UNIVERSIDAD DE MÁLAGA

ESCUELA DE INGENIERÍAS INDUSTRIALES

ESCUELA DE INGENIERÍAS INDUSTRIALES
GRADO EN INGENIERÍA ELECTRÓNICA, ROBÓTICA Y MECATRÓNICA

MENCIÓN EN SISTEMAS MECATRÓNICOS EN VEHÍCULOS

TRABAJO DE FIN DE GRADO

Comunicación entre controladoras SEVCON y el PC de control principal del vehículo Panter mediante CANopen y ROS 2

Communication between SEVCON controllers and the Panter's main control PC using CANopen and ROS 2

Realizado por:
José Luis Jaramillo Gago

Tutorizado por:
David Padial Allué
Cotutorizado por:
Javier Serón Barba

Departamento:
Ingeniería de Sistemas y Automática

UNIVERSIDAD DE MÁLAGA

Málaga, Junio de 2025

Agradecimientos

En primer lugar, quiero agradecer a mis padres y a mi hermano por su apoyo durante estos últimos cinco años; sin ellos, no estaría hoy aquí. Al resto de mi familia, a mis tíos, primos y abuelos, tanto a los que ya no están y no han podido verme sufrir y alegrarme en este camino, como a mi abuela Reme, que me ha acompañado a lo largo de toda mi vida y me ha visto empezar y terminar cada una de mis etapas como estudiante y como persona. Puede decirse que esta carrera también es, en parte, vuestra.

Quiero acordarme también de mis amigos: los de toda la vida del pueblo, que se mantienen en la distancia; a los que he conocido durante mis años en Sevilla, con quienes no solo he compartido tardes interminables de estudio intentando resolver problemas imposibles, sino también cervezas, fútbol, viajes y muchos momentos que se quedarán para siempre. Muchos de ellos han acabado convirtiéndose en amigos para toda la vida. Y a quienes me han acogido en este último curso en Málaga, haciéndome sentir en casa desde el principio. No me olvido tampoco de los compañeros de piso que he tenido a lo largo de estos años, con quienes he compartido tantas horas de convivencia.

Por último, quiero agradecer a todos mis profesores, que durante mi paso por la universidad me han ayudado a descubrir y aprender una gran cantidad de cosas y me han confirmado que esta rama de la ingeniería era, sin duda, la elección adecuada. También agradezco especialmente a mis tutores del Trabajo Fin de Grado, por ofrecerme un proyecto tan completo, con tantos recursos a mi alcance, y que he disfrutado de principio a fin.

José Luis Jaramillo Gago
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Málaga, junio de 2025

Resumen:

Este Trabajo de Fin de Grado presenta el diseño e implementación de un sistema de control para el vehículo Panter de la Universidad de Málaga. Se parte del PC seleccionado para manejar el sistema en el que se instala el sistema operativo Ubuntu 22.04 y se integra en la red CAN, estableciendo comunicación con las controladoras del vehículo. A continuación, se incluye un sistema de control remoto y, una vez integrado en el bus y configurado, se desarrollan nodos de ROS 2 que permitan conseguir los objetivos propuestos, como el control simple a distancia, el giro sobre el eje vertical del vehículo o la simulación de un diferencial trasero mediante el modelo de Ackermann, además de esto, se incluyen ciertas funcionalidades con la botonera del mando. El proyecto concluye con una serie de pruebas de funcionamiento en el taller, apoyadas con cálculos teóricos, que validan el funcionamiento correcto del sistema y finalmente se proponen algunas mejoras de cara a futuros avances en el vehículo.

Palabras clave: ROS 2, bus CAN, Ackermann, radiocontrol, Ubuntu 22.04, controladoras

---

Abstract:

This Final Degree Project presents the design and implementation of a control system for the Panter vehicle at the University of Málaga. It starts from the selected PC used to manage the system, where the Ubuntu 22.04 operating system is installed and then integrated into the CAN network, establishing communication with the vehicle's controllers. A remote control system is then included, and once it is integrated into the bus and configured, ROS 2 nodes are developed to achieve the proposed objectives, such as basic remote control, rotation about the vehicle's vertical axis, or simulating a rear differential using the Ackermann model. Additionally, certain functionalities using the controller's button panel are included. The project concludes with a series of tests, supported by theoretical calculations, which validate the correct functioning of the system. Finally, some improvements are proposed for future developments on the vehicle.

Keywords: ROS 2, CAN bus, Ackermann, radio control, Ubuntu 22.04, motor controllers

Glosario de siglas y acrónimos

AC Alternating Current – Corriente Alterna

API Application Programming Interface – Interfaz de Programación de Aplicaciones

BLDC Brushless DC – motor de corriente continua sin escobillas

BMS Battery Management System – Sistema de Gestión de Baterías

CAN Controller Area Network – Red de Área del Controlador

CAN_H / CAN_L Señales alta (High) y baja (Low) del bus CAN

CANopen Protocolo de alto nivel sobre CAN

CiA Can in Automation

COB-ID Communication Object Identifier – Identificador de Objetos de Comunicación

CPU Central Processing Unit – Unidad Central de Procesamiento

DB9 Tipo de conector serie de 9 pines

DDS Data Distribution Service – Servicio de Distribución de Datos

DVT Device Verification Tool – Software de Sevcon para las controladoras

ECU Electronic Control Unit – Unidad de Control Electrónica

ESD Electrostatic Discharge – Descarga Electrostática

FOC Field-Oriented Control – Control de Orientación de Campo

GND Ground – Tierra

GPS Global Positioning System – Sistema de Posicionamiento Global

HDMI High-Definition Multimedia Interface – Interfaz Multimedia de Alta Definición

I/O Input / Output – Entrada / Salida

ID Identifier – Identificador

iDOOR Sistema de expansión modular de Advantech

Imagen ISO Archivo donde se almacena una copia exacta de un sistema de archivos

IP40 / IP65 / IP66 Índices de protección contra polvo y líquidos

IPM / SPM Interior/Surface Permanent Magnet – Imanes Superficiales/Internos

J1939 Protocolo CAN estandarizado para ciertos vehículos

kbps Kilobits por segundo

LiFePO4 Litio-fosfato de hierro – tipo de batería

LiDAR Light Detection and Ranging – tecnología de detección mediante láser

Linux Sistema operativo basado en UNIX

Mbps Megabits por segundo

mPCIe mini Peripheral Component Interconnect express – estándar de expansión

mSATA / SATA (mini) Serial Advanced Technology Attachment

NMT Network Management – tipo de mensaje de CANopen

NXP Next eXPerience – Fabricante de semiconductores (SJA-1000)

PC Personal Computer – Ordenador Personal

PDM18 Modelo del motor eléctrico utilizado

PDO Process Data Object – Objeto de Datos de Proceso

PMAC Permanent Magnet AC – Imanes Permanentes Corriente Alterna

PWM Pulse Width Modulation – Modulación por Ancho de Pulsos

RAM Random Access Memory – Memoria de Acceso Aleatorio

ROS 2 Robot Operating System 2

RPDO Receive PDO – Datos que recibe un nodo desde otros

RS232 Estándar de comunicación serie

SDO Service Data Object – Objeto de Datos de Servicio

SJA1000 Controlador CAN de NXP

SOH State Of Health – Estado de Salud (baterías)

SOC State Of Charge – Estado de Carga (baterías)

SPI Serial Peripheral Interface – Interfaz de periféricos serie

SSD Solid State Drive – Unidad de Estado Sólido

TPDO Transmit PDO – Datos enviados por un nodo al resto

TX / RX Transmisión / Recepción

UART Transmisor-Receptor Asíncrono Universal

USB Universal Serial Bus

Índice general

1. Introducción **1**
- 1.1. Antecedentes . . . 1
- 1.2. Objeto . . . 1
- 1.3. Motivación . . . 2
- 1.4. Plan de trabajo . . . 3

2. Fundamentos teóricos **5**
- 2.1. Motores PMAC . . . 5
- 2.2. Baterías . . . 6
- 2.3. Controladoras . . . 7
- 2.4. Unidad de control . . . 8
  - 2.4.1. Arquitectura de comunicación . . . 9
- 2.5. Entorno ROS 2 *Humble* . . . 9
- 2.6. Bus CAN . . . 11
  - 2.6.1. CANopen . . . 13
- 2.7. Comunicación por radio . . . 14

3. Hardware utilizado **17**
- 3.1. Motores PDM18 . . . 17
  - 3.1.1. Encoder . . . 19
- 3.2. Vanguard Lithium Ion Battery . . . 20
- 3.3. Controladora Sevcon Gen4 . . . 23
  - 3.3.1. Contactores . . . 24
- 3.4. PC Advantech UNO-2484G . . . 25
  - 3.4.1. Extensión para manejar CAN . . . 26
  - 3.4.2. Sistema operativo . . . 27
- 3.5. Equipo de radiocontrol Scanreco . . . 28
  - 3.5.1. Mando Scanreco Mini . . . 28
  - 3.5.2. Receptor Scanreco G3B . . . 29

4. Conexionado del hardware **31**
- 4.1. PC de control principal . . . 31
- 4.2. Receptor de radio Scanreco . . . 32
  - 4.2.1. Manguera A (Operación) . . . 32
  - 4.2.2. Manguera B (Configuración) . . . 33
  - 4.2.3. Manguera C (No utilizada) . . . 34

4.3. Controladoras SEVCON 34
4.4. Red de comunicación CAN 36
4.5. Suministro eléctrico 38
4.6. Diagrama final de conexiones 40

5. Ejecución del proyecto **43**
5.1. Inicialización del Advantech UNO-2484G 43
5.2. Programación de dispositivos 45
5.2.1. Programación interna del receptor Scanreco G3B 45
5.2.2. Configuración de las controladoras 46
5.3. Diseño del software 47
5.3.1. Primera versión - Envío y recepción de mensajes 47
5.3.2. Segunda versión - Manejo con joysticks 49
5.3.3. Versión final - Modularización e inclusión de botones 54
5.4. Corrección de errores 69

6. Pruebas y validación del sistema **71**
6.1. Velocidad máxima sin limitador 73
6.2. Velocidad máxima con limitador del 20 % 74
6.3. Marcha hacia adelante con giro Ackermann 75
6.4. Marcha atrás con giro Ackermann 77
6.5. Giro sobre sí mismo (derecha) 78
6.6. Giro sobre sí mismo (izquierda) 80
6.7. Prueba de arranque y parada 81

7. Resultados del trabajo **83**

8. Ideas de trabajo futuro **85**

Bibliografía **87**

ANEXOS **91**

A. Códigos desarrollados **93**
A.1. Códigos básicos para interactuar con CAN 93
receptor_can.c 93
envio_can_BASICO.c 95
envio_can_ACK.c 96
A.2. Códigos finales del proyecto 103
receptor.c 103
ackermann.c 108
emisor_can.c 115

B. Puesta en marcha del ordenador **125**
B.1. Instalación del sistema operativo 125
B.2. Instalación de ROS 2 *Humble* 127
B.3. Configuración de los puertos CAN 128

C. Scripts de inicialización del puerto can0 **129**

D. Diccionario de consignas CAN **133**

E. Documentación Scanreco **137**

F. Documentación Advantech UNO-2484G **147**

G. Documentación módulo PDM-26D2CA **153**

H. Códigos de error del Scanreco G3B **157**

Índice de figuras

1.1. Vehículo Panter 4x4 de Movelco. Fuente: [3] 2
1.2. Argo Rover J8. Fuente: [4] 3

2.1. Diferencia entre rotores. Fuente: [5] 6
2.2. ROS 2 Humble. Fuente: [8] 9
2.3. Modelo de arquitectura en ROS 2. Fuente: Elaboración propia 10
2.4. Diagrama bus CAN. Fuente: [10] 11
2.5. Esquema de transmisión de bits en bus CAN. Fuente: [11] 12
2.6. Dispositivo USB-to-CAN de IXXAT. Fuente: [12] 13
2.7. Control remoto de maquinaria pesada. Fuente: [15] 15

3.1. Motor PDM18. Fuente: [17] 17
3.2. Curvas de Par y Potencia del motor PDM18. Fuente: Elaboración propia. . 19
3.3. Comportamiento del *encoder*. Fuente: [16] 20
3.4. Batería Vanguard 48V 5kWh Lithium Ion Battery Fuente: [19] 21
3.5. Controladora Sevcon Gen4 Size 6. Fuente: [21] 23
3.6. Contactores Albright SW200. Fuente: [22] 24
3.7. Advantech UNO-2484G Fuente: [23] 25
3.8. Módulo de expansión EKBE instalado sobre el UNO-2484G. Fuente: [25] 26
3.9. Módulo CAN PCM-26D2CA-BE con sus dos puertos DB9. Fuente: [25] . 26
3.10. Sistema operativo Ubuntu. Fuente: [27] 27
3.11. Logo de Scanreco. Fuente: [15] 28
3.12. Mando Scanreco Mini. Fuente: Elaboración propia 28
3.13. Batería recargable del mando. Fuente: [15] 29
3.14. Receptor Scanreco G3B. Fuente: [15] 29

4.1. Parte delantera y trasera del PC ensamblado. Fuente: Elaboración propia. 31
4.2. Disco duro de la marca WD Blue utilizado. Fuente: [28] 32
4.3. Conector M12 de 5 pines hembra utilizado para la manguera A. Fuente: [29]. 33
4.4. Conector M12 de 5 pines macho utilizado para la manguera B. Fuente: [29]. 34
4.5. Adaptador USB a puerto serie. Fuente: Elaboración propia. 34
4.6. Diagrama de pines de la controladora. Fuente: [31] 35
4.7. Resistencia de terminación del PCM-26D2CA-BE seleccionada mediante jumper. Fuente: Elaboración propia 37
4.8. Diagrama simple del bus CAN completo. Fuente: Elaboración propia . . . 37
4.9. Esquema objetivo de nodos del bus CAN. Fuente: [33] 38
4.10. Diagrama de conexiones provisional. Fuente: Elaboración propia 40

4.11. Diagrama de conexiones definitivas. Fuente: Elaboración propia 40

5.1. Rango del acelerador configurado. Fuente: Software DVT [36] 47
5.2. Mapeo de los valores del joystick de aceleración. Fuente: Elaboración propia 52
5.3. Distribución de botones en el Scanreco Mini. Fuente: [38] 54
5.4. Tipología de cada botón. Fuente: [38] 55
5.5. Diagrama de flujo del nodo `ackermann.c`. Fuente: Elaboración propia . . 59
5.6. Diagrama generado con `rqt_graph`. Fuente: Elaboración propia 66

6.1. Montaje utilizado en el taller durante las pruebas. Fuente: Elaboración propia 71
6.2. Velocidad máxima sin limitador. Fuente: Elaboración propia 73
6.3. Velocidad máxima con limitador del 20 %. Fuente: Elaboración propia . . 74
6.4. Marcha hacia adelante con giro a la izquierda. Fuente: Elaboración propia 75
6.5. Marcha atrás con giro a la derecha. Fuente: Elaboración propia 77
6.6. Giro sobre sí mismo hacia la derecha. Fuente: Elaboración propia 79
6.7. Giro sobre sí mismo hacia la izquierda. Fuente: Elaboración propia 80
6.8. Mensajes CAN en arranque y parada del sistema. Fuente: Elaboración propia 81

