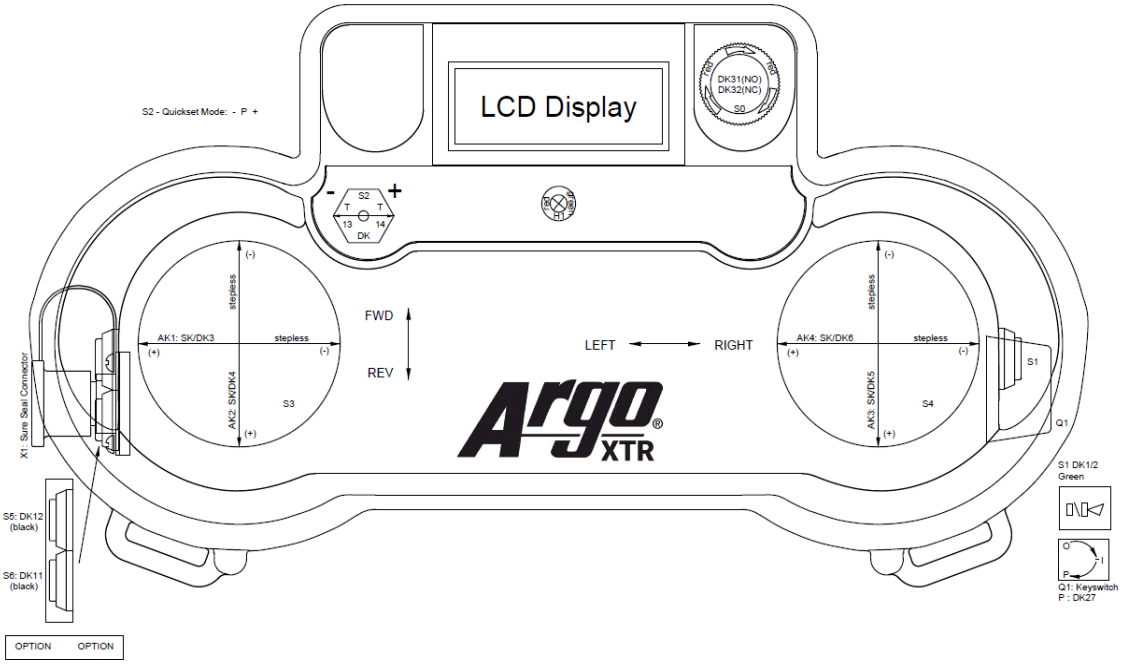


Figura 5.11: Representación del control remoto y de sus botones.

Las señales que decodificaremos en este subapartado serán las de los botones DK11 y DK12, los cuales se encuentran en la parte izquierda del mando (ver

Figura 5.15), y son los encargados de activar los contactores para darle potencia al motor y desactivar los frenos o de desactivar dichos contactores, respectivamente. También las señales DK13 y DK14, las cuales son las encargadas de aumentar o disminuir la marcha a la que se mueve el vehículo (ver Figura 5.16), respectivamente. Por último analizaremos las señales DK31 (NO) y DK32 (NC), las cuales hacen referencia al EStop (ver Figura 5.17), estas señales son antagonistas, pues NO significa "*Normally Opened*" y NC "*Normally Closed*", por lo tanto, si una está activada, la otra estará desactivada.

Al ser señales enviadas por botones, estas tendrán valores binarios según estos estén pulsados o no (ver Figuras 5.12, 5.13 y 5.14).

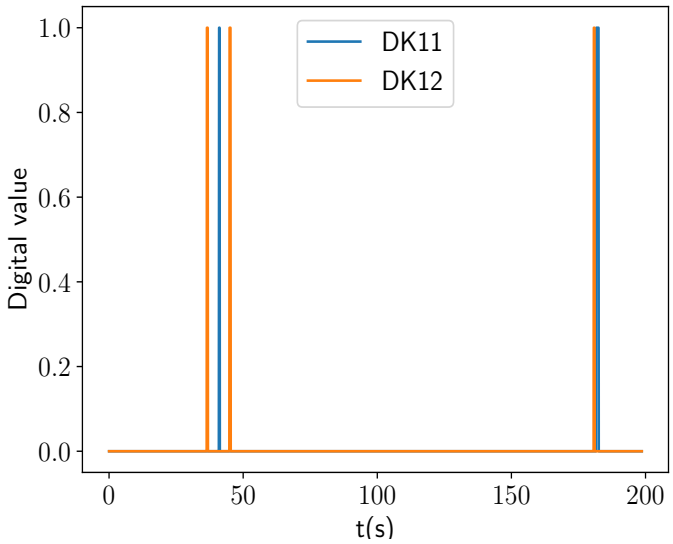


Figura 5.12: Señales binarias de activación/desactivación de los contactores.

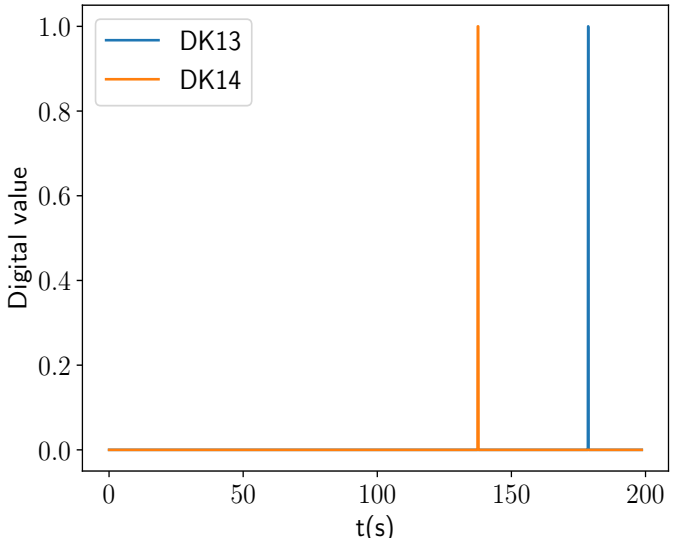


Figura 5.13: Señales del cambio de marchas.

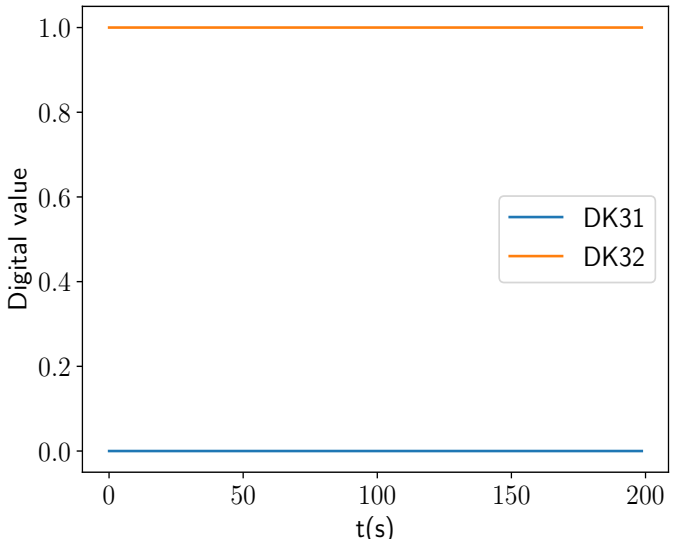


Figura 5.14: Señales de EStop.

En la Figura 5.14 se puede verificar que las dos señales del EStop son antagonistas. La que está activa durante todo el experimento es DK32 (*Normally*

Closed), lo que quiere decir que el botón de parada de emergencia no es pulsado en ningún momento.

Figura 5.15: Botones para las señales DK11 y DK12 encargados de la activación de los contactores.

Figura 5.16: Botón para las señales DK13 y DK14 encargado de subir y bajar la marcha.

Figura 5.17: Botón EStop del control remoto.

5.3.3. Señales de texto de *feedback*

Estos mensajes representan lo que aparecía por la pantalla LCD del control remoto *Hetronic* (ver Figura 5.21) durante el experimento. En las Figuras 5.18, 5.19 y 5.20 se muestran varios extractos de dichos mensajes con información relevante.

Mensaje 2: Linea: 1.0, Columna: 1.0, Mensaje: SOC:__■■
Mensaje 3: Linea: 1.0, Columna: 7.0, Mensaje: 71____■■
Mensaje 4: Linea: 2.0, Columna: 1.0, Mensaje: VLTG:_■■
Mensaje 5: Linea: 2.0, Columna: 7.0, Mensaje: 54.3__■■
Mensaje 6: Linea: 3.0, Columna: 1.0, Mensaje: CTOR:_■■
Mensaje 7: Linea: 3.0, Columna: 7.0, Mensaje: ON____■■
Mensaje 8: Linea: 4.0, Columna: 1.0, Mensaje: SPD:__■■
Mensaje 9: Linea: 4.0, Columna: 7.0, Mensaje: 0_____■■

Figura 5.18: Extracto 1 de los mensajes de *feedback*.

Mensaje 1826: Linea: 1.0, Columna: 1.0, Mensaje: SOC:__■■
Mensaje 1827: Linea: 1.0, Columna: 7.0, Mensaje: 70____■■
Mensaje 1828: Linea: 2.0, Columna: 1.0, Mensaje: VLTG:_■■
Mensaje 1829: Linea: 2.0, Columna: 7.0, Mensaje: 54.2__■■
Mensaje 1830: Linea: 3.0, Columna: 1.0, Mensaje: CTOR:_■■
Mensaje 1831: Linea: 3.0, Columna: 7.0, Mensaje: ON____■■
Mensaje 1832: Linea: 4.0, Columna: 1.0, Mensaje: SPD:__■■
Mensaje 1833: Linea: 4.0, Columna: 7.0, Mensaje: 0_____■■

Figura 5.19: Extracto 2 de los mensajes de *feedback*.

Mensaje 1858: Linea: 1.0, Columna: 1.0, Mensaje: SOC:__■■
Mensaje 1859: Linea: 1.0, Columna: 7.0, Mensaje: 70____■■
Mensaje 1860: Linea: 2.0, Columna: 1.0, Mensaje: VLTG:_■■
Mensaje 1861: Linea: 2.0, Columna: 7.0, Mensaje: 54.2__■■
Mensaje 1862: Linea: 3.0, Columna: 1.0, Mensaje: CTOR:_■■
Mensaje 1863: Linea: 3.0, Columna: 7.0, Mensaje: ON____■■
Mensaje 1864: Linea: 4.0, Columna: 1.0, Mensaje: SPD:__■■
Mensaje 1865: Linea: 4.0, Columna: 7.0, Mensaje: 1_____■■

Figura 5.20: Extracto 3 de los mensajes de *feedback*.

Podemos observar que hay cuatro tipos de señales: SOC (*State of charge*), VLTG (*Voltage*), CTOR (Contactores) y SPD (*speed*, hace referencia a la velocidad de las marchas).

En cuanto al estado de carga, en los extractos 1 y 2 podemos comprobar que las baterías se encuentran al 71 % del total de su carga al principio del experimento, y baja al 70 % conforme avanza el tiempo, lo cual concuerda con la Figura 5.6 en la que se muestra la evolución del estado de carga de las baterías.

Fijandonos en los mensajes VLTG, vemos que estos varían entre 54.2 V y 54.3 V. Esto se encuentra en concordancia con la Figura 5.7, en la que se ve que la tensión de las baterías varía entre 54.4 V y 53.2 V durante el experimento.

Los contactores (CTOR) se encuentran activados (*ON*) en todo momento, y esto es porque deben estarlo para que el vehículo se pueda mover. Se puede comprobar comparandolo con la Figura 5.12, en la que vemos como en un corto periodo de tiempo se activan, desactivan y vuelven a activar los contactores, quedando activados hasta practicamente el final del experimento.

Por último, se puede apreciar como en un principio SPD (marchas) se encuentra a 0, y más tarde pasa a ser 1, esto se debe a que en un principio el vehículo no se mueve, y más tarde sí. Esta información concuerda con la obtenida de la Figura 5.13, en la que observamos que no se aumenta la marcha hasta casi el final del experimento.

Figura 5.21: Pantalla LCD del control remoto.

Capítulo 6

Conclusiones y trabajos futuros

Contenido

6.1. Recapitulación

Se nos presentaba un problema inicial con el rover J8, puesto que se trata de un sistema cerrado, siendo imposible conocer su comportamiento. El objetivo del departamento es convertir dicho vehículo en uno autónomo, que se pudiera controlar desde una computadora externa. Para ello, primero es necesario conocer el comportamiento del vehículo, de sus motores (tracción y dirección), de sus baterías y cargadores, y de su receptor de radio.

A través de dos conversores USB2CAN se consiguió conectar el portatil externo a las dos redes CAN del vehículo (CAN0 y CAN1), y mediante la herramienta *candump* del paquete *canutils* de Linux se almacenaron los mensajes publicados en dichas redes en distintos archivos (*logs*). Estos *logs* se han utilizado posteriormente, junto a unos diccionarios de conversión (archivos DBC), para decodificar los mensajes importantes para conocer el comportamiento del vehículo.