B.1. Software para crear la memoria USB booteable. Fuente [41] 125
B.2. Configuración de Rufus utilizada. Fuente: Elaboración propia 126

Índice de tablas

3.1. Especificaciones técnicas del motor PDM18. Fuente: [16] 18
3.2. Datos de las curvas de Par y Potencia del motor PDM18. Fuente: [18] . . 19
3.3. Especificaciones técnicas del encoder. Fuente: [16] 20
3.4. Especificaciones técnicas de las baterías Vanguard 48V 5kWh. Fuente: [20] 22
3.5. Características principales del UNO-2484G. Fuente: [24] 25
3.6. Características del módulo CAN PCM-26D2CA-BE. Fuente: [26] 27

4.1. Conexiones relevantes de las controladoras. Fuente: [31] 36

5.1. Topics en los que publica `receptor.c`. Fuente: Elaboración propia. . . . 56
5.2. Valores de `vmax` posibles. Fuente: Elaboración propia. 57
5.3. Topics a los que se suscribe y donde publica `ackermann.c`. Fuente: Elaboración propia. 58
5.4. Topics a los que se suscribe `emisor_can.c`. Fuente: Elaboración propia. 62

Índice de códigos

5.1. Ejecución del `talker` . . . 44
5.2. Ejecución del `listener` . . . 44
5.3. Configuración y activación del puerto CAN . . . 45
5.4. Instalación de herramientas de diagnóstico CAN . . . 45
5.5. Construcción de mensajes en `envio_can_BASIC0.c` . . . 48
5.6. Filtrado de mensajes en `receptor_can.c` . . . 49
5.7. Inicialización del socket CAN . . . 51
5.8. Gestión de mensajes entrantes por ID . . . 51
5.9. Mapeado de aceleración del joystick . . . 52
5.10. Cálculo del ángulo y radio de giro . . . 53
5.11. Ajuste de velocidades para el giro . . . 53
5.12. Envío de consignas de velocidad por CAN . . . 53
5.13. Creación del socket CAN `can0` . . . 56
5.14. Procesamiento de mensajes del joystick (ID `0x186`) . . . 56
5.15. Extracción de valores del byte de control . . . 57
5.16. Publicación del valor `vmax` . . . 57
5.17. Manejador de señales y liberación de recursos . . . 58
5.18. Timer del nodo . . . 60
5.19. Definición de parámetros físicos del vehículo . . . 60
5.20. Mapeo del valor del joystick a velocidad . . . 61
5.21. Escalado de la velocidad máxima . . . 61
5.22. Función auxiliar `send_can_frame()` . . . 63
5.23. Giro sobre eje en `process_can()` . . . 63
5.24. Ackermann en `process_can()` . . . 64
5.25. Velocidades de motores en `process_can()` . . . 64
5.26. Arranque/parada en `process_can()` . . . 64
5.27. Manejador de señal y liberación de recursos . . . 65
5.28. Archivo de lanzamiento `sistema.launch.py` . . . 68
A.1. Nodo en C que lee del bus CAN y muestra en el terminal, permitiendo filtrado . . . 93
A.2. Nodo en C que permite publicar en el bus . . . 95
A.3. Nodo en C que lee del bus CAN y calcula y envia velocidades segun joysticks 96
A.4. Nodo en C que lee del bus CAN y publica en distintos topics . . . 103
A.5. Nodo en C que lee de distintos topics y publica velocidades calculadas mediante Ackermann en otros . . . 108
A.6. Nodo en C que lee de distintos topics y publica en bus CAN . . . 115

B.1. Actualización del sistema tras la instalación 126
B.2. Compilación e instalación de los drivers 128
B.3. Carga de módulos 128
B.4. Carga de módulos SocketCAN 128
C.1. Script de configuración de la interfaz CAN 129
C.2. Dar permisos de ejecucion al script 130
C.3. Contenido del archivo de servicio 130
C.4. Habilitar el servicio systemd 131
C.5. Iniciar el servicio manualmente 131
C.6. Verificar el estado del servicio 131

CAPÍTULO 1

Introducción

En este capítulo se expone el marco general del Trabajo de Fin de Grado, comenzando por el objetivo que se pretende alcanzar y siguiendo con los antecedentes que han servido de base para el desarrollo. Se hace una mención a la motivación que ha impulsado su elaboración y, finalmente, se presenta el plan de trabajo seguido.

1.1. Antecedentes

Para este proyecto se tomarán como antecedentes otros trabajos que también han sido tutorizados por el departamento de Ingeniería de Sistemas y Automática, que serán:

- **Configuración e implantación de controladoras en un vehículo eléctrico** [1], donde se detalla el procedimiento seguido para la implementación de las controladoras Sevcon Gen4 Size6 (serán presentadas posteriormente) con los motores, usando como comunicación entre ellas bus CAN y en el que se define el formato de las tramas.
- **Diseño del cableado de un vehículo de rescate** [2], donde se desarrolla y describe toda la instalación eléctrica del vehículo que será de utilidad más adelante, la parte de control más que la de potencia aunque ambas serán necesarias.

1.2. Objeto

Este Trabajo de Fin de Grado forma parte del proyecto Panter de la Universidad de Málaga, cuyo objetivo es crear un vehículo de rescate basado en el Panter 4x4 (Figura 1.1), del que se utilizará únicamente el chasis, rediseñando el resto para optimizar su rendimiento y que pueda ser empleado en actividades de este tipo.

Figura 1.1: Vehículo Panter 4x4 de Movelco. Fuente: [3]

El propósito de este proyecto es la instalación en un vehículo eléctrico real, como es el Panter, de un ordenador con el sistema operativo Linux Ubuntu 22.04 y el entorno de programación ROS 2, con el fin de permitir la comunicación con dos controladoras a través de bus CAN, un protocolo de comunicación muy empleado en la industria automotriz por su gran resistencia y excelente rechazo al ruido electromagnético.

Una vez implementada la comunicación con el ordenador, que será el UNO-2484G, la adición de nuevos dispositivos será sencilla, puesto que solo habrá que ampliar la instalación física de los dos hilos que portan los mensajes CAN y gestionar los nuevos mensajes en el PC. Por ello, una vez instalado el ordenador y cuando se logre la comunicación con las controladoras, se ampliará el bus añadiendo un dispositivo de control remoto y se adaptará para que tenga una funcionalidad completa con ROS 2.

1.3. Motivación

La principal motivación de este trabajo es el hecho de participar en el desarrollo y mejora de un vehículo eléctrico real como es el Panter, vehículo de rescate que está desarrollando la Universidad de Málaga, y que sirve como broche a la mención de Sistemas Mecatrónicos en Vehículos. Este proyecto trata de complementar las funcionalidades del Argo Rover J8 [4], vehículo militar de rescate adquirido por la Universidad de Málaga, mediante la creación de otro vehículo todoterreno que pueda realizar misiones de rescate junto a él.

Se plantea como un reto porque, aunque una parte de los conocimientos ha sido adquirida a lo largo de los años de estudio, como el funcionamiento de los motores de corriente alterna e imanes permanentes (PMAC) o los procedimientos para las instalaciones eléctricas y fundamentos de control entre otras cosas, otros, como la comunicación mediante bus CAN o la programación en ROS 2, han sido investigados y estudiados sin docencia.

Figura 1.2: Argo Rover J8. Fuente: [4]

Aunque aprender algo nuevo apartado de las clases se antoje difícil, el hecho de que sean dos herramientas que se utilizan con mucha frecuencia en entornos de trabajo a los que está destinado un titulado de este grado, hace que el trabajo sea aún más interesante, útil y satisfactorio.

1.4. Plan de trabajo

El plan de trabajo puede dividirse en distintas fases, aunque varias de ellas estarán presentes de forma simultánea:

- **Fase 1: Investigación**: en esta fase se adquirirán los conocimientos necesarios para trabajar con CANopen y ROS 2 Humble mediante libros, documentación oficial o trabajos anteriores.
- **Fase 2: Adecuación y diseño del sistema**: se pondrá en funcionamiento el Advantech UNO-2484G, se le instalará el sistema operativo elegido (Linux Ubuntu 22.04) y se armará el entorno de trabajo de ROS 2 Humble.
- **Fase 3: Desarrollo del proyecto**: en esta fase se empezará a programar para que el ordenador sea un nodo más del sistema de controladoras, haciendo que sea capaz de mandar y recibir mensajes a los motores que, en principio, serán consignas de velocidad deseada, además, se incluirá en el bus el receptor de radio.

- **Fase 4: Pruebas**: ensayo en el laboratorio con los equipos reales y perfeccionamiento del código. Tras llegar a un estado que se considere óptimo de funcionamiento, se dará por válido.
- **Fase 5: Documentación**: aunque estará presente durante todo el desarrollo del proyecto, en esta fase se detallará en una memoria todo el marco teórico, los pasos seguidos, los problemas encontrados y el resultado final junto a futuras posibles mejoras.

CAPÍTULO 2

Fundamentos teóricos

En este capítulo se comenta el estado actual y los usos de las diferentes tecnologías utilizadas en este proyecto, con el fin de ofrecer una idea general del entorno con el que se va a trabajar. Se presentan las características técnicas y algunos ejemplos prácticos que ayuden a entender cómo funcionan y por qué se han elegido; de esta forma, se sientan las bases para continuar con el resto del trabajo.

2.1. Motores PMAC

Un PMAC (**P**ermanent **M**agnet **A**lternating **C**urrent) es un tipo de motor eléctrico que funciona a partir del efecto del campo magnético producido en el estátor por la corriente alterna trifásica, que se le proporciona desde la fuente de alimentación, sobre los imanes permanentes que se encuentran solidarios al rotor. Esta interacción permite la transformación de energía eléctrica en energía mecánica.

Los motores PMAC pueden pertenecer a dos grandes grupos según la fabricación de su rotor:

- De imanes permanentes superficiales (SPM): al tener los imanes situados en la superficie del rotor se consigue un diseño simple, el control de este tipo de motores es muy sencillo teniendo a cambio un rango de velocidades bastante limitado.
- De imanes permanentes internos (IPM): en este caso los imanes están incrustados en el rotor, lo que mejora la eficiencia y permite aprovechar el par de reluctancia a cambio de un mayor coste de mantenimiento y fabricación. Estos motores resultan bastante útiles en aplicaciones que requieren un rango grande de velocidades, aunque su control es más complejo.

La ventaja de los motores de imanes permanentes sobre los de inducción radica principalmente en la eficiencia, puesto que no existen pérdidas en el rotor y tienen una mayor densidad de potencia, lo que permite diseños compactos. Además, se puede optimizar su

rendimiento para cada tarea específica mediante, por ejemplo, variadores de frecuencia o controladores específicos que permiten un control muy preciso.

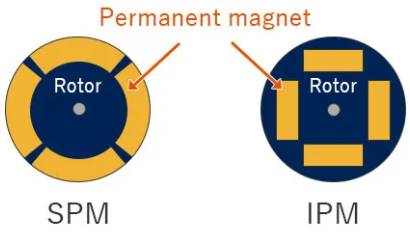


Figura 2.1: Diferencia entre rotores. Fuente: [5]

Estos motores se utilizan en una amplia variedad de aplicaciones industriales y de movilidad eléctrica (como es este caso). Como se mencionó antes, se pueden integrar en sistemas de control avanzado como los utilizados en el sector de la automoción y que permiten mejorar la eficiencia energética y la dinámica de respuesta en sistemas de alto rendimiento.

2.2. Baterías

La batería es uno de los componentes más importantes en un vehículo eléctrico, ya que determina factores como la autonomía, el rendimiento, el peso o los requisitos de carga. Su correcta selección y dimensionamiento son clave para garantizar un funcionamiento eficiente del vehículo.

Actualmente, los principales tipos de baterías empleados en vehículos eléctricos son:

- Plomo-ácido; aunque económicas, son pesadas y tienen baja densidad energética, por lo que su uso se limita a aplicaciones auxiliares como la de arranque en un vehículo de combustión.
- Níquel-metal hidruro (NiMH); ofrecen mejores características, pero presentan problemas de autodescarga y efecto memoria (disminución de la capacidad por descargas parciales).
- Ion-litio (Li-ion), destacan por su alta densidad energética, bajo peso, buen rendimiento térmico y elevada eficiencia, lo que las ha convierte en la opción más utilizada en la actualidad.
- Litio-fosfato de hierro ($LiFePO_4$), un tipo dentro de las de ion-litio que ofrece mayor seguridad y durabilidad sacrificando ligeramente densidad energética.

Al elegir una batería para un vehículo eléctrico, deben considerarse factores como la tensión nominal (que debe ser compatible con motores y controladoras), la capacidad energética (kWh) que representa la cantidad de energía que puede almacenar, la capacidad de entrega de corriente en carga máxima, el número de ciclos de vida útil, el tiempo de recarga, la eficiencia energética, la presencia de un sistema de gestión de batería (BMS), la compatibilidad con protocolos de comunicación como CANbus o J1939 para gestionar el mismo o acceder a datos de salud, la seguridad frente a sobretemperatura o cortocircuitos, y certificaciones, como la IP66, que garanticen su resistencia a polvo y agua.

2.3. Controladoras

La controladora de un vehículo eléctrico es el dispositivo intermedio entre las baterías y el motor. Su función principal consiste en regular la energía, proveniente de las baterías, que se tiene que suministrar al motor, ajustándose a parámetros como la velocidad, el par y la dirección de rotación. Además se encarga de gestionar el frenado regenerativo, permitiendo recuperar parte de la energía cinética del vehículo que se devuelve a las baterías.

Las controladoras trabajan convirtiendo la corriente continua (DC) que obtienen de las baterías en corriente alterna (AC) mediante un inversor, este proceso es necesario para alimentar motores de corriente alterna como los PMAC (Sección 2.1), utilizados en este proyecto. El inversor puede utilizar distintos métodos para modular la potencia suministrada al motor, por ejemplo:

- **Modulación por ancho de pulso (PWM)**: Técnica en la que se varía el ciclo de trabajo de la señal de conmutación para controlar la tensión media aplicada al motor.
- **Control de orientación de campo (FOC)**: Algoritmo más avanzado que permite un control preciso del flujo magnético en el motor, optimizando la eficiencia y la respuesta del dispositivo conectado, en este caso el motor PMAC.
- **Ajuste de frecuencia**: Método utilizado en motores de inducción de corriente alterna, en el que se modifica la frecuencia de la alimentación para regular la velocidad del motor.

Las controladoras también incluyen conexiones para sensores externos que, además de contribuir a la seguridad, proporcionan información importante para el funcionamiento óptimo del sistema. Entre ellos destacan:

- **Sensores de corriente**: Permiten medir el consumo del motor y optimizar la entrega de potencia.

- **Sensores de temperatura**: Protegen tanto a la controladora como al motor contra posibles sobrecalentamientos.
- **Encoders o sensores de efecto Hall**: Proveen información sobre la posición y velocidad del rotor, esencial para la implementación del control.

En el ámbito del control de motores, es importante saber que existen dos estrategias básicas según la consigna que se quiera enviar: el control de par y el control de velocidad.