6.2. Aprendizaje

Gracias a los experimentos que se han llevado a cabo se ha podido conocer la estructura de comunicación de J8 y como interpretar los mensajes que nos eran de utilidad. También se ha aprendido a generar diccionarios de decodificación (archivos DBC), a cómo interpretarlos y a utilizarlos en un código Python. Por último, se ha entendido cómo almacenar los mensajes extraidos de las redes CAN en distintos archivos mediante Linux y la herramienta *candump*.

6.3. Trabajos futuros

Al haber cumplido los objetivos que se proponían en un principio, el rover J8 podrá ser controlado por una computadora externa (actuando como control remoto) y se conocerán sus medidas en todo momento.

Como trabajo futuro se puede plantear el desarrollo de una interfaz ROS (*Robot Operating System*) para hacer accesible la información sensorial del robot así como para poder darle consignas de movimiento.

Apéndice A

Diccionarios de decodificación

Contenido

Sinopsis

En este apéndice se incluyen los archivos DBC (Diccionarios de decodificación) utilizados en el proyecto. En el caso del diccionario de las baterías y cargadores, debido a su gran extensión, solo se mostrarán las reglas de decodificación de los mensajes que hemos descifrado.

A.1. Motores

VERSION ””

NS_ :

NS_DESC_
CM_
BA_DEF_
BA_
VAL_
CAT_DEF_
CAT_
FILTER
BA_DEF_DEF_
EV_DATA_
ENVVAR_DATA_
SGTYPE_
SGTYPE_VAL_
BA_DEF_SGTYPE_
BA_SGTYPE_
SIG_TYPE_REF_
VAL_TABLE_
SIG_GROUP_
SIG_VALTYPE_
SIGTYPE_VALTYPE_
BO_TX_BU_
BA_DEF_REL_
BA_REL_
BA_DEF_DEF_REL_
BU_SG_REL_
BU_EV_REL_
BU_BO_REL_
SG_MUL_VAL_

BS_:

BU_:

BO_ 933 DRIVE_SP: 2 Vector__XXX

```
    SG_ speed_sp : 0|16@1- (1,0) [0|31402] çmd"Vector__XXX

BO_ 934 STEER_SP: 2 Vector__XXX
    SG_ speed_sp : 0|16@1- (1,0) [0|31402] çmd"Vector__XXX

BO_ 421 DRIVE_FB: 8 Vector__XXX
    SG_ speed : 0|16@1- (1,0) [0|0]  Vector__XXX
    SG_ position : 16|32@1- (1,0) [0|0]  Vector__XXX
    SG_ FB3 : 48|16@1- (1,0) [0|0]  Vector__XXX

BO_ 293 DRIVE_FB2: 6 Vector__XXX
    SG_ A : 0|8@1- (1,0) [0|0]  Vector__XXX
    SG_ B : 24|24@1- (1,0) [0|0]  Vector__XXX

BO_ 422 STEER_FB: 8 Vector__XXX
    SG_ speed : 0|16@1- (1,0) [0|0]  Vector__XXX
    SG_ position : 16|32@1- (1,0) [0|0]  Vector__XXX
    SG_ FB3 : 48|16@1- (1,0) [0|0]  Vector__XXX

BO_ 294 STEER_FB2: 6 Vector__XXX
    SG_ A : 0|8@1- (1,0) [0|0]  Vector__XXX
    SG_ B : 24|24@1- (1,0) [0|0]  Vector__XXX

CM_ BO_ 933 "Driver controller set point";
CM_ SG_ 933 speed_sp "Driver speed set point";
CM_ BO_ 934 "Steering controller set point";
CM_ SG_ 934 speed_sp "Steer speed set point";
CM_ BO_ 421 "Driver controller feedback";
CM_ BO_ 422 "Steer controller feedback";
CM_ BO_ 293 "Driver controller feedback 2";
CM_ BO_ 294 "Steer controller feedback 2";

BA_DEF_ SG_ "SPNÏNT 0 524287;
BA_DEF_ BO_ "VFrameFormat" ENUM "StandardCAN",.ExtendedCAN",reserved","J1939PG";
BA_DEF_ "DatabaseVersion"STRING ;
BA_DEF_ "BusType"STRING ;
BA_DEF_ "ProtocolType"STRING ;
BA_DEF_ "DatabaseCompiler"STRING ;
```

```
BA_DEF_DEF_ "SPN"0;
BA_DEF_DEF_ "VFrameFormatJ1939PG";
BA_DEF_DEF_ "DatabaseVersionDEMO PLUS";
BA_DEF_DEF_ "BusType";
BA_DEF_DEF_ "ProtocolType";
BA_DEF_DEF_ "DatabaseCompiler";
BA_ "ProtocolTypeJ1939";
BA_ "BusTypeCAN";
BA_ "DatabaseCompilerCSS ELECTRONICS (WWW.CSSELECTRONICS.COM)";
BA_ "DatabaseVersion1.0.0";
```

A.2. Baterías y cargadores

```
VERSION ""

NS_ :
    NS_DESC_
    CM_
    BA_DEF_
    BA_
    VAL_
    CAT_DEF_
    CAT_
    FILTER
    BA_DEF_DEF_
    EV_DATA_
    ENVVAR_DATA_
    SGTYPE_
    SGTYPE_VAL_
    BA_DEF_SGTYPE_
    BA_SGTYPE_
    SIG_TYPE_REF_
    VAL_TABLE_
    SIG_GROUP_
    SIG_VALTYPE_
    SIGTYPE_VALTYPE_
    BO_TX_BU_
    BA_DEF_REL_
    BA_REL_
    BA_DEF_DEF_REL_
    BU_SG_REL_
    BU_EV_REL_
    BU_BO_REL_
    SG_MUL_VAL_

BS_:

BU_: BMS_F3 CHARGER BMS_F4 BMS_F5 BMS_F6 BMS_x80 Charger BMS_F3
VCM

BO_ 2365523699 F3_SOC: 8 BMS_F3
```

```
    SG_ F3_SOC : 0|16@1+ (1,0) [0|0]  Vector__XXX

BO_ 2364575987 F3_HVESSD1: 8 BMS_F3
    SG_  F3_HVESSD1_Discharge_Power_Lim  :  0|16@1+  (0.05,0)  [0|3212.75]
"kW"Vector__XXX
    SG_ F3_HVESSD1_Regen_Power_Lim : 16|16@1+ (0.05,0) [0|3212.75] "kW"Vector__XXX
    SG_ F3_HVESSD1_Current : 48|16@1+ (0.05,-1600) [-1600|1612.75] ”Amps"Vector__XXX
    SG_ F3_HVESSD1_Voltage : 32|16@1+ (0.05,0) [0|3212.75] "Volts"Vector__XXX

BO_ 2365523700 F4_SOC: 8 BMS_F4
    SG_ F4_SOC : 0|16@1+ (1,0) [0|0]  Vector__XXX

BO_ 2364575988 F4_HVESSD1: 8 BMS_F4
    SG_  F4_HVESSD1_Discharge_Power_Lim  :  0|16@1+  (0.05,0)  [0|3212.75]
"kW"Vector__XXX
    SG_ F4_HVESSD1_Regen_Power_Lim : 16|16@1+ (0.05,0) [0|3212.75] "kW"Vector__XXX
    SG_ F4_HVESSD1_Current : 48|16@1+ (0.05,-1600) [-1600|1612.75] ”Amps"Vector__XXX
    SG_ F4_HVESSD1_Voltage : 32|16@1+ (0.05,0) [0|3212.75] "Volts"Vector__XXX

BO_ 2365523701 F5_SOC: 8 BMS_F5
    SG_ F5_SOC : 0|16@1+ (1,0) [0|0]  Vector__XXX

BO_ 2364575989 F5_HVESSD1: 8 BMS_F5
    SG_  F5_HVESSD1_Discharge_Power_Lim  :  0|16@1+  (0.05,0)  [0|3212.75]
"kW"Vector__XXX
    SG_ F5_HVESSD1_Regen_Power_Lim : 16|16@1+ (0.05,0) [0|3212.75] "kW"Vector__XXX
    SG_ F5_HVESSD1_Current : 48|16@1+ (0.05,-1600) [-1600|1612.75] ”Amps"Vector__XXX
    SG_ F5_HVESSD1_Voltage : 32|16@1+ (0.05,0) [0|3212.75] "Volts"Vector__XXX

BO_ 2365523702 F6_SOC: 8 BMS_F6
    SG_ F6_SOC : 0|16@1+ (1,0) [0|0]  Vector__XXX

BO_ 2364575990 F6_HVESSD1: 8 BMS_F6
    SG_  F6_HVESSD1_Discharge_Power_Lim  :  0|16@1+  (0.05,0)  [0|3212.75]
"kW"Vector__XXX
    SG_ F6_HVESSD1_Regen_Power_Lim : 16|16@1+ (0.05,0) [0|3212.75] "kW"Vector__XXX
    SG_ F6_HVESSD1_Current : 48|16@1+ (0.05,-1600) [-1600|1612.75] ”Amps"Vector__XXX
    SG_ F6_HVESSD1_Voltage : 32|16@1+ (0.05,0) [0|3212.75] "Volts"Vector__XXX
```

```
BO_ 2566850803 F3_STATUS: 8 BMS_F3
    SG_ F3_STATUS_RunMode : 0|8@1+ (1,0) [0|0] "" Vector__XXX

BO_ 2566850804 F4_STATUS: 8 BMS_F4
    SG_ F4_STATUS_RunMode : 0|8@1+ (1,0) [0|0] "" Vector__XXX

BO_ 2566850805 F5_STATUS: 8 BMS_F5
    SG_ F5_STATUS_RunMode : 0|8@1+ (1,0) [0|0] "" Vector__XXX

BO_ 2566850806 F6_STATUS: 8 BMS_F6
    SG_ F6_STATUS_RunMode : 0|8@1+ (1,0) [0|0] "" Vector__XXX
```

A.3. Receptor de radio

```
VERSION ""

NS_ :
    NS_DESC_
    CM_
    BA_DEF_
    BA_
    VAL_
    CAT_DEF_
    CAT_
    FILTER
    BA_DEF_DEF_
    EV_DATA_
    ENVVAR_DATA_
    SGTYPE_
    SGTYPE_VAL_
    BA_DEF_SGTYPE_
    BA_SGTYPE_
    SIG_TYPE_REF_
    VAL_TABLE_
    SIG_GROUP_
    SIG_VALTYPE_
```

```
    SIGTYPE_VALTYPE_
    BO_TX_BU_
    BA_DEF_REL_
    BA_REL_
    BA_DEF_DEF_REL_
    BU_SG_REL_
    BU_EV_REL_
    BU_BO_REL_
    SG_MUL_VAL_

BS_:

BU_:

BO_ 484 ANALOG_CTRL: 8 Vector__XXX
    SG_ AK1 : 0|8@1+ (1,0) [0|0]  Vector__XXX
    SG_ FWD_REV : 8|8@1+ (1,0) [0|0]  Vector__XXX
    SG_ AK3 : 16|8@1+ (1,0) [0|0]  Vector__XXX
    SG_ LEFT_RIGHT : 24|8@1+ (1,0) [0|0]  Vector__XXX
    SG_ AK5 : 32|8@1+ (1,0) [0|0]  Vector__XXX
    SG_ AK6 : 40|8@1+ (1,0) [0|0]  Vector__XXX
    SG_ AK7 : 48|8@1+ (1,0) [0|0]  Vector__XXX
    SG_ AK8 : 56|8@1+ (1,0) [0|0]  Vector__XXX

BO_ 740 DIGITAL_CTRL: 8 Vector__XXX
    SG_ DK31_Estop_NO_ContactBit : 0|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK2_Horn : 1|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK3_AK1_SafetyBit : 2|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK4_FWD_REV_SafetyBit : 3|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK5_AK3_SafetBit : 4|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK6_LEFT_RIGHT_SafetyBit : 5|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK7 : 6|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK8 : 7|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK9 : 8|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK10 : 9|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK11_Option : 10|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK12_Option : 11|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK13 : 12|1@1+ (1,0) [0|0]  Vector__XXX
    SG_ DK14 : 13|1@1+ (1,0) [0|0]  Vector__XXX
```

```
 SG_ DK15 : 14|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK16 : 15|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK17 : 16|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK18 : 17|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK19 : 18|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK20 : 19|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK21 : 20|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK22 : 21|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK23 : 22|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK24 : 23|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK25 : 24|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK26 : 25|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK27 : 26|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK28 : 27|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK29 : 28|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK30 : 29|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK1_Start : 30|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ DK32_Estop_NC_ContactBit : 31|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ RES1 : 32|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ RES2 : 40|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ RES3 : 48|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ Estop_Transmission : 56|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ No_Data_Receiving : 57|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ Receiving_QualityBit1 : 58|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ Receiving_QualityBit2 : 59|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ Joystick_ErrorBit_1 : 60|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ Joystick_ErrorBit_2 : 61|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ Joystick_ErrorBit_3 : 62|1@1+ (1,0) [0|0]  Vector__XXX
 SG_ Joystick_ErrorBit_4 : 63|1@1+ (1,0) [0|0]  Vector__XXX

BO_ 868 TXT_FB: 8 Vector__XXX
 SG_ line : 0|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ column : 8|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ CHAR1 : 16|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ CHAR2 : 24|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ CHAR3 : 32|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ CHAR4 : 40|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ CHAR5 : 48|8@1+ (1,0) [0|0]  Vector__XXX
 SG_ CHAR6 : 56|8@1+ (1,0) [0|0]  Vector__XXX
```