El **control de par** se basa en la regulación directa del par generado por el motor, lo que se traduce en un ajuste bastante preciso de la corriente que circula a través de las fases del estátor. Esta técnica resulta especialmente útil en aplicaciones en las que se trabaja con cargas variables, como en maquinaria industrial, brazos robóticos o sistemas que están expuestos a perturbaciones externas.

En cuanto al **control de velocidad**, se centra en mantener estable la velocidad angular del eje del motor. Para conseguirlo, se utiliza un bucle de control que compara la velocidad medida con la consigna y actúa para mantenerse cerca de la misma. Esta estrategia es habitual en aplicaciones donde el seguimiento de trayectorias es importante, como en vehículos autónomos.

2.4. Unidad de control

La unidad de control es el componente encargado de coordinar el funcionamiento de los distintos componentes del vehículo, procesa la información proveniente de sensores y otros elementos del sistema para generar señales dirigidas a actuadores como pueden ser las controladoras de los motores.

En vehículos eléctricos, la unidad de control se encarga de la gestión de la velocidad, la monitorización del estado de carga de la batería o la supervisión de condiciones de seguridad (temperatura, tensión, etc.), así como de la coordinación de los diferentes sistemas que tienen que interactuar entre sí, entre otras cosas.

Además, también puede encargarse del registro de eventos, la comunicación con el exterior y de la aplicación de protecciones para velar por el buen estado del sistema ante posibles fallos.

Existen diferentes tipos de unidades de control empleadas en vehículos eléctricos dependiendo de la aplicación, el nivel de complejidad del sistema, la arquitectura del vehículo y los requisitos de rendimiento:

- **Unidades de control electrónicas (ECUs):** diseñadas para controlar tareas concretas, como el sistema de tracción, la batería o los frenos.

- **Microcontroladores:** ejecutan un firmware específico.
- **Computadoras:** Unidades potentes que pueden ejecutar sistemas operativos (como Linux) y permiten la integración de software como ROS 2.

2.4.1. Arquitectura de comunicación

La unidad de control no trabaja de forma aislada, sino que se integra dentro de una red de dispositivos con los que necesita intercambiar información, para ello la arquitectura de comunicación más empleada es la red CAN (Controller Area Network), que permite a las diferentes ECUs intercambiar información en tiempo real.

Además del bus CAN, pueden emplearse otras redes como LIN o Ethernet dependiendo de diversos factores como la necesidad de rechazo a posibles ruidos en la señal. La unidad de control central suele encargarse de recopilar datos, tomar decisiones y enviar comandos al resto de componentes electrónicos mediante estos buses.

2.5. Entorno ROS 2 *Humble*

ROS 2 (**R**obot **O**perating **S**ystem **2** [6] [7]) es un sistema operativo de código abierto diseñado para el desarrollo de software en robótica, compuesto por un conjunto de bibliotecas, herramientas y convenciones. Surge como la evolución de ROS 1 para superar límites de seguridad, escalabilidad y solucionar la comunicación en tiempo real.

Figura 2.2: ROS 2 Humble. Fuente: [8]

ROS 2 está construido sobre el *middleware* DDS (**D**ata **D**istribution **S**ervice). Un *middleware* es un software que permite la comunicación entre aplicaciones y sistemas operativos, lo que asegura una comunicación robusta, distribuida y configurable entre nodos.

Entre las principales características, cabe mencionar las siguientes:

- **Sistema distribuido**: cada nodo puede ser ejecutado en un mismo dispositivo o en múltiples máquinas conectadas entre sí.
- **Comunicaciones robustas** mediante DDS.
- **Soporte para tiempo real** imprescindible para aplicaciones críticas y de control.

- **Modularidad**: facilita el desarrollo de sistemas más complejos usando paquetes, nodos y librerías ya existentes.
- **Lenguajes compatibles**: destacan *C*, *C++* y *Python*

ROS 2 utiliza un sistema de comunicación basado en un modelo *publisher - subscriber* (publicador - suscriptor) mediante el uso de *topics* (temas). Un *topic* es un canal de comunicación identificado por un nombre único, a través del cual los nodos pueden intercambiar mensajes. Cada nodo puede actuar como *publisher* (enviando datos a un *topic*), como *subscriber* (recibiendo datos desde un *topic*) o como ambos a la vez. Un mismo nodo puede publicar en múltiples *topics* y suscribirse simultáneamente a otros, lo que permite una arquitectura desacoplada. Esto facilita que los distintos módulos del sistema se comuniquen sin necesidad de conocerse entre sí, lo que mejora la escalabilidad y reutilización del código, además de facilitar su mantenimiento y edición.

En la Figura 2.3 se muestra un ejemplo de esta arquitectura: varios nodos actúan como publicadores enviando información a diversos *topics*, los cuales a su vez son consumidos por otros nodos suscriptores. Es habitual que un mismo *topic* reciba publicaciones de más de un nodo o que su contenido sea consumido por múltiples nodos suscriptores, según las necesidades del sistema.

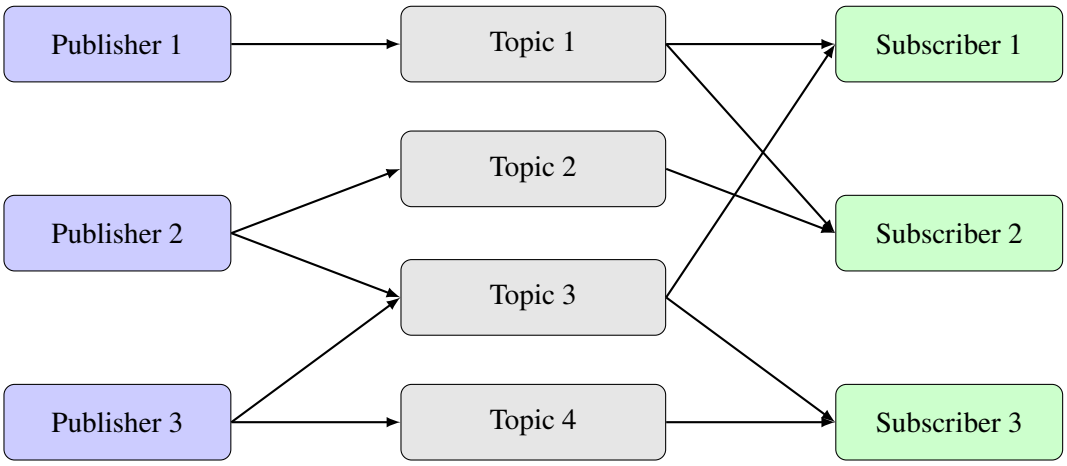


Figura 2.3: Modelo de arquitectura en ROS 2. Fuente: Elaboración propia

Concretamente, la distribución de ROS 2 elegida es *Humble*, la octava versión estable del sistema operativo lanzada en mayo de 2022 y con soporte a largo plazo (al menos hasta mayo de 2027), lo que la convierte en una opción adecuada para proyectos industriales.

Está basada en Ubuntu 22.04 que, como se explicará en la Sección 3.4, es el sistema operativo que se ha seleccionado para el PC del vehículo. Además, posee soporte completo en los lenguajes mencionados anteriormente (*C*, *C++* y *Python*) y tiene una amplia compatibilidad con bibliotecas de ROS 1 que han sido migradas a ROS 2 *Humble*.

Otra ventaja de esta versión es la gran cantidad de documentación disponible, que resulta de ayuda para el desarrollo de programas y la resolución de posibles problemas.

En conclusión, se ha elegido la versión *Humble* por ser una versión altamente recomendada para proyectos que requieren estabilidad a largo plazo, soporte ante posibles fallos y acceso a funcionalidades modernas de ROS 2.

2.6. Bus CAN

El bus **CAN** (**C**ontroller **A**rea **N**etwork) [9] es un protocolo de comunicación creado por *Bosch GmbH* a inicios de 1980 ante la necesidad de disminuir la gran cantidad de cableado presente en automóviles y reducir así el coste y la complejidad de los sistemas. Desde entonces, su aplicación se ha llevado a una gran cantidad de campos muy distintos de la industria automotriz, abarcando desde la robótica y la automatización industrial hasta todo tipo de entornos donde se necesita una comunicación eficiente, robusta y de alta fiabilidad entre varios dispositivos.

CAN posibilita la conexión de numerosos nodos dentro de una red, sin necesidad de un controlador central y permitiendo establecer múltiples *masters* en lugar de restringir la conexión a un único *master* y varios *slaves*, es decir, es un protocolo con capacidad multimaestro. Cada uno de los nodos tiene la capacidad de enviar y recibir mensajes, es esta característica la que lo convierte en una solución ideal para sistemas donde el intercambio continuo de datos entre módulos es la base del funcionamiento general.

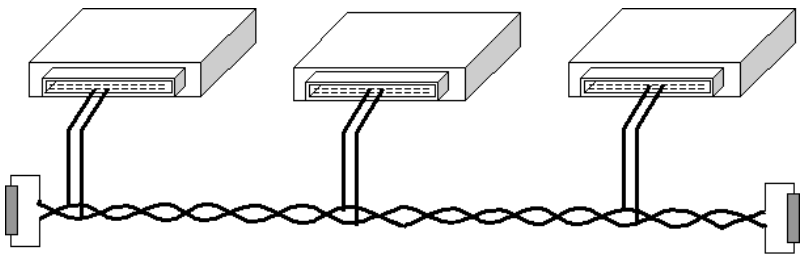


Figura 2.4: Diagrama bus CAN. Fuente: [10]

Una de las principales características de CAN es su enfoque en los mensajes en lugar de en direcciones físicas. Cada mensaje transmitido por el bus contiene un identificador exclusivo que señala su prioridad y tipo, lo que permite que cualquier nodo de la red determine, según este identificador, si se debe procesar o si, en cambio, no es útil y está destinado a otro dispositivo (nodo). Esto lleva a una manera eficiente y directa de comunicación centrada en el contenido, resultando especialmente útil cuando varios dispositivos deben responder al mismo tipo de información, puesto que en lugar de reenviar

el mismo mensaje con diferentes direcciones, se envía uno único con un identificador que captan los dispositivos que lo necesiten.

Además, el protocolo implementa un mecanismo de arbitraje que se encarga de que cuando varios nodos intentan transmitir al mismo tiempo, el mensaje con mayor prioridad (el que tiene el ID más bajo a no ser que se indique lo contrario) se transmita sin colisiones y sin pérdida de información.

La robustez del bus CAN no solo se debe a su mecanismo de arbitraje, sino también a su forma de transmisión diferencial de cada uno de los bits del mensaje. El sistema de comunicación CAN está basado en una topología en bus con dos líneas de comunicación diferenciadas, denominadas CAN High (CAN_H) y CAN Low (CAN_L). Estas líneas son implementadas físicamente mediante un par de hilos trenzados (y normalmente apantallados), lo que también proporciona una alta inmunidad al ruido electromagnético. Esta forma de transmisión hace que el bus CAN mantenga una alta fiabilidad incluso en entornos desfavorables.

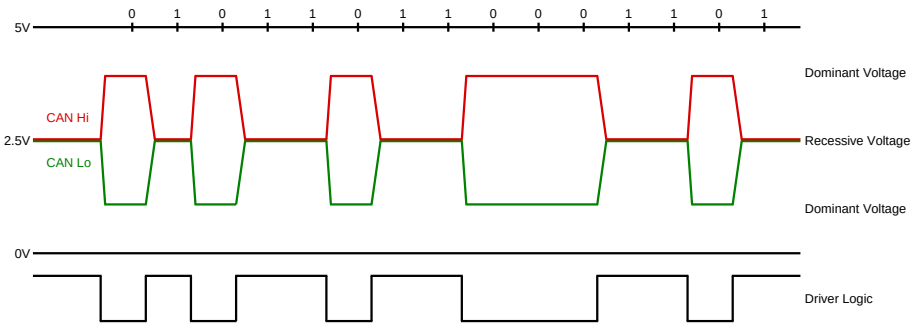


Figura 2.5: Esquema de transmisión de bits en bus CAN. Fuente: [11]

Esta robustez eléctrica y la efectividad de la transmisión diferencial son atributos importantes en aplicaciones como la de este proyecto. El vehículo que se está desarrollando requiere un sistema que permita manejar datos en tiempo real y también que sea confiable en condiciones adversas. En este caso, el uso del bus CAN se justifica por su capacidad para mantener una comunicación sin interferencias entre el PC, controladoras y otros componentes del sistema.

Desde el punto de vista del diseño del sistema, uno de los beneficios de la utilización de CAN es que en lugar de conectar cada sensor o actuador directamente a la unidad central mediante cables, cada dispositivo se conecta al bus común y éste a uno de los puertos del PC. Esto reduce muy significativamente el número total de cables, simplifica el mantenimiento y permite una mayor modularidad del sistema. La escalabilidad del bus facilita la integración progresiva de nuevos módulos sin alterar la instalación existente; únicamente es necesario tener en cuenta que habrá que instalar en los extremos resistencias

de 120 Ω para que el bus tenga sus terminaciones definidas y funcione correctamente.

Aunque existen dos tipos distintos de bus CAN, el de alta velocidad (*high-speed*) es el indicado para la comunicación con motores, frenos y sistemas que requieran seguridad; en cambio, el de baja velocidad (*low-speed*) servirá para el accionamiento de otros sistemas, como la iluminación. Más adelante se definirá la velocidad que se usará, teniendo en cuenta que el estándar CAN 2.0 permite velocidades de hasta 1 Mbps, más que suficiente para la mayoría de aplicaciones en vehículos.

La elección del bus CAN en el presente proyecto no se determina únicamente por su amplia adopción industrial, sino también por su idoneidad técnica frente a los requisitos del sistema. El vehículo semiautónomo que se desarrolla en este trabajo requiere un sistema de comunicaciones que soporte interacciones constantes entre la unidad de control, las controladoras, los sensores y los dispositivos de entrada como el joystick (Sección 2.7). Estas interacciones deben producirse de forma fiable, con baja latencia y sin comprometer la integridad de los datos. El bus CAN satisface estas condiciones de manera eficiente, al tiempo que permite futuras ampliaciones del sistema.

Por otro lado, el empleo del protocolo CAN facilita también la supervisión y diagnóstico del sistema en tiempo real. Durante el desarrollo del trabajo, se utilizarán tanto herramientas como CANanalyser (junto al USB-to-CAN de IXXAT) como el propio puerto CAN del PC Advantech que permitirán visualizar los mensajes que circulan por el bus, detectar errores y realizar pruebas funcionales sin necesidad de intervenir directamente sobre el hardware. Esta capacidad de depuración y análisis es especialmente valiosa durante las fases de desarrollo y validación del sistema, y contribuye a reducir los tiempos de ensayo y error en el diseño experimental.

Figura 2.6: Dispositivo USB-to-CAN de IXXAT. Fuente: [12]

2.6.1. CANopen

CANopen [13] es un protocolo basado en el bus CAN, diseñado para facilitar la comunicación entre dispositivos dentro de redes distribuidas, fue desarrollado por la organización

CiA (*CAN in Automation* [14]) y es muy utilizado en aplicaciones de automatización, robótica o vehículos (es el que utilizan las controladoras).

Este protocolo establece no solo el formato de los mensajes que se intercambian entre nodos, sino también una arquitectura común de dispositivo a través de objetos de comunicación y parámetros de configuración. Esto posibilita que aparatos de distintos fabricantes se comuniquen de manera estandarizada.