CM_ BO_ 484 ”Analog control signal";
CM_ SG_ 484 AK1 ”Actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders.â€¦";
CM_ BO_ 740 "Digital control signal";
CM_ SG_ 740 DK31_Estop_NO_ContactBit "Wheel-Based Vehicle Speed: Speed of the vehicle as calculated from wheel or tailshaft speed.";
CM_ BO_ 868 "Text feedback signal";
BA_DEF_ SG_ "SPN” INT 0 524287;
BA_DEF_ BO_ "VFrameFormat” ENUM "StandardCAN”,”ExtendedCAN",”reserved","J1939PG";
BA_DEF_ "DatabaseVersion"STRING ;
BA_DEF_ "BusType"STRING ;
BA_DEF_ "ProtocolType"STRING ;
BA_DEF_ "DatabaseCompiler"STRING ;
BA_DEF_DEF_ "SPN"0;
BA_DEF_DEF_ "VFrameFormatJ1939PG";
BA_DEF_DEF_ "DatabaseVersionDEMO PLUS";
BA_DEF_DEF_ "BusType";
BA_DEF_DEF_ "ProtocolType";
BA_DEF_DEF_ "DatabaseCompiler";
BA_ "ProtocolTypeJ1939";
BA_ "BusTypeCAN";
BA_ "DatabaseCompilerCSS ELECTRONICS (WWW.CSSELECTRONICS.COM)";
BA_ "DatabaseVersion1.0.0";
BA_ "VFrameFormat"BO_ 2364540158 3;
BA_ "VFrameFormat"BO_ 2566844926 3;
BA_ "SPN"SG_ 2364540158 EngineSpeed 190;
BA_"SPN"SG_ 2566844926 WheelBasedVehicleSpeed 84;

Apéndice B

Códigos para la decodificación de mensajes

Contenido

Sinopsis

En este apéndice se incluyen los códigos utilizados en Python para la decodificación de los mensajes. Se han utilizado cuatro códigos distintos, pero similares en estructura, para los mensajes de los motores, de las baterías, de los cargadores y del receptor de radio.

B.1. Motores

```
import cantools
from pprint import pprint
import can
import numpy as np
import
matplotlib.pyplot as mp

mp.rcParams.update({
    "text.usetex": True,

"font.family": "sans-serif",
    "font.size": 16
})

exp =
'motor_2'
logfile = '/home/alvarocardona/Desktop/TFG/decodificacion/can1/' + exp +
'.log'

dicfile =
'/home/alvarocardona/Desktop/TFG/decodificacion/can1/Motores/file2.dbc'
N=6000

t_dsp  =
np.zeros(N)
i_dsp = 0
dsp = np.zeros(N)

t_ssp  = np.zeros(N)
i_ssp = 0
ssp =
np.zeros(N)

t_dpos  = np.zeros(N)
i_dpos = 0
dpos = np.zeros(N)
dspeed = np.zeros(N)
dfb3  =
np.zeros(N)

t_spos  = np.zeros(N)
i_spos = 0
spos = np.zeros(N)
sspeed = np.zeros(N)
sfb3  =
np.zeros(N)

t_dFB2 = np.zeros(N)
i_dFB2 = 0
dA = np.zeros(N)
dB = np.zeros(N)

db =
cantools.database.load_file(dicfile)

with can.CanutilsLogReader(logfile) as can_log:
    for
msg in can_log:
        if(msg.arbitration_id==0x3A5):
            t_dsp[i_dsp]=msg.timestamp

          dmesg=db.decode_message(msg.arbitration_id, msg.data)

dsp[i_dsp]=dmesg['speed_sp']
            i_dsp=i_dsp+1

elif(msg.arbitration_id==0x3A6):
            t_ssp[i_ssp]=msg.timestamp

dmesg=db.decode_message(msg.arbitration_id, msg.data)
            ssp[i_ssp]=dmesg['speed_sp']

           i_ssp=i_ssp+1
        elif(msg.arbitration_id==0x1A5):
```

```
t_dpos[i_dpos]=msg.timestamp
            dmesg=db.decode_message(msg.arbitration_id, msg.data)

           dpos[i_dpos]=dmesg['position']
            dspeed[i_dpos]=dmesg['speed']

i_dpos=i_dpos+1
        elif(msg.arbitration_id==0x1A6):

t_spos[i_spos]=msg.timestamp
            dmesg=db.decode_message(msg.arbitration_id, msg.data)

           spos[i_spos]=dmesg['position']
            sspeed[i_spos]=dmesg['speed']

i_spos=i_spos+1

# %% Set-point/Speed
ratio
mp.figure()
mp.plot(t_dsp[0:i_dsp]-t_dsp[0],dsp[0:i_dsp]/1355*0.95,t_speed_d,
speed_d,t_ssp[0:i_ssp]-t_ssp[0],ssp[0:i_ssp]/5960*0.95,t_speed_s,speed_s)
mp.legend((r'Drive\_s
etpoint',r'Drive\_speed',r'Steer\_setpoint',r'Steer\_speed'))
mp.ylabel(r'$\omega$
(rad/s)')
mp.xlabel(r'$t$ (s)')
mp.grid('on')
mp.savefig('drive_steer_speeds_sp.pdf')
```

B.2. Baterías

```
import cantools
from pprint import pprint
import can
import numpy as np
import
matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import
PdfPages

plt.rcParams.update({
    "text.usetex": True,
    "font.family":
"sans-serif",
    "font.size": 16
})

exp = 'motor_2'
logfile =
'/home/alvarocardona/Desktop/TFG/decodificacion/can1/' + exp + '.log'

dicfile =
'/home/alvarocardona/Desktop/TFG/decodificacion/can1/Baterias/Universal_BMS_FW_2_2+_5BAT_ext.db
c'
db = cantools.database.load_file(dicfile)

N = 6000

t_B1_SOC = np.zeros(N)
i_B1_SOC =
0
B1_SOC = np.zeros(N)

t_B2_SOC = np.zeros(N)
i_B2_SOC = 0
B2_SOC = np.zeros(N)

t_B3_SOC =
np.zeros(N)
i_B3_SOC = 0
B3_SOC = np.zeros(N)

t_B4_SOC = np.zeros(N)
i_B4_SOC = 0
B4_SOC =
np.zeros(N)

t_B1_VC = np.zeros(N)
i_B1_VC = 0
B1_V = np.zeros(N)
B1_C = np.zeros(N)

t_B2_VC =
np.zeros(N)
i_B2_VC = 0
B2_V = np.zeros(N)
B2_C = np.zeros(N)

t_B3_VC = np.zeros(N)
i_B3_VC =
0
B3_V = np.zeros(N)
B3_C = np.zeros(N)

t_B4_VC = np.zeros(N)
i_B4_VC = 0
B4_V =
np.zeros(N)
B4_C = np.zeros(N)

with can.CanutilsLogReader(logfile) as can_log:
    for msg in
can_log:
        if(msg.arbitration_id == 0x0CFF06F3):
            t_B1_SOC[i_B1_SOC] =
```

```
msg.timestamp
            dmesg = db.decode_message(msg.arbitration_id, msg.data)

B1_SOC[i_B1_SOC] = dmesg['F3_SOC']
            i_B1_SOC += 1
        elif(msg.arbitration_id ==
0x0CFF06F4):
            t_B2_SOC[i_B2_SOC] = msg.timestamp
            dmesg =
db.decode_message(msg.arbitration_id, msg.data)
            B2_SOC[i_B2_SOC] = dmesg['F4_SOC']

           i_B2_SOC += 1
        elif(msg.arbitration_id == 0x0CFF06F5):

t_B3_SOC[i_B3_SOC] = msg.timestamp
            dmesg = db.decode_message(msg.arbitration_id,
msg.data)
            B3_SOC[i_B3_SOC] = dmesg['F5_SOC']
            i_B3_SOC += 1

elif(msg.arbitration_id == 0x0CFF06F6):
            t_B4_SOC[i_B4_SOC] = msg.timestamp

   dmesg = db.decode_message(msg.arbitration_id, msg.data)
            B4_SOC[i_B4_SOC] =
dmesg['F6_SOC']
            i_B4_SOC += 1
        elif(msg.arbitration_id == 0x0CF090F3):

      t_B1_VC[i_B1_VC] = msg.timestamp
            dmesg =
db.decode_message(msg.arbitration_id, msg.data)
            B1_V[i_B1_VC] =
dmesg['F3_HVESSD1_Voltage']
            B1_C[i_B1_VC] = dmesg['F3_HVESSD1_Current']

i_B1_VC += 1
        elif(msg.arbitration_id == 0x0CF090F4):
            t_B2_VC[i_B2_VC] =
msg.timestamp
            dmesg = db.decode_message(msg.arbitration_id, msg.data)

B2_V[i_B2_VC] = dmesg['F4_HVESSD1_Voltage']
            B2_C[i_B2_VC] =
dmesg['F4_HVESSD1_Current']
            i_B2_VC += 1
        elif(msg.arbitration_id ==
0x0CF090F5):
            t_B3_VC[i_B3_VC] = msg.timestamp
            dmesg =
db.decode_message(msg.arbitration_id, msg.data)
            B3_V[i_B3_VC] =
dmesg['F5_HVESSD1_Voltage']
            B3_C[i_B3_VC] = dmesg['F5_HVESSD1_Current']

i_B3_VC += 1
        elif(msg.arbitration_id == 0x0CF090F6):
            t_B4_VC[i_B4_VC] =
msg.timestamp
            dmesg = db.decode_message(msg.arbitration_id, msg.data)

B4_V[i_B4_VC] = dmesg['F6_HVESSD1_Voltage']
            B4_C[i_B4_VC] =
dmesg['F6_HVESSD1_Current']
            i_B4_VC += 1

# Crear un objeto PdfPages para guardar
los gráficos en archivos PDF independientes
pdf_pages_soc =
PdfPages('/home/alvarocardona/Desktop/TFG/decodificacion/can1/Baterias/grafica_soc.pdf')
pdf_pa
ges_voltage =
PdfPages('/home/alvarocardona/Desktop/TFG/decodificacion/can1/Baterias/grafica_voltaje.pdf')
pd
```

```
f_pages_current =
PdfPages('/home/alvarocardona/Desktop/TFG/decodificacion/can1/Baterias/grafica_corriente.pdf')


# Graficar el Estado de Carga (SOC) de cada batería y guardar en el PDF
correspondiente
plt.figure()
plt.plot(t_B1_SOC[0:i_B1_SOC]-t_B1_SOC[0], B1_SOC[0:i_B1_SOC],
label='Battery 1')
plt.plot(t_B2_SOC[0:i_B2_SOC]-t_B2_SOC[0], B2_SOC[0:i_B2_SOC],
label='Battery 2')
plt.plot(t_B3_SOC[0:i_B3_SOC]-t_B3_SOC[0], B3_SOC[0:i_B3_SOC],
label='Battery 3')
plt.plot(t_B4_SOC[0:i_B4_SOC]-t_B4_SOC[0], B4_SOC[0:i_B4_SOC],
label='Battery 4')
plt.xlabel('Tiempo')
plt.ylabel('Estado de Carga (%)')
plt.title('Estado de
Carga de Baterías')
plt.legend()
pdf_pages_soc.savefig()
plt.close()

# Graficar el Voltaje de
cada batería y guardar en el PDF
correspondiente
plt.figure()
plt.plot(t_B1_VC[0:i_B1_VC]-t_B1_VC[0], B1_V[0:i_B1_VC],
label='Battery 1')
plt.plot(t_B2_VC[0:i_B2_VC]-t_B2_VC[0], B2_V[0:i_B2_VC], label='Battery
2')
plt.plot(t_B3_VC[0:i_B3_VC]-t_B3_VC[0], B3_V[0:i_B3_VC], label='Battery
3')
plt.plot(t_B4_VC[0:i_B4_VC]-t_B4_VC[0], B4_V[0:i_B4_VC], label='Battery
4')
plt.xlabel('Tiempo')
plt.ylabel('Voltaje (V)')
plt.title('Voltaje de
Baterías')
plt.legend()
pdf_pages_voltage.savefig()
plt.close()

# Graficar la Corriente de
cada batería y guardar en el PDF
correspondiente
plt.figure()
plt.plot(t_B1_VC[0:i_B1_VC]-t_B1_VC[0], B1_C[0:i_B1_VC],
label='Battery 1')
plt.plot(t_B2_VC[0:i_B2_VC]-t_B2_VC[0], B2_C[0:i_B2_VC], label='Battery
2')
plt.plot(t_B3_VC[0:i_B3_VC]-t_B3_VC[0], B3_C[0:i_B3_VC], label='Battery
3')
plt.plot(t_B4_VC[0:i_B4_VC]-t_B4_VC[0], B4_C[0:i_B4_VC], label='Battery
4')
plt.xlabel('Tiempo')
plt.ylabel('Corriente (A)')
plt.title('Corriente de
Baterías')
plt.legend()
pdf_pages_current.savefig()
plt.close()

# Cerrar los objetos
PdfPages
pdf_pages_soc.close()
pdf_pages_voltage.close()
pdf_pages_current.close()
```