Uno de los componentes esenciales de CANopen es el diccionario de objetos, que funciona como una tabla que almacena todos los parámetros y variables de proceso de cada nodo.

Respecto a la clase de mensaje, CANopen se organiza en dos grandes categorías:

- **SDO (Service Data Object)**, facilita el acceso directo a cualquier elemento del diccionario de objetos, ya sea para lectura o para escritura. Se emplea para ajustes no regulares o cuando se requiere acceder a parámetros concretos del dispositivo. Es un método de comunicación enfocado en cliente-servidor.
- **PDO (Process Data Object)**, empleados para el intercambio de datos en tiempo real. Son mensajes compactos que transmiten los valores de ciertas variables definidas en el diccionario. Están ajustados para la rapidez y no contienen encabezados extras.
  - **TPDO (Transmit PDO)**: Datos enviados por un nodo al resto.
  - **RPDO (Receive PDO)**: Datos que recibibe un nodo desde otros.

Además, CANopen incluye otros tipos de mensajes como los NMT (Network Management) usados para gestionar, por ejemplo, el estado (operacional/preoperacional) de los nodos, o el Heartbeat, que permite supervisar la presencia y estado de los dispositivos conectados a la red.

Esta estructura facilita el diseño de proyectos como el actual en el que se incluyen dispositivos de distintos fabricantes que tienen que comunicarse e interactuar entre sí.

2.7. Comunicación por radio

La comunicación por radio es una tecnología que permite la transmisión de información entre dos aparatos sin necesidad de enlace físico, utilizando en su lugar ondas electromagnéticas. Es especialmente útil en entornos donde no es viable establecer conexiones cableadas ya sea por movilidad, distancia o seguridad.

En los sistemas de control remoto industrial como los utilizados en maquinaria pesada, grúas, vehículos autónomos o robots móviles, la comunicación por radiofrecuencia permite

enviar órdenes y recibir información desde un terminal remoto, lo que facilita el control a distancia y protege al operador de situaciones peligrosas.

Figura 2.7: Control remoto de maquinaria pesada. Fuente: [15]

Un sistema básico consta de dos elementos: un **transmisor** que genera y envía la señal, y un **receptor** que la capta e interpreta. La información se codifica y se transmite mediante ondas de radio en una banda de frecuencia determinada, normalmente dentro del espectro de uso libre, como 433 MHz, 868 MHz o 2.4 GHz, dependiendo de la normativa vigente. La modulación más habitual en estos sistemas es la digital, que permite transmitir datos binarios de forma robusta, incluso en presencia de interferencias.

Las principales ventajas de esta tecnología en entornos industriales y móviles son:

- **Movilidad**: permite al operador moverse libremente sin estar conectado físicamente al vehículo o sistema.
- **Seguridad**: aumenta la seguridad del operador al mantenerlo a distancia de ciertas zonas peligrosas.
- **Rapidez de despliegue**: no requiere cableado ni instalaciones físicas.
- **Flexibilidad**: puede configurarse para distintas funciones, equipos y protocolos (CAN, puerto serie, etc).

No obstante, también presenta algunas limitaciones que deben tenerse en cuenta:

- **Interferencias**: aunque no es lo habitual por los mecanismos de seguridad que se incorporan, la comunicación puede verse afectada por otros dispositivos que operen en la misma banda de frecuencia.
- **Alcance limitado**: dependiendo de la potencia y el entorno, el alcance puede verse reducido por obstáculos físicos.

- **Seguridad**: si no se emplean mecanismos de emparejamiento o autenticación, puede haber riesgos de interferencia o suplantación de señales.

Para poder interactuar con otros componentes electrónicos, como microcontroladores o unidades de control, los sistemas de radiocomunicación incluyen interfaces de comunicación digital, como UART, SPI o, más comúnmente en entornos industriales, CAN (Sección 2.6). Este último permite una integración directa con buses de datos robustos, como los que se emplean en vehículos.

Además, es común que los receptores dispongan de salidas digitales o analógicas para activar directamente actuadores o comunicar eventos, aunque en sistemas avanzados toda la comunicación se hace mediante bus CAN.

CAPÍTULO 3

Hardware utilizado

En este capítulo se describen en detalle los componentes físicos que conforman el sistema de vehículo eléctrico. Se presentan las características técnicas y las funcionalidades de los dispositivos comerciales específicos que permiten implementar las tecnologías presentadas en el capítulo anterior.

3.1. Motores PDM18

En el contexto de este proyecto, se ha optado por sustituir el único motor de 7.5 kW incluido en el diseño original del vehículo [3] por cuatro motores de corriente alterna de imanes permanentes, concretamente los PDM18. Estos motores tienen una potencia continua de 12.9 kW y una potencia pico de 30 kW [16], lo que mejorará notablemente las prestaciones.

Figura 3.1: Motor PDM18. Fuente: [17]

El modelo PDM18 pertenece a la categoría de motores de imanes permanentes internos (IPM), ya que sus imanes de neodimio están incrustados en el rotor. Esta configuración permite mejorar la eficiencia, disminuye las pérdidas y amplía el rango de velocidades mediante técnicas de control avanzadas. Además, el uso de imanes incrustados en el rotor mejora la estabilidad del motor a altas velocidades y permite una mejor distribución de las fuerzas electromagnéticas en el rotor.

Las principales características del motor se resumen en la Tabla 3.1.

Parámetro	**Valor**
Tipo de motor	PMAC IPM
Peso	24 kg
Tensión requerida	24-96 VDC
Potencia continua	12.9 kW (a 96 VDC)
Potencia de pico	30 kW
Velocidad máxima	8 000 rpm (prueba de rotura a 11000 rpm)
Eficiencia	92 %
Protección	IP65
Sistema de aislamiento	UL Clase H
Estándares de diseño	NEMA 1004
Rodamiento	NTN6006 LLUC3/5K
Sensor de posición	Encoder seno/coseno
Especificaciones de corriente y par	
Corriente continua	150 A
Corriente de pico	450 A
Par continuo	34 Nm
Par de pico	120 Nm
Especificaciones del rotor	
Tipo de imanes	Neodimio
Temperatura nominal	150°C
Forma de la FEM inducida	Sinusoidal
Recubrimiento de los imanes	Níquel
Estrategias de control	*Field Weakening*

Tabla 3.1: Especificaciones técnicas del motor PDM18. Fuente: [16]

Entre otras cosas, y atendiendo a la tabla anterior, estos motores han sido seleccionados por su idoneidad en aplicaciones de movilidad eléctrica como la del vehículo de este proyecto. Su diseño permite operar en un rango de tensión de 24 a 96 VDC, con una velocidad de rotación de hasta 8000 rpm (aunque mediante tests de rotura se sabe que llega hasta las 11000 rpm), además la carcasa del motor cuenta con una protección IP65, lo que lo hace resistente al polvo y al agua, garantizando su fiabilidad en todo tipo de entornos.

Asimismo, las curvas de par y potencia se pueden hallar a partir de la siguiente tabla que proporciona el fabricante:

Motor Speed (RPM)	Peak Torque (N·m)	Mechanical Peak Power (kW)
0	120,00	-
1 800	120,00	22,62
2 000	116,00	24,29
2 500	102,00	26,70
3 000	86,00	27,02
4 000	67,00	28,06
5 000	56,00	29,32
6 000	49,00	30,79
7 000	43,00	31,52
8 000	38,00	31,83

Tabla 3.2: Datos de las curvas de Par y Potencia del motor PDM18. Fuente: [18]

Lo que gráficamente se puede expresar como:

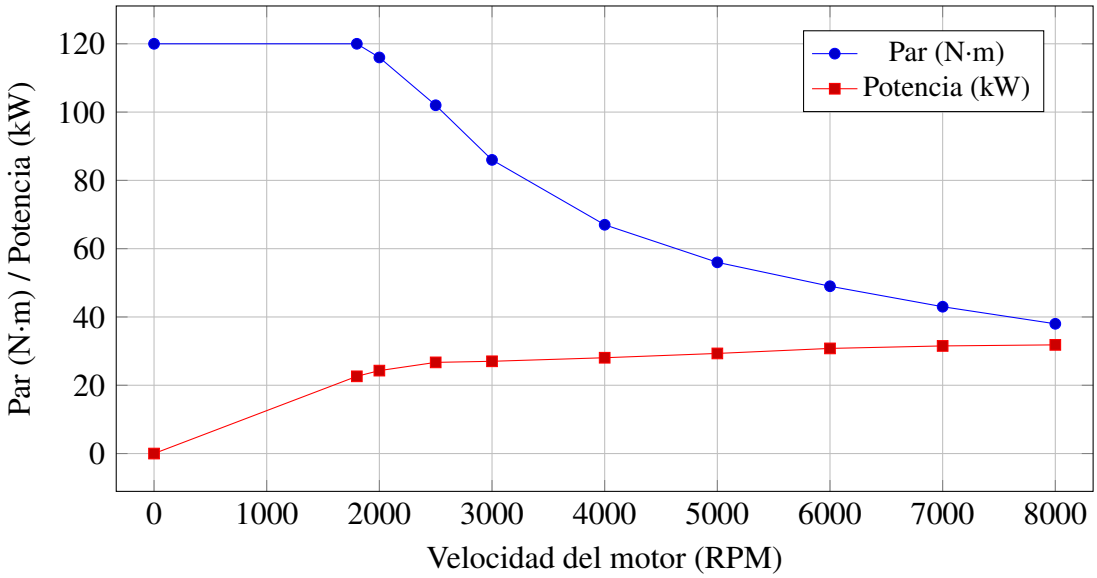


Figura 3.2: Curvas de Par y Potencia del motor PDM18. Fuente: Elaboración propia.

3.1.1. Encoder

Para el control preciso del motor, los PDM18 están equipados con un encoder de tipo seno/coseno, que permite conocer en todo momento la posición del rotor con respecto al estátor, así como su velocidad de giro. Estos sensores requieren una alimentación de 5 V

y tienen una capacidad de medición de hasta 60000 rpm, lo que los hace adecuados para el rango de operación del motor.

El encoder emite una señal de onda tipo seno o coseno completa por cada vuelta del rotor, con una amplitud de 1,1 V y una precisión de ±0, 6°. Gracias a esta información, la unidad de control del motor puede ajustar de manera precisa la conmutación de las fases y optimizar la entrega de par en función de la velocidad y la carga aplicada.

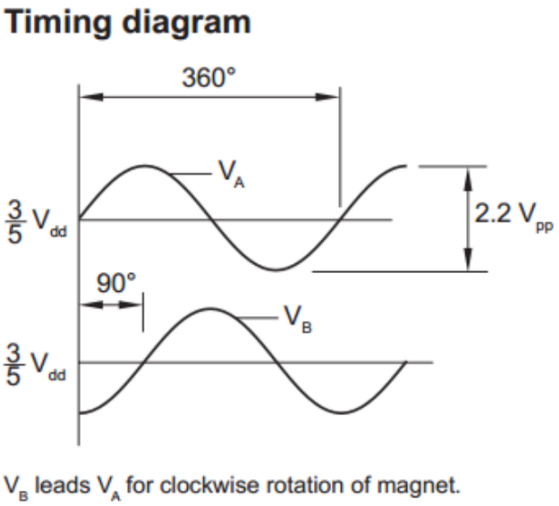


Figura 3.3: Comportamiento del *encoder*. Fuente: [16]

Especificaciones del *encoder*	
Alimentación	5 V ± 5 %
Rango de temperatura	-40°C a 105°C
Velocidad máxima medida	60 000 rpm
Resolución	1 onda seno/coseno por revolución
Salida seno/coseno	1,1 V ± 0,2 V
Consumo de potencia	20 mA
Precisión	±0, 6°
Histéresis	1,62° a 30 000 rpm

Tabla 3.3: Especificaciones técnicas del encoder. Fuente: [16]

3.2. Vanguard Lithium Ion Battery

Las baterías seleccionadas por su compatibilidad tanto con el motor (Sección 3.1) como con las controladoras (Sección 3.3) son las Vanguard 48V 5kWh Lithium Ion Battery.

Estas baterías de ion-litio operan con una tensión nominal de 48 V y cuentan con una capacidad de almacenamiento de 5 kWh. Su diseño modular permite la conexión en paralelo de hasta 16 baterías, lo que aumenta la capacidad total del sistema sin comprometer la estabilidad. En el vehículo actual se conectan en serie dos grupos de dos baterías en paralelo para alcanzar los 96 VDC.

Figura 3.4: Batería Vanguard 48V 5kWh Lithium Ion Battery Fuente: [19]

Entre sus principales características destacan:

- **Ciclo de vida prolongado**: Diseñadas para soportar hasta 2 000 ciclos de carga y descarga sin una degradación significativa.
- **Sistema de gestión de batería (BMS)**: Cuenta con un BMS avanzado que gestiona la temperatura, protege contra sobrecargas y cortocircuitos, y permite la integración mediante bus CAN.
- **Eficiencia energética**: Su tecnología de ion-litio presenta una eficiencia de carga y descarga superior al 95 %, minimizando pérdidas energéticas.
- **Modularidad y potencia**: Cada unidad proporciona 100 A (5 kW) de potencia continua y puede alcanzar 15 kW durante 10 segundos o 25 kW durante 1 en momentos donde haya picos de demanda.
- **Recarga**: El tiempo de recarga en condiciones normales es de 6 horas.
- **Protección y durabilidad**: Clasificación IP66, lo que garantiza resistencia a polvo y agua, y, como extra, posee protección contra lavado a presión.

Además de lo comentado anteriormente acerca del BMS, este sistema también permite que la batería tenga distintos modos de operación según el momento y las necesidades del sistema: descarga, carga o híbrido. Estos modos se establecen externamente a través de señales digitales que se conectan al conector principal de la batería. Por ejemplo, para habilitar la descarga es necesaria una señal de 5 V (que se obtiene del pin 9) a través del pin

2. Esta lógica de activación permite mantener la batería desconectada cuando el sistema está apagado, lo que mejora tanto la seguridad como la eficiencia.

Una de las ventajas principales de estas baterías es su composición química basada en fosfato de hierro y litio ($LiFePO_4$). En comparación con otras tecnologías, esta ofrece una mayor seguridad al ser más estable térmicamente, también tiene una vida útil sensiblemente más larga y tolera mejor las temperaturas extremas.

Cada batería tiene unas dimensiones de 57 cm de largo, 27 cm de ancho y 36 cm de alto, con un peso de 45 kg. Su temperatura de operación varía entre -30 °C y 60 °C en descarga, y entre 0 °C y 50 °C en carga.

Estas baterías cumplen con la certificación UN38.3, que garantiza que han superado pruebas rigurosas de seguridad, incluyendo térmicas, de vibración, cortocircuito, sobrecarga e impacto, entre otras, y son aptas para su transporte seguro por aire, mar y tierra.

Para la monitorización y el diagnóstico, la batería se comunica mediante el protocolo CAN bajo el estándar J1939 con una trama de 29 bits. A través de esta interfaz se puede acceder a información como el estado de carga (SOC), estado de salud (SOH), temperatura y posibles avisos de problemas.

A modo de resumen, en la Tabla 3.4 se muestran las principales características técnicas de estas baterías.

Parámetro	**Valor**
Tecnología química	$LiFePO_4$ (litio-fosfato de hierro)
Voltaje nominal	48 V
Capacidad energética	5 kWh
Potencia pico (10 s / 1 s)	15 kW / 25 kW
Vida útil estimada	> 2 000 ciclos
Tiempo de recarga	6 h (modo estándar)
Rango temp. operación (descarga)	-30 °C a 60 °C
Rango temp. operación (carga)	0 °C a 50 °C
Dimensiones (L × A × H)	57 × 27 × 36 cm
Peso	45 kg
Eficiencia energética	> 95 %
Protección	IP66 + resistencia a lavado a presión
Comunicación	CANbus J1939 (29-bit)
Configuración permitida	Hasta 16 en paralelo

Tabla 3.4: Especificaciones técnicas de las baterías Vanguard 48V 5kWh. Fuente: [20]

3.3. Controladora Sevcon Gen4

Las controladoras seleccionadas para este proyecto son las Sevcon Gen4 Size 6, su función principal es convertir la tensión continua de 96 V suministrada por las baterías (descritas en la Sección 3.2) en corriente alterna, para alimentar a los motores PMAC (Sección 3.1) asegurando un control preciso de velocidad y par. Se utiliza una estrategia basada en FOC puesto que proporciona un control eficiente y suave en estos motores, lo que es especialmente útil en vehículos eléctricos.

Estos dispositivos permiten comunicación mediante bus CAN, facilitando su integración con el resto de sistemas del vehículo.

Figura 3.5: Controladora Sevcon Gen4 Size 6. Fuente: [21]

Entre sus principales características destacan:

- Soporte para motores PMAC, BLDC y de inducción AC.
- Control de orientación de campo (FOC) para un funcionamiento eficiente.
- Modulación por ancho de pulso (PWM) para regulación de potencia.
- Conectividad mediante bus CAN para integración con otros sistemas.
- Frenado regenerativo configurable.
- Protección contra sobrecarga y sobrecalentamiento.
- Algoritmos avanzados de control de velocidad y par.
- Entradas para sensores de corriente, temperatura y posición del rotor.

Al estar enfocado en un vehículo que pretende funcionar de forma autónoma, se ha decidido implementar un control de velocidad. Esta decisión se basa en la necesidad de

lograr un movimiento fácilmente regulable. Además, dado que la lógica de movimiento se va a diseñar basada en el modelo de Ackermann para los giros, es más intuitivo trabajar directamente con consignas de velocidad para cada rueda que usar valores de par.

3.3.1. Contactores

Las controladoras Sevcon Gen4 requieren contactores externos para su correcto funcionamiento. En este caso, se han seleccionado los contactores Albright SW200, los cuales son fundamentales para la gestión de la conexión y desconexión del sistema de potencia.

Figura 3.6: Contactores Albright SW200. Fuente: [22]

Estos contactores solo pueden cerrarse una vez que la tensión del sistema alcanza al menos 60 V. La propia controladora es la encargada de gestionar la activación y desactivación de estos dispositivos, asegurando una operación segura del sistema eléctrico.

Además, forman parte de la secuencia de arranque y parada del sistema: mientras el vehículo permanece en estado preoperacional, los contactores se mantienen abiertos, impidiendo la alimentación de los motores. Cuando se cambia al estado operacional, las controladoras permiten su cierre. Esta lógica garantiza que no haya tensión antes de que todo el sistema esté preparado, mejorando así la seguridad y evitando posibles daños eléctricos.

Algunas características destacadas de los contactores SW200:

- Corriente nominal de hasta 400 A.
- Compatible con sistemas de 96 V DC.
- Alta durabilidad mecánica (> 5 millones de operaciones).
- Protección contra cargas inductivas y resistivas.

3.4. PC Advantech UNO-2484G

El ordenador seleccionado para ser el núcleo de este proyecto es el Advantech UNO-2484G.

Figura 3.7: Advantech UNO-2484G Fuente: [23]

Se trata de un equipo con una construcción robusta en aluminio y sin cables internos, diseñado para entornos industriales donde existen condiciones adversas como vibraciones, polvo o temperaturas extremas. Está basado en una arquitectura modular y *fanless*, lo que le aporta fiabilidad y escalabilidad. Permite la incorporación de módulos de expansión mediante la tecnología iDoor de Advantech, entre ellos, la extensión CAN utilizada en este proyecto.

La Tabla 3.5 recoge las especificaciones técnicas más relevantes del UNO-2484G:

Especificación	**Valor**
Procesador	Intel Core i7-7600U (7ª gen)
Memoria RAM	8 GB DDR4 integrada (ampliable a 16 GB)
Almacenamiento	1 x mSATA + 2 x SATA 2.5"(RAID 0/1)
Puertos de red	4 x RJ45 Ethernet
Puertos USB	4 x USB 3.0
Puertos serie	4 x RS232
Vídeo	1 x HDMI, 1 x DisplayPort
Temperatura de operación	-20 °C a 60 °C
Alimentación	10 - 36 VDC
Consumo	55 W
Diseño	Fanless, carcasa de aluminio, IP40
Soporte iDoor	Sí, con módulo EKBE

Tabla 3.5: Características principales del UNO-2484G. Fuente: [24]

3.4.1. Extensión para manejar CAN

Para habilitar la comunicación con el bus CAN del vehículo, se ha empleado una extensión mediante módulos iDoor sobre el UNO-2484G. En concreto, se ha utilizado el módulo EKBE como segunda capa del sistema, que permite la instalación de hasta tres módulos mPCIe adicionales. En este proyecto se ha utilizado uno de estos slots para integrar el módulo PCM-26D2CA-BE.

Figura 3.8: Módulo de expansión EKBE instalado sobre el UNO-2484G. Fuente: [25]

El módulo EKBE proporciona la capacidad de expansión necesaria para incorporar funcionalidades específicas como el bus CAN utilizado en este caso. El módulo PCM-26D2CA-BE es el encargado de gestionar la comunicación CAN en el vehículo. Este componente permite la conexión a dos redes CAN independientes a través de conectores DB9, soportando el protocolo CAN 2.0 A/B y ofreciendo una protección de aislamiento de 2,500 VDC para una mayor fiabilidad en entornos adversos.

Figura 3.9: Módulo CAN PCM-26D2CA-BE con sus dos puertos DB9. Fuente: [25]

La Tabla 3.6 presenta las especificaciones técnicas más relevantes del módulo PCM-26D2CA-BE:

Especificación	Valor
Tipo de bus	PCI Express Mini Card Rev. 1.2
Conectores	2 x DB9 macho
Controlador CAN	NXP SJA-1000
Transceptor CAN	NXP TJA1051T
Protocolos compatibles	CAN 2.0 A/B
Velocidad máxima	Hasta 1 Mbps
Aislamiento óptico	2,500 VDC
Resistencia de terminación	120 Ω (seleccionable por jumper)
Compatibilidad software	Windows 10, Ubuntu 18.04/20.04/22.04
Dimensiones	51 x 30 x 12.4 mm
Temperatura de operación	-20 °C a 60 °C
Protección ESD	15 kV
Protección contra sobretensiones	1,000 VDC

Tabla 3.6: Características del módulo CAN PCM-26D2CA-BE. Fuente: [26]

3.4.2. Sistema operativo

El sistema operativo elegido es Linux Ubuntu en su versión 22.04, en primer lugar porque es un entorno estable, además de ser el adecuado para poder trabajar con ROS 2.

Figura 3.10: Sistema operativo Ubuntu. Fuente: [27]

Las principales ventajas que puede ofrecer Linux como sistema operativo para un ordenador industrial (como es este caso), respecto a otras opciones como podría ser Windows son las siguientes:

- **Fiabilidad**: en un sistema que necesite estar funcionando de forma prolongada, se debe elegir un S.O. que no sea propenso a fallos o reinicios inesperados y que incluso no requiera actualizaciones automáticas que interrumpan la operación.
- **Rendimiento**: Linux tiene un consumo de recursos (RAM, CPU) muy bajo además de no necesitar una interfaz gráfica pesada ni procesos en segundo plano que no sean estrictamente necesarios.
- **Seguridad**: es muy poco vulnerable a virus y malware y minimiza el riesgo de ejecución de software malicioso.

- **Compatibilidad con software de código abierto**: la mayoría de herramientas industriales, incluida ROS 2, tienen mejor compatibilidad y soporte en Linux.
- **Compatibilidad con protocolos industriales**: en este caso, para usar el protocolo CAN, no se necesitarán drivers específicos o software propietario como sí ocurriría en caso de utilizar, por ejemplo, Windows. Además de ser mucho más indicado para procesos en tiempo real.

3.5. Equipo de radiocontrol Scanreco

Para implementar esta funcionalidad se opta por la marca Scanreco, ampliamente utilizada en sectores como la construcción o el rescate. El paquete seleccionado incluye tanto el control remoto (emisor) como el receptor, y destaca por su fiabilidad, compatibilidad con el protocolo CAN y facilidad de integración con los sistemas del vehículo.

Figura 3.11: Logo de Scanreco. Fuente: [15]

A continuación, se describen los dos componentes principales del sistema de radiocontrol elegido.

3.5.1. Mando Scanreco Mini

Figura 3.12: Mando Scanreco Mini. Fuente: Elaboración propia

El mando a distancia empleado en este proyecto es el modelo *Mini* de la marca Scanreco (Figura 3.12). Es una unidad bastante ligera, creada para ser utilizada durante largos

intervalos en aplicaciones que requieren un control remoto exacto y seguro. Su estructura modular posibilita la personalización de sus controles, permitiendo ajustar la cantidad y tipo de componentes (joysticks, botones, interruptores, etc.) a las exigencias del proyecto.

En este caso, el control se ha configurado con dos joysticks que permiten manejar con exactitud tanto la velocidad como la dirección del vehículo; estos están diseñados para producir señales digitales que se envían de manera continua al receptor.

Aparte de los mencionados joysticks, el control cuenta con botones auxiliares que pueden asignarse a funciones específicas, como activar modos de operación, gestionar el arranque o paro del sistema, o bien interactuar con otros componentes del vehículo. Es importante mencionar que el mando incorpora sistemas de seguridad como la seta de emergencia central, que hay que pulsar para apagar el mando, y que debe ser rearmada para volver a encenderlo.

El dispositivo cuenta con baterías recargables; además, al incluir una pareja, se puede hacer un uso continuado durante varias horas. La comunicación entre el mando y el receptor se realiza mediante radiofrecuencia, concretamente en la banda de 433 MHz, garantizando un alcance adecuado incluso en entornos complejos.

Figura 3.13: Batería recargable del mando. Fuente: [15]

3.5.2. Receptor Scanreco G3B

Figura 3.14: Receptor Scanreco G3B. Fuente: [15]

El dispositivo receptor empleado en este proyecto es el modelo G3B (Figura 3.14), también de la casa Scanreco. Es una unidad que ha sido extensamente probada en aplicaciones móviles y que está diseñada para recibir las señales que transmite el mando Scanreco Mini (Subsección 3.5.1) mediante radiofrecuencia y enviarlas, en este caso al ordenador del vehículo, utilizando el protocolo de comunicación CAN.

Una característica clave del G3B es su carcasa robusta y estanca, capaz de soportar condiciones difíciles (a las que se prevé que estará sometido este vehículo) como polvo, humedad o vibraciones con plenas garantías de funcionamiento. Su construcción está orientada a ofrecer una alta fiabilidad en condiciones exigentes, como las que enfrentan vehículos todoterreno, maquinaria pesada o, en este caso, un vehículo de rescate.

Mediante la configuración interna del receptor es posible modificar una gran cantidad de parámetros, desde la velocidad del bus CAN hasta los identificadores de los mensajes, entre otros aspectos; esto simplifica su integración con los demás nodos del sistema. Además, el procedimiento de vinculación con el mando de control remoto garantiza una comunicación exclusiva, previniendo interferencias con otros dispositivos Scanreco que pudieran estar en el mismo entorno.

El receptor se alimenta directamente desde el sistema eléctrico del vehículo (toma auxiliar de 12 V de las baterías, Sección 3.2), y puede incluir salidas adicionales para controlar actuadores o señales auxiliares, si fuera necesario. En el contexto del presente proyecto, su uso se centra principalmente en la interpretación de los movimientos del joystick y en el reconocimiento del estado de ciertos botones o interruptores que tendrán distintas funciones, lo cual será desarrollado más adelante en las secciones de implementación.

En conjunto, el sistema de radiocontrol Scanreco proporciona una interacción precisa y segura con el vehículo. Su integración con el resto de componentes del sistema se tratará en los siguientes capítulos.

CAPÍTULO 4

Conexionado del hardware

En este capítulo se detalla el conexionado físico de todos los componentes del vehículo Panter con los que se va a trabajar. Se aborda la organización e interconexión de todos los componentes hardware principales, incluyendo el PC Advantech, la extensión EKBE con su módulo PCM-26D2CA-BE para la comunicación por bus CAN, las controladoras SEVCON, el receptor de radio Scanreco y los motores de tracción. Se especifica también la alimentación mediante el sistema de baterías y las conexiones de todos los sensores y entradas de las controladoras.

4.1. PC de control principal

En este proyecto, el módulo de expansión EKBE junto con el PCM-26D2CA-BE fueron suministrados ya ensamblados por el proveedor, lo que simplifica su integración con el ordenador principal. Para acoplar el conjunto al Advantech UNO-2484G, se desmonta la tapa inferior del equipo, se fija el módulo mediante tornillos y se recoloca la tapa original en la parte inferior del nuevo conjunto. La Figura 4.1 muestra el resultado final del montaje. Este módulo permite habilitar la comunicación con dispositivos a través del bus CAN.

Figura 4.1: Parte delantera y trasera del PC ensamblado. Fuente: Elaboración propia.

En el interior del equipo se ha incorporado un disco duro SATA de estado sólido (SSD) de 500 GB, destinado al sistema operativo y al almacenamiento de archivos.

Figura 4.2: Disco duro de la marca WD Blue utilizado. Fuente: [28]

Respecto a la alimentación, se utiliza inicialmente el adaptador incluido, que presenta por un extremo un enchufe macho europeo y por el otro dos cables sin conector estandarizado. Este adaptador incorpora un conversor de corriente alterna a continua, reduciendo los 230 V de la red eléctrica a una tensión de salida comprendida entre 10 y 36 V, compatible con la especificación del Advantech UNO-2484G (véase la Tabla 3.5).

Antes de realizar la conexión, se verifica la polaridad de los conductores mediante un multímetro, con el fin de evitar posibles daños al equipo derivados de una inversión de polaridad. Finalmente, los cables se conectan a la entrada de alimentación del UNO-2484G, que dispone de terminales de tornillo.

4.2. Receptor de radio Scanreco

El receptor de radio Scanreco se conecta al sistema utilizando dos mangueras M12 de 5 pines: la manguera A, utilizada para la operación, y la manguera B, destinada a la configuración. La manguera C no se utilizará en este caso.