B.3. Cargadores

```
import cantools
from pprint import pprint
import can
import numpy as np
import
matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import
PdfPages

plt.rcParams.update({
    "text.usetex": True,
    "font.family":
"sans-serif",
    "font.size": 16
})

exp = 'motor_2'
logfile =
'/home/alvarocardona/Desktop/TFG/decodificacion/can1/' + exp + '.log'

dicfile =
'/home/alvarocardona/Desktop/TFG/decodificacion/can1/Baterias/Universal_BMS_FW_2_2+_5BAT_ext.db
c'
db = cantools.database.load_file(dicfile)

N = 6000

t_C1 = np.zeros(N)
i_C1 = 0
C1_RM =
np.zeros(N)
C1_CS = np.zeros(N)

t_C2 = np.zeros(N)
i_C2 = 0
C2_RM = np.zeros(N)
C2_CS =
np.zeros(N)

t_C3 = np.zeros(N)
i_C3 = 0
C3_RM = np.zeros(N)
C3_CS = np.zeros(N)

t_C4 =
np.zeros(N)
i_C4 = 0
C4_RM = np.zeros(N)
C4_CS = np.zeros(N)

with
can.CanutilsLogReader(logfile) as can_log:
    for msg in can_log:

if(msg.arbitration_id == 0x18FF08F3):
            t_C1[i_C1] = msg.timestamp
            dmesg
= db.decode_message(msg.arbitration_id, msg.data)
            C1_RM[i_C1] =
dmesg['F3_STATUS_RunMode'].value
            C1_CS[i_C1] = dmesg['F3_STATUS_ChargerStatus']

        i_C1 += 1
        elif(msg.arbitration_id == 0x18FF08F4):
            t_C2[i_C2] =
msg.timestamp
            dmesg = db.decode_message(msg.arbitration_id, msg.data)

C2_RM[i_C2] = dmesg['F4_STATUS_RunMode'].value
            C2_CS[i_C1] =
dmesg['F4_STATUS_ChargerStatus']
            i_C2 += 1
        elif(msg.arbitration_id ==
0x18FF08F5):
```

```
            t_C3[i_C3] = msg.timestamp
            dmesg =
db.decode_message(msg.arbitration_id, msg.data)
            C3_RM[i_C3] =
dmesg['F5_STATUS_RunMode'].value
            C3_CS[i_C1] = dmesg['F5_STATUS_ChargerStatus']

        i_C3 += 1
        elif(msg.arbitration_id == 0x18FF08F6):
            t_C4[i_C4] =
msg.timestamp
            dmesg = db.decode_message(msg.arbitration_id, msg.data)

C4_RM[i_C4] = dmesg['F6_STATUS_RunMode'].value
            C4_CS[i_C1] =
dmesg['F6_STATUS_ChargerStatus']
            i_C4 += 1

# Continuación del código
anterior

output_pdf_path_runmode =
'/home/alvarocardona/Desktop/TFG/decodificacion/can1/Cargadores/Modo_de_los_cargadores.pdf'
out
put_pdf_path_chargerstatus =
'/home/alvarocardona/Desktop/TFG/decodificacion/can1/Cargadores/Estado_de_los_cargadores.pdf'


# Crear un objeto PdfPages para guardar las gráficas en el archivo PDF de RunMode
with
PdfPages(output_pdf_path_runmode) as pdf_runmode:

    # Crear la gráfica para RunMode

plt.figure(figsize=(12, 8))
    plt.plot(t_C1[0:i_C1]-t_C1[0], C1_RM[0:i_C1], label='Cargador
1')
    plt.plot(t_C2[0:i_C2]-t_C2[0], C2_RM[0:i_C2], label='Cargador 2')

plt.plot(t_C3[0:i_C3]-t_C3[0], C3_RM[0:i_C3], label='Cargador 3')

plt.plot(t_C4[0:i_C4]-t_C4[0], C4_RM[0:i_C4], label='Cargador 4')

    # Ajustar los ticks del
eje y
    plt.yticks([0, 1, 2, 4, 6, 8, 16])

    # Añadir etiquetas y título

plt.xlabel('Tiempo (s)')
    plt.ylabel('Modo de Ejecución')
    plt.title('Modo de ejecución
de los cargadores')
    plt.legend()

    # Guardar la gráfica en el archivo PDF de RunMode

 pdf_runmode.savefig()
    plt.close()

# Crear un objeto PdfPages para guardar las gráficas
en el archivo PDF de ChargerStatus
with PdfPages(output_pdf_path_chargerstatus) as
pdf_chargerstatus:

    # Crear la gráfica para ChargerStatus
    plt.figure(figsize=(12, 8))

   plt.plot(t_C1[0:i_C1]-t_C1[0], C1_CS[0:i_C1], label='Cargador 1')

plt.plot(t_C2[0:i_C2]-t_C2[0], C2_CS[0:i_C2], label='Cargador 2')

plt.plot(t_C3[0:i_C3]-t_C3[0], C3_CS[0:i_C3], label='Cargador 3')

plt.plot(t_C4[0:i_C4]-t_C4[0], C4_CS[0:i_C4], label='Cargador 4')
```

```
    # Añadir etiquetas y
título
    plt.xlabel('Tiempo (s)')
    plt.ylabel('Estado')
    plt.title('Estado de los
Cargadores')
    plt.legend()

    # Guardar la gráfica en el archivo PDF de ChargerStatus

pdf_chargerstatus.savefig()
    plt.close()

# Fin del código
```