4.2.1. Manguera A (Operación)

La manguera A es un conector M12 de 5 pines macho que se utiliza para la conexión con alimentación y el bus CAN. Los pines de la manguera A están asignados de la siguiente manera:

- Pin 1: DV1 (sin conectar)
- Pin 2: Alimentación + (12 - 24 V)
- Pin 3: GND

- Pin 4: CAN_H
- Pin 5: CAN_L

Para establecer la conexión con el sistema de alimentación y el bus CAN, se emplea un conector M12 de 5 pines hembra. En este caso, las conexiones de los 5 cables se realizan mediante tornillos, lo que proporciona una conexión firme y segura. La Figura 4.3 muestra el conector utilizado para la manguera A.

Figura 4.3: Conector M12 de 5 pines hembra utilizado para la manguera A. Fuente: [29].

4.2.2. Manguera B (Configuración)

La manguera B es un conector M12 de 5 pines hembra que se utiliza para la configuración y la comunicación con el mando de radio. Los pines de la manguera B están asignados de la siguiente manera:

- Pin 1: DATA (sin conectar)
- Pin 2: GND
- Pin 3: RS 232 TX
- Pin 4: RS 232 RX
- Pin 5: Salida alimentación (sin conectar en modo configuración)

Para realizar la conexión de la manguera B, se utiliza un conector M12 de 5 pines macho, al que se conectan tres cables en los pines 2, 3 y 4. Estos cables se sueldan a los pines 5, 2 y 3 respectivamente de un conector DB9 hembra siguiendo el manual [30]. A su vez, este DB9 hembra se conecta a un adaptador USB a puerto serie, como el que se muestra en la Figura 4.5. Este adaptador permite la comunicación con el sistema a través de un puerto serie.

Figura 4.4: Conector M12 de 5 pines macho utilizado para la manguera B. Fuente: [29].

Figura 4.5: Adaptador USB a puerto serie. Fuente: Elaboración propia.

La manguera B también se utiliza para la comunicación cableada con el mando, lo que permite operar sin necesidad de usar la conexión inalámbrica. Además, sirve para hacer el emparejamiento inicial de los dos dispositivos y que el receptor solo procese señales de ese mando en concreto. En caso de establecer la comunicación por cable, es necesario extraer la batería del mando, ya que la manguera B proporciona la alimentación requerida a través del pin 5.

4.2.3. Manguera C (No utilizada)

La manguera C es un conector M12 de 5 pines macho que, aunque no se utiliza en este caso específico, está disponible para otras configuraciones del sistema. En concreto, la manguera C se emplea en sistemas donde se requieren bucles de control u otras configuraciones de comunicación adicionales. Dado que no es necesaria en este proyecto, se mantiene sin conectar.

4.3. Controladoras SEVCON

Esta sección está más detallada en el Trabajo de Fin de Grado de Rafael Onieva Molina [1], como ya se mencionó en la Sección 1.1 Antecedentes.

El sistema cuenta con dos controladoras SEVCON Gen4 (Sección 3.3), cada una de ellas conectada a un motor trifásico mediante sus salidas M1, M2 y M3, de cada uno de

ellos parten también un encoder de tipo seno/coseno y un termistor cuyas señales llegan directamente a la controladora correspondiente. El encoder se conecta a través de las entradas específicas U/V/W (pines 5, 17 y 29), y el termistor a través del pin 33 (utilizando el B- de la alimentación como referencia).

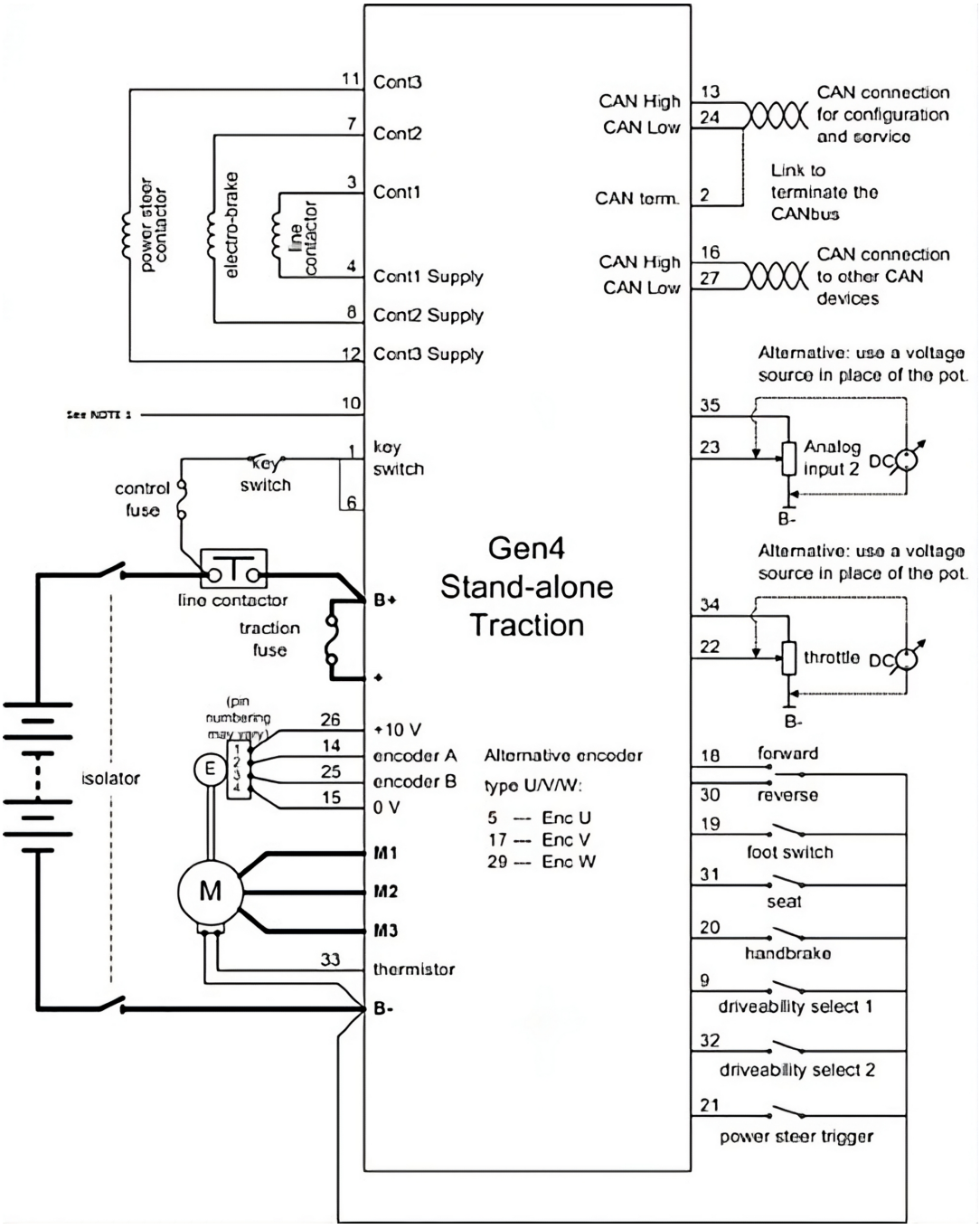


Figura 4.6: Diagrama de pines de la controladora. Fuente: [31]

Ambas controladoras reciben la alimentación mediante los bornes B+ y B- conectados a las baterías, y cada una está protegida por un contactor que ellas mismas se encargan de abrir o cerrar a través de los pines 3 (Cont1) y 4 (Cont Supply +).

La comunicación entre ellas se realiza a través del bus CAN, utilizando los pines 16 (CAN_H) y 27 (CAN_L). Además, la primera controladora se conecta con el PC a través de los pines 13 (CAN_H) y 24 (CAN_L).

Pin	Nombre	Función en el sistema
3	Cont1	Control del contactor de cada controladora
4	Cont1 Supply	Alimentación positiva para el contactor
5 / 17 / 29	Encoder U / V / W	Señales seno/coseno del el encoder del motor
13 / 24	CAN High / Low	Comunicación con el PC
16 / 27	CAN High / Low	Comunicación entre las dos controladoras
15 / 26	+10 V / 0 V	Alimentación del encoder
25	Encoder B input	Canal B del encoder seno/coseno
33	Motor thermistor in	Entrada del termistor del motor
M1, M2, M3	M1, M2, M3	Salidas trifásicas hacia el motor
B+ / B-	B+ / B-	Conexión de alimentación desde las baterías

Tabla 4.1: Conexiones relevantes de las controladoras. Fuente: [31]

4.4. Red de comunicación CAN

La red CAN del sistema se configura en línea, comenzando desde el puerto CAN del PC, que utiliza un conector DB9 del cual parten dos señales: `CAN_H` (desde el pin 7) y `CAN_L` (desde el pin 2). Estas líneas se conectan respectivamente a los pines 13 (`CAN_H`) y 24 (`CAN_L`) de la primera controladora SEVCON Gen4.

Desde esta primera controladora, las señales continúan hacia la segunda unidad utilizando los pines 16 (`CAN_H`) y 27 (`CAN_L`), conectados a los mismos pines en la segunda controladora. A partir de esta última, se establece la comunicación con el receptor Scanreco mediante la conexión de los pines 16 y 27 a los pines 4 (`CAN_H`) y 5 (`CAN_L`) de la manguera A del receptor.

Al principio, se encuentran activadas las resistencias de terminación de 120 Ω en las dos controladoras mediante un puente entre los pines 2 y 24 de cada una de ellas. Al añadir el PC y el receptor, se desactivan dichas terminaciones y se establece: por un lado uno de los extremos de la red CAN en el receptor Scanreco mediante software, como se detalla en la Subsección 5.2.1, y por otro el extremo opuesto en la salida CAN del PCM-26D2CA-BE (Subsección 3.4.1) mediante un jumper tal y como se indica en el manual [32], uniendo los pines 1 y 2 (Figura 4.7).

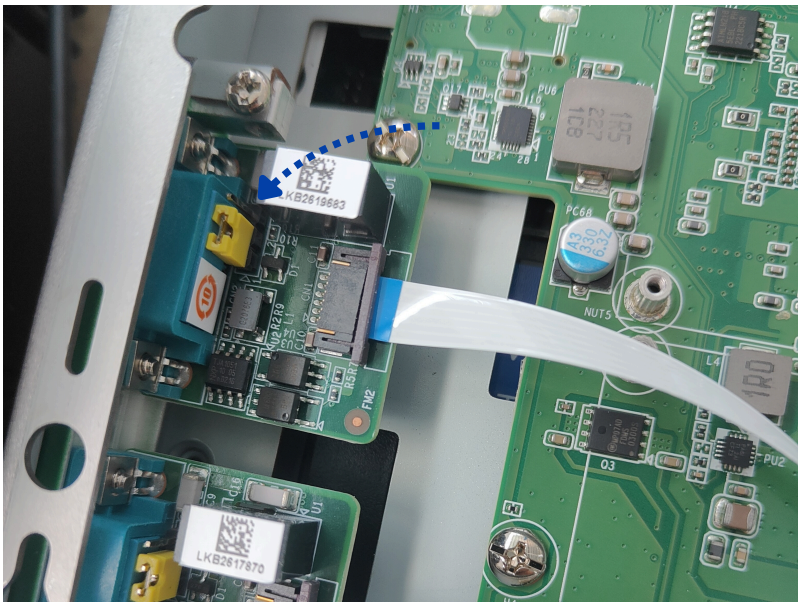


Figura 4.7: Resistencia de terminación del PCM-26D2CA-BE seleccionada mediante jumper. Fuente: Elaboración propia

En la Figura 4.8 se representa un esquema simple de las conexiones entre los dispositivos utilizados en este trabajo y, en la Figura 4.9, una imagen de la red final que se pretende alcanzar al finalizar el proyecto Panter completo.

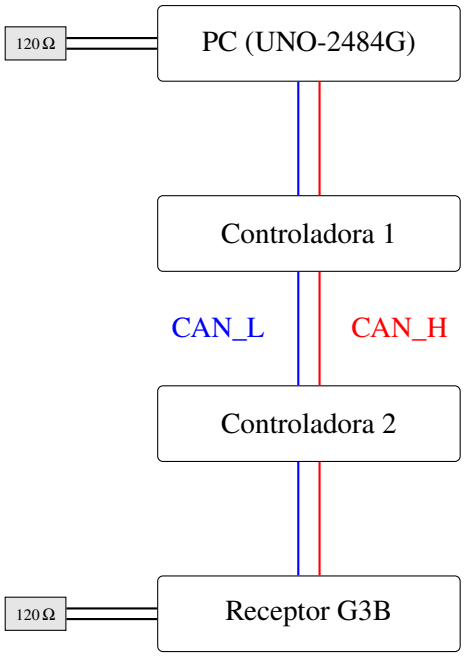


Figura 4.8: Diagrama simple del bus CAN completo. Fuente: Elaboración propia

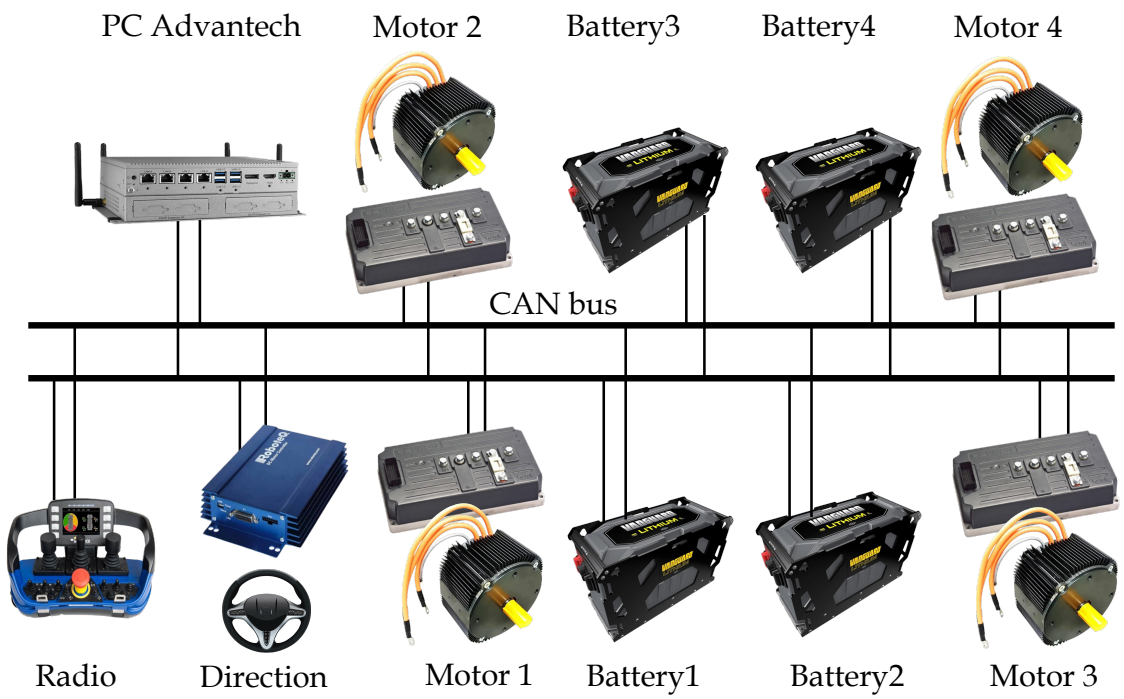


Figura 4.9: Esquema objetivo de nodos del bus CAN. Fuente: [33]

De este esquema se logra el funcionamiento en conjunto del PC, las dos controladoras junto a los dos motores traseros y el sistema de receptor y emisor de radio, así como la dirección, que se desarrolla paralelamente en otro proyecto pero que va integrada en el bus tal y como se indica y cuyo funcionamiento junto a los demás componentes ya está comprobado.