B.4. Receptor de radio

```
import cantools
from pprint import pprint
import can
import numpy as np
import
matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import
PdfPages

plt.rcParams.update({
    "text.usetex": True,
    "font.family":
"sans-serif",
    "font.size": 16
})

logfile =
'/home/alvarocardona/Desktop/TFG/decodificacion/Radio_can0/radio_2.log'

dicfile =
'/home/alvarocardona/Desktop/TFG/decodificacion/Radio_can0/Radio_Unsigned.dbc'
db =
cantools.database.load_file(dicfile)

N = 6000

t_ACS = np.zeros(N)
i_ACS = 0
FWD_REV =
np.zeros(N)
LEFT_RIGHT = np.zeros(N)

t_DCS = np.zeros(N)
i_DCS = 0
DK1 = np.zeros(N)
DK11 =
np.zeros(N)
DK12 = np.zeros(N)
DK13 = np.zeros(N)
DK14 = np.zeros(N)
DK27 = np.zeros(N)
DK31 =
np.zeros(N)
DK32 = np.zeros(N)

with can.CanutilsLogReader(logfile) as can_log:
    for msg in
can_log:
        if(msg.arbitration_id == 0x2e4):
            t_DCS[i_DCS] = msg.timestamp

       dmesg = db.decode_message(msg.arbitration_id, msg.data)
            DK1[i_DCS] =
dmesg['DK1_Start']
            DK11[i_DCS] = dmesg['DK11_Option']
            DK12[i_DCS] =
dmesg['DK12_Option']
            DK13[i_DCS] = dmesg['DK13']
            DK14[i_DCS] =
dmesg['DK14']
            DK27[i_DCS] = dmesg['DK27']
            DK31[i_DCS] =
dmesg['DK31_Estop_NO_ContactBit']
            DK32[i_DCS] = dmesg['DK32_Estop_NC_ContactBit']

          i_DCS += 1
        elif(msg.arbitration_id == 0x1e4):
            t_ACS[i_ACS] =
msg.timestamp
            dmesg = db.decode_message(msg.arbitration_id, msg.data)

FWD_REV[i_ACS] = dmesg['FWD_REV']
            LEFT_RIGHT[i_ACS] = dmesg['LEFT_RIGHT']
```

```
  i_ACS += 1

# Directorio para guardar los archivos PDF
output_directory =
'/home/alvarocardona/Desktop/TFG/decodificacion/Radio_can0/Mensajes_decodificados'

# Crear
PDFs en el directorio especificado
with PdfPages(f'{output_directory}/Encendido_Vehiculo.pdf')
as pdf:
    plt.figure()
    plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK1[0:i_DCS], label='DK1')

plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK27[0:i_DCS], label='DK27')
    plt.title('Encendido del
vehículo')
    plt.xlabel('Time')
    plt.ylabel('Value')
    plt.legend()
    pdf.savefig()

  plt.close()

with PdfPages(f'{output_directory}/Activacion_contactores.pdf') as pdf:

plt.figure()
    plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK11[0:i_DCS], label='DK11')

plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK12[0:i_DCS], label='DK12')
    plt.title('Activación
contactores')
    plt.xlabel('Time')
    plt.ylabel('Value')
    plt.legend()

pdf.savefig()
    plt.close()

with PdfPages(f'{output_directory}/Cambio_de_marcha.pdf') as
pdf:
    plt.figure()
    plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK13[0:i_DCS], label='DK13')

plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK14[0:i_DCS], label='DK14')
    plt.title('Cambio de
marcha')
    plt.xlabel('Time')
    plt.ylabel('Value')
    plt.legend()
    pdf.savefig()

plt.close()

with PdfPages(f'{output_directory}/EStop.pdf') as pdf:
    plt.figure()

plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK31[0:i_DCS], label='DK31')

plt.plot(t_DCS[0:i_DCS]-t_DCS[0], DK32[0:i_DCS], label='DK32')
    plt.title('E-Stop')

plt.xlabel('Time')
    plt.ylabel('Value')
    plt.legend()
    pdf.savefig()
    plt.close()


with PdfPages(f'{output_directory}/Analog_control_signal.pdf') as pdf:
    plt.figure()

plt.plot(t_ACS[0:i_ACS]-t_ACS[0], FWD_REV[0:i_ACS], label='FWD_REV')

plt.plot(t_ACS[0:i_ACS]-t_ACS[0], LEFT_RIGHT[0:i_ACS], label='LEFT_RIGHT')
```

```
plt.title('Analog control signal')
    plt.xlabel('Time')
    plt.ylabel('Value')

plt.legend()
    pdf.savefig()
    plt.close()
```

Bibliografía

1. ARGO (2007). Atlas J8 Operator's Manual.
2. CAN bus. (2023). En Wikipedia. https://en.wikipedia.org/wiki/CAN_bus
3. CANopen. (2023). En Wikipedia. https://en.wikipedia.org/wiki/CANopen
4. Martin Falch. (2022). CAN DBC File Explained - A Simple Intro. https://www.csselectronics.com/pages/can-dbc-file-database-intro
5. Rameez Khan, Fahad Mumtaz Malik, Abid Raza and Naveed Mazhar. (2020). Comprehensive study of skid-steer wheeled mobile robots: development and challenges. Industrial Robot: the international journal of robotics research and application. DOI 10.1103/IR-04-2020-0082.
6. CSS Electronics. (2023). CAN bus - The Ultimate Guide.
7. Manuel-Alonso Castro Gil, Gabriel Díaz Orueta, Francisco Mur Pérez, Rafael Sebastián Fernández, Elio Sancristóbal Ruiz, Víctor Miguel Sempere Paya, Javier Silvestre Blanes, Josep Maria Fuertes Armengol, Pau Marti Colom, José Gregorio Yepez Castillo, Perfecto Mariño Espiñeira, Miguel Ángel Domínguez Gómez, Ricardo Mayo Bayón. (2007). Comunicaciones Industriales: Sistemas Distribuidos y Aplicaciones. Librería UNED: c/ Bravo Murillo, 38; 28015 Madrid. ISBN 978-84-362-5467-9.
8. Agile Scientific (2017). https://agilescientific.com/
9. 8 devices. USB2CAN converter datasheet (2023). https://www.8devices.com/
10. Building Automation. Communication Systems with EIB/KNX, LON, and Bacnet (2009). H. Merz, T. Hausemann y C. Hübner. Springer. ISBN: 978-3-540-88828-4.