4.5. Suministro eléctrico

El suministro eléctrico del sistema parte de las cuatro baterías Vanguard descritas en la Sección 3.2, las cuales se han configurado en dos grupos en serie. Esta topología permite obtener una tensión total de 96 VDC, que será lo requerido por el conjunto de controladoras y motores.

Desde el punto de vista físico, cada par de baterías en serie se interconecta mediante un cable de potencia, uniendo el terminal positivo de una de ellas con el negativo de la otra. Como resultado, entre los terminales libres se tiene una tensión de aproximadamente 96 VDC, que puede ser superior cuando las baterías tengan un nivel alto de carga.

El terminal negativo de la asociación de baterías se conecta al terminal negativo (B-)

de dos de las controladoras, mientras que el positivo se dirige a un contactor individual asignado a cada una de ellas.

A continuación, el otro terminal de cada contactor se conecta al terminal positivo de la correspondiente controladora (B+), conexión que además está protegida mediante un fusible para prevenir fallos eléctricos. La activación de los contactores es gestionada por las propias controladoras, como se explica en la Sección 4.3.

Desde los bornes M1, M2 y M3 de las controladoras (Figura 4.6) parte el cableado trifásico hacia cada uno de los motores PDM18 (Sección 2.1), completando la ruta desde las baterías hasta los actuadores. Antes de establecer las conexiones, es necesario hacer una comprobación de la secuencia de fases de cada motor para poder conectarlos correctamente a cada una de las fases de salida de las controladoras. El procedimiento seguido para esto se detalla en el Trabajo de Fin de Grado de Rafael Onieva Molina [1] y el resultado final para ambos conjuntos fue el siguiente:

- **M1** de la controladora se conecta a **M1** del motor.
- **M2** de la controladora se conecta a **M3** del motor.
- **M3** de la controladora se conecta a **M2** del motor.

Por otra parte, la alimentación del ordenador industrial Advantech UNO-2484G se plantea inicialmente a través de su adaptador original, que incorpora un conversor de corriente alterna a continua (Sección 4.1). Sin embargo, en la integración definitiva del sistema, este adaptador se eliminará con el fin de conectar directamente el PC a otra batería, manteniendo la tensión en el rango especificado.

Por último, para la alimentación del receptor de radio (Sección 2.7), actualmente se emplea de forma provisional una fuente de tensión configurada para proporcionar 12 V. En una futura integración, este sistema provisional será sustituido por una de las salidas de 12 V auxiliares de una de las baterías (pin 3: 12 V y pin 6: tierra). La conexión se realizará a través de la manguera A descrita en la Sección 4.2

Por tanto, cada pareja de controladoras se alimenta mediante un grupo de dos baterías conectadas en serie, contando cada una con su propio contactor como medida de seguridad. El cableado de salida hacia los motores ha sido verificado para garantizar un funcionamiento correcto, y el receptor de radio cuenta con una fuente de alimentación provisional de 12 V, cuya sustitución por una conexión auxiliar de las baterías ya ha sido prevista. Además, el ordenador principal se alimentará directamente de otra batería, prescindiendo del adaptador de corriente original.

4.6. Diagrama final de conexiones

Para cerrar este capítulo, el diagrama de la Figura 4.10 sirve a modo de resumen de las conexiones que lleva el sistema mientras se están haciendo pruebas en el taller.

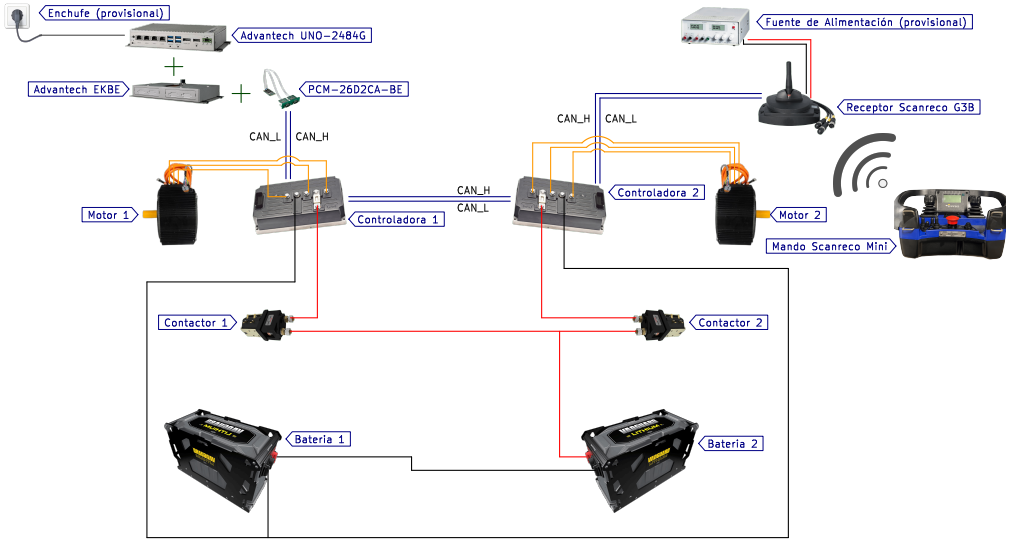


Figura 4.10: Diagrama de conexiones provisional. Fuente: Elaboración propia

Por otro lado, en la Figura 4.11 se representa el conexionado de la parte con la que se trabaja una vez se monte en el vehículo.

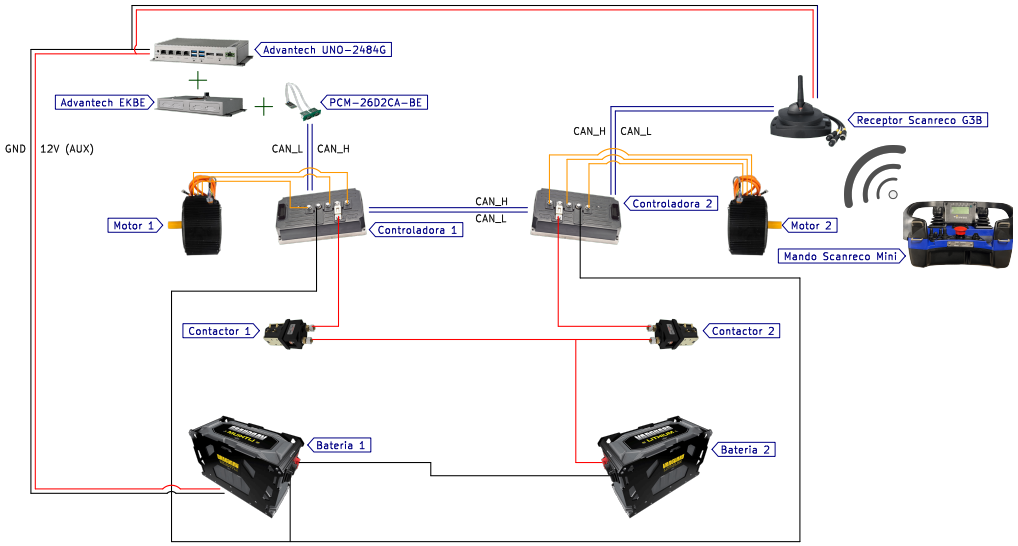


Figura 4.11: Diagrama de conexiones definitivas. Fuente: Elaboración propia

Hay que señalar que, por simplicidad, en ambas representaciones se han omitido las conexiones de los termistores y encoders seno/coseno de los motores a las controladoras y algunas otras conexiones, como las de activación de los contactores.

CAPÍTULO 5

Ejecución del proyecto

En este capítulo se describe detalladamente el procedimiento seguido y las distintas fases que se llevaron a cabo para la puesta en marcha y la correcta integración del sistema completo. Primero se explica la puesta en marcha del ordenador, abarcando la instalación del sistema operativo, la configuración de los puertos CAN, y la instalación de ROS 2 en su versión *Humble*, además de las librerías necesarias para el desarrollo del código.

A continuación, se trabaja la programación de los distintos componentes del sistema. Se describe la configuración del software del receptor Scanreco G3B, así como la adecuación de las controladoras Sevcon Gen4 para garantizar su compatibilidad con el resto del sistema.

Finalmente, se explica el software desarrollado, detallando su evolución desde una primera versión simple hasta la versión definitiva con todas las funcionalidades requeridas. Este proceso incluye el envío y recepción de mensajes CAN, la incorporación del movimiento mediante el joystick del mando (incluyendo los cálculos de Ackermann), y la posterior modularización del sistema con el fin de aprovechar las funcionalidades que ofrece ROS 2 con el uso de *topics*, *publishers* y *subscribers*. También se incluyen otras mejoras en la versión final, como la inclusión del manejo de botones y la disminución de tráfico en el bus.

5.1. Inicialización del Advantech UNO-2484G

Para empezar a usar el ordenador, es necesaria la instalación del sistema operativo elegido (Subsección 3.4.2) mediante una imagen obtenida del sitio web oficial [27]. Este proceso se explica con detalle en el Anexo B.1.

Una vez completada la instalación del sistema operativo, se procede con la instalación de ROS 2 *Humble* (Anexo B.2), al terminar la misma, se hace una prueba de funcionamiento con los archivos de ejemplo `talker` y `listener` que se incluyen por defecto. Para ello, se abre un terminal nuevo y se ejecuta:

```
ros2 run demo_nodes_cpp talker
```

Código 5.1: Ejecución del `talker`

Y se abre otro distinto en el que se ejecuta:

```
ros2 run demo_nodes_cpp listener
```

Código 5.2: Ejecución del `listener`

Si en el `listener` se puede ver lo que publica el `talker` se dará la instalación por completada.

Selección de librerías

Para el desarrollo correcto de los nodos de ROS 2 (escritos en lenguaje `C`), del compilador y demás recursos que utiliza ROS 2, es necesario instalar una serie de librerías. A continuación se deja una lista de las más importantes junto a una breve explicación:

- **rcl**: Proporciona la ROS Client Library para C, que contiene las funciones básicas necesarias para crear nodos, publishers y subscribers.
- **rclc**: Implementa una capa sobre `rcl`, facilitando la creación de nodos, la gestión de múltiples ejecutores y la organización estructurada de callbacks.
- **rclc executor**: Submódulo de `rclc` que permite la ejecución eficiente y ordenada de callbacks, gestionando internamente la suscripción a topics de forma optimizada.
- **gpp**: Biblioteca utilizada para la gestión de colas de mensajes y procesamiento paralelo de datos. Se utiliza principalmente durante la instalación de ROS 2.
- **gcc y g++**: Compiladores estándar para C y C++ en sistemas GNU/Linux. Son esenciales para construir los nodos del proyecto, asegurando la compatibilidad y el correcto funcionamiento de las aplicaciones en ROS 2. Se recomienda utilizar versiones recientes para evitar incompatibilidades con las características modernas de C++ utilizadas en ROS 2.

Una vez instalado el sistema operativo, se procede a configurar el módulo PCM-26D2CA-BE (Figura 3.9) montado mediante la extensión EKBE (Figura 3.8), que añade dos puertos CAN al PC. Este módulo se basa en el chip NKP-SJA1000 [26], por lo que es necesario instalar controladores específicos para que pueda ser reconocido por el sistema y se pueda utilizar con SocketCAN de Linux.

Una vez más, el proceso detallado de configuración se incluye en el Anexo B.3.

Tras finalizar la configuración, lo único que resta es configurar la velocidad de transmisión del bus CAN, que en este caso se establece a 500 kbps, utilizando el siguiente comando:

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

Código 5.3: Configuración y activación del puerto CAN

La interfaz `can1` no se utiliza en este caso, por lo que no es necesario activarla. Si en un futuro se quiere disponer de otra red CAN independiente, se activaría de la misma manera que `can0`.

Esta configuración (instalar drivers, cargar módulos y activar el puerto) hay que hacerla cada vez que se reinicia el ordenador. Para evitar esto, se crea un script de inicialización que realiza tanto la carga de módulos como la activación de la interfaz. Además, se incluye este script en la carpeta `bin` y se crea un archivo `systemd` para que se ejecute automáticamente al encender el PC; todo esto se encuentra desarrollado en el Anexo C.

Por último, para facilitar el envío y recepción de mensajes a través del bus, también se instala el paquete `can-utils`, que proporciona herramientas que serán útiles posteriormente en el desarrollo del código como `cansend`, `candump` o `cangen`:

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

Código 5.4: Instalación de herramientas de diagnóstico CAN

Tras todo esto, el sistema queda completamente preparado para la comunicación mediante bus CAN.

5.2. Programación de dispositivos

En esta sección se describen los ajustes necesarios para la correcta integración de los dispositivos principales del sistema: el receptor Scanreco G3B y las controladoras de motor Sevcon Gen4.

5.2.1. Programación interna del receptor Scanreco G3B

Para establecer una comunicación adecuada con el resto de dispositivos de la red CAN, es esencial hacer una serie de configuraciones internas en el receptor para asegurar que funcione como master con una tasa de transmisión adecuada al resto, entre otras cosas.

Para ello se utiliza el software WinSCI, solicitado al fabricante. Para realizar la programación, se debe conectar la manguera B del receptor (como se indica en la Sección 4.2) a

un puerto serie de un PC con Windows en el que se haya instalado el software de la forma que se indica en el manual del mismo [30].

Una vez seleccionado el modelo *G3B*, se descarga el programa que incluye el dispositivo de serie. Tras esto, y siguiendo el manual correspondiente [34] se procede a modificar los siguientes valores clave:

- **SWID1**: Originalmente configurado con el valor 003 (modo slave a 250 kbps), debe cambiarse a 010 (modo master a 500 kbps) [34].
- **SWID2**: El valor predeterminado es 100 (número de nodo), que se debe cambiar a 006. Para establecerlo como nodo 6, se han tenido en cuenta las dos controladoras que ya hay (nodos 1 y 2), las dos controladoras extra que se introducirán (nodos 3 y 4) y por último el sistema de dirección (nodo 5).
- **CAN_CONF**: Su valor inicial es 000, pero debe modificarse a 002 para activar la resistencia de terminación del bus (configuración de terminación de CAN) [34].

El código completo del Scanreco G3B no se adjunta a la memoria como anexo debido a su gran extensión. Está disponible para su consulta en el repositorio GitHub público correspondiente a esta sección del proyecto [35].

5.2.2. Configuración de las controladoras

La programación de las controladoras utilizada en este proyecto se basa íntegramente en el mapeo de objetos descrito en el trabajo de Rafael Onieva Molina [1], por lo que no ha sido necesario realizar cambios adicionales en la estructura de objetos CANopen.

No obstante, para su correcta integración en el sistema, es imprescindible realizar dos ajustes:

- **Velocidad del bus CAN**: Asegurar que ambas controladoras estén configuradas para operar a una velocidad de 500 kbps. Este ajuste debe verificarse y, en caso necesario, modificarse mediante el software DVT de Sevcon (utilizando también el PC con Windows).
- **Terminación del bus CAN**: Al haberse configurado el receptor Scanreco G3B como nueva terminación del bus (Subsección 5.2.1), es necesario desactivar la terminación física en las dos controladoras. Esto se realiza desconectando los pines 2 y 24 (esquema detallado en Figura 4.6), que originalmente están unidos para habilitar la resistencia de terminación interna.

Además, como se van a enviar consignas de acelerador, hay que asegurarse entre qué valores opera el mismo. Consultando el software DVT se comprueba que la controladora prevé mensajes de aceleración entre los 5 y los 6 V (Figura 5.1), por lo que se mandarán consignas de acelerador entre esos valores de la forma `0x00 0X05`, donde el byte 0

corresponde a los decimales y el otro a los 5 V del mínimo. El valor de los decimales puede ir desde 00 (5,00 V) hasta FF (5,99 V), por lo tanto hay 254 valores posibles dentro del rango, lo que da una precisión de 0,0039 V.

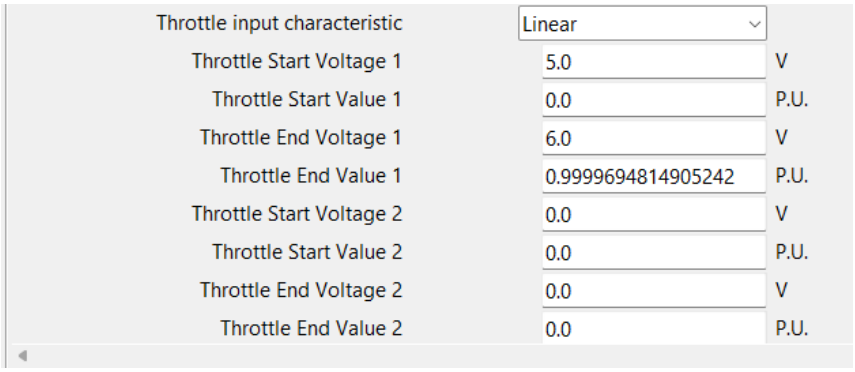


Figura 5.1: Rango del acelerador configurado. Fuente: Software DVT [36]

El diccionario completo de consignas se encuentra en el Anexo D.

5.3. Diseño del software

El diseño del software del vehículo sigue un desarrollo por etapas, en el que cada versión incorpora nuevas funcionalidades y corrige limitaciones de etapas anteriores. Se parte de una implementación básica, centrada en la comunicación por el bus CAN, y se evoluciona desde una siguiente versión que calcula velocidades de las ruedas mediante Ackermann hasta alcanzar una arquitectura modular que implementa estos mismos cálculos, procesamiento de distintas señales del mando a distancia (joysticks y algunos botones) y estructuración en nodos mediante ROS 2.

El desarrollo se organiza en dos paquetes: `tfg_jljg_inicial`, que contiene los primeros programas orientados a validar la comunicación y los cálculos de velocidad; y `tfg_jljg_completo`, que implementa una arquitectura modular basada en topics, facilitando la escalabilidad y el mantenimiento del sistema.

El workspace completo, con sus correspondientes archivos para la compilación, se encuentra disponible en otro repositorio de GitHub [37]. El código de cada uno de los nodos también está incluido para su consulta en el Anexo A de esta memoria.

5.3.1. Primera versión - Envío y recepción de mensajes

En la primera etapa, se comprueba que la comunicación a través del bus CAN funciona correctamente, para ello se crean dos programas en `C` que utilizan herramientas como

cansend y candump (del paquete can-utils) a través de llamadas al sistema. Estos programas se incluyen en el paquete tfg_jljg_inicial (Anexo A.1) y permiten realizar pruebas manuales sin necesidad de configurar directamente sockets ni estructuras avanzadas.

El objetivo es permitir tanto el envío manual de consignas CAN desde un terminal, como la recepción con posibilidad de filtrado de los mensajes del bus desde otro (útil en el desarrollo de siguientes versiones), con el fin de verificar el comportamiento del sistema.

Envío de mensajes CAN desde consola (envio_can_BASICO.c)

Este programa construye el comando completo al introducir de forma manual un mensaje en formato <COB_ID>#<DATA> para enviarlo a través del bus CAN utilizando la herramienta cansend. Por ejemplo, para una entrada como:

```
301#0000
```

el programa genera y ejecuta:

```
cansend can0 301#0000
```

El funcionamiento se resume básicamente en el siguiente fragmento:

```
1  char input[100];
2  fgets(input, sizeof(input), stdin);
3  input[strcspn(input, "\n")] = 0; // Elimina el salto de linea
4
5  if (strcmp(input, "exit") == 0) {
6      break;
7  }
8
9  char command[150];
10 snprintf(command, sizeof(command), "cansend can0 %s", input);
11 system(command);
```

Código 5.5: Construcción de mensajes en envio_can_BASICO.c

Esto permite enviar cualquier mensaje sin necesidad de recompilar y poder así realizar pruebas rápidas. Además, se incluye un mecanismo de cierre del programa introduciendo la palabra "exit" en el mismo terminal.

Recepción de mensajes CAN con filtro (receptor_can.c)

El segundo programa permite visualizar los mensajes que circulan por el bus CAN. Se basa en la herramienta candump e incluye la posibilidad de filtrar por COB-ID para

facilitar el seguimiento de mensajes específicos.

El usuario introduce hasta cinco COB-IDs separados por espacios. El programa construye un comando que ejecuta `candump` con los filtros adecuados. Por ejemplo, si el usuario introduce:

```
186 301
```

el programa genera el siguiente comando:

```
candump can0,186:7FF can0,301:7FF
```

lo cual limita la salida a los mensajes con COB-IDs `0x186` y `0x301`.

La parte de código encargada de construir estos filtros es:

```
char entrada[BUFFER_TAM];
fgets(entrada, sizeof(entrada), stdin);
entrada[strcspn(entrada, "\n")] = 0;

char *token = strtok(entrada, " ");
while (token != NULL && count < MAX_FILTROS) {
    char filtro_str[20];
    snprintf(filtro_str, sizeof(filtro_str), "can0,%s:7FF ", token);
    strcat(filtros, filtro_str);
    token = strtok(NULL, " ");
}
```

Código 5.6: Filtrado de mensajes en `receptor_can.c`

Si no se introduce ningún filtro, el programa ejecuta simplemente `candump can0`, lo que permite observar todos los mensajes del bus.

Esta primera versión del software permite comprobar rápidamente el estado de la comunicación CAN y entender cómo reaccionan los distintos nodos del sistema ante mensajes concretos. Además, facilita la verificación de los mensajes generados por el mando a distancia o los sensores, sin necesidad de desarrollar aún con ROS 2.

5.3.2. Segunda versión - Manejo con joysticks

En esta segunda versión del sistema de control, se abandona parcialmente el uso del teclado para controlar el vehículo, introduciendo en su lugar los joysticks del mando Scanreco Mini (Subsección 3.5.1). Esta nueva configuración permite controlar tanto la velocidad como la dirección, el procesamiento de las señales del joystick se realiza recibiendo los valores de ambos joysticks y calculando en tiempo real las velocidades individuales de cada rueda trasera según un modelo de cinemática Ackermann.

Dirección Ackermann

La dirección Ackermann es un modelo geométrico que permite que las ruedas de un vehículo describan trayectorias circulares con radios distintos durante un giro, lo que evita deslizamientos laterales y reduce el desgaste. Este principio establece que, durante una curva, todos los ejes de las ruedas deben coincidir en un punto: el centro instantáneo de giro.

Normalmente, este modelo se implementa ajustando mecánicamente el ángulo de giro de las ruedas delanteras mediante un sistema de dirección. Sin embargo, en este trabajo no se aborda el control de estas ruedas, sino que se utiliza el modelo Ackermann como una base teórica para calcular el radio de giro del vehículo y establecer una diferencia de velocidades en las ruedas traseras que permita un desplazamiento correcto del mismo.

A partir del ángulo máximo de giro de la rueda delantera interior $\theta_{\max}$ y del valor del joystick de dirección, se calcula un ángulo θ (en radianes). Con este ángulo se obtiene el radio de giro teórico:

$$R = \frac{L}{\tan(\theta)}$$

donde L es la distancia entre ejes. Las ruedas traseras, encargadas de la tracción, deben girar a velocidades distintas para seguir trayectorias curvas que respeten esa geometría:

$$R_{\text{int}} = R - \frac{w}{2}, \quad R_{\text{ext}} = R + \frac{w}{2}$$

donde w es el ancho de vía.

Partiendo de una consigna de velocidad v (calculada según el mapeo del joystick izquierdo, Figura 5.2), se ajustan las velocidades de las ruedas traseras siguiendo:

$$v_{\text{int}} = v \cdot \frac{R_{\text{int}}}{R_{\text{ext}}}, \quad v_{\text{ext}} = v$$

Se podría haber optado también por escalar las velocidades de la forma:

$$v_{\text{int}} = v \cdot \frac{R_{\text{int}}}{R}, \quad v_{\text{ext}} = v \cdot \frac{R_{\text{ext}}}{R}$$

Pero se mantiene la primera opción para que ninguna rueda supere nunca el valor de velocidad que se envía desde el joystick.

En este caso, las ruedas a las que se aplican estas velocidades son las traseras, ya que son las que ejercen la tracción; esto permite que el vehículo pueda trazar curvas sin que las ruedas derrapen.

Cálculo de velocidades según joysticks (`envio_can_ACK.c`)

El archivo `envio_can_ACK.c` implementa la lógica principal para calcular y enviar velocidades a cada una de las ruedas traseras en función de la lectura de los dos joysticks del mando Scanreco Mini, el comportamiento se basa en el modelo cinemático explicado en el apartado anterior, aplicando Ackermann para adaptar la velocidad de cada rueda al giro.

Por un lado, el sistema recibe entradas desde los ID 0x186 (joysticks), 0x201 y 0x202 (sentidos de marcha de los motores), y por otro lado emite consignas de acelerador a la controladora de cada motor con ID 0x301 y 0x302 (en el Anexo D se incluye el diccionario).

En primer lugar, se abre un socket asociado a la interfaz `can0` mediante la función `open_can_socket()`, que configura y enlaza la comunicación CAN:

```
s = open_can_socket();
if (s < 0) return 1;
```

Código 5.7: Inicialización del socket CAN

El programa se mantiene en un bucle de lectura continua, esperando tramas CAN de entrada. Al recibir una trama, se distingue su origen según el ID y se actualiza el sentido de marcha (0x07 para marcha alante, 0x0B para marcha atrás y 0x00 parado) o se procesa el movimiento:

```
switch (frame.can_id) {
case COBID_ESTADO_1:
estado_301 = frame.data[0];
break;
case COBID_ESTADO_2:
estado_302 = frame.data[0];
break;
case COBID_JOYSTICK:
calcular_y_enviar_velocidades(frame.data[1], frame.data[3]);
break;
}
```

Código 5.8: Gestión de mensajes entrantes por ID

La función `calcular_y_enviar_velocidades()` es la parte central del programa; a partir de los valores de aceleración y dirección que se reciben de los joysticks, calcula las velocidades para cada rueda teniendo en cuenta el sentido de giro correspondiente.

Primero, el valor del joystick de aceleración (byte 1 del ID `0x186`) se transforma mediante una función de mapeado, que genera una consigna entre 0x00 y 0xFD en función del sentido de marcha almacenado en los ID `0x201` y `0x202` (avance: 0x07 o retroceso: 0x0B):

```
1 unsigned char base_speed_301 = mapeo_joystick_velocidad(joystick_vel,
    estado_301);
2 unsigned char base_speed_302 = mapeo_joystick_velocidad(joystick_vel,
    estado_302);
```

Código 5.9: Mapeado de aceleración del joystick

La relación entre el valor del joystick de aceleración y la consigna enviada se representa en la siguiente figura, diferenciando el comportamiento en marcha adelante y marcha atrás. El punto neutro (0x7F) corresponde a velocidad cero, y desde ahí se escala linealmente hasta el valor máximo 0xFD:

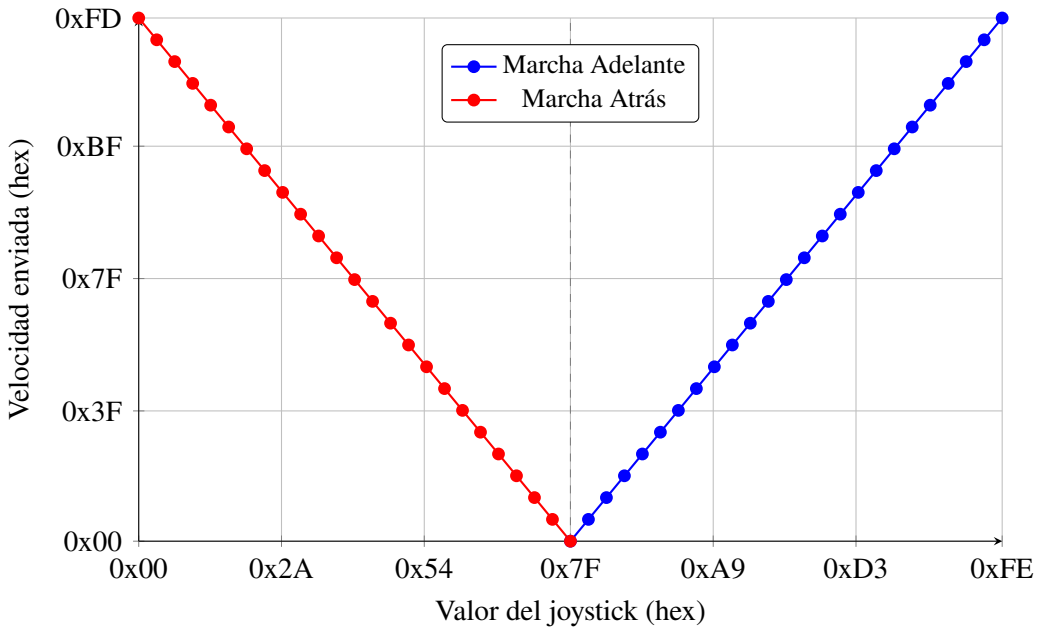


Figura 5.2: Mapeo de los valores del joystick de aceleración. Fuente: Elaboración propia

Tras esto, el valor del joystick de dirección (byte 3) se convierte en un ángulo de giro expresado en grados. Dicho ángulo se transforma en radianes para calcular el radio de giro, de acuerdo con el modelo Ackermann:

```
if (joystick_turn < 0x7F)
angulo_grad = -GIRO_MAX_GRAD * ((0x7F - joystick_turn) / 127.0);
else if (joystick_turn > 0x7F)
angulo_grad = GIRO_MAX_GRAD * ((joystick_turn - 0x7F) / 127.0);

double radio_giro = fabs(DISTANCIAEJES_MM / tan(angulo_grad * GRAD_A_RAD))
    ;
```

Código 5.10: Cálculo del ángulo y radio de giro

Una vez obtenido el radio, se calculan los radios interiores y exteriores de giro para las ruedas traseras, ajustando la velocidad de la rueda interior proporcionalmente. Como se ha explicado en el apartado teórico, se opta por escalar solo una rueda, de modo que la exterior mantenga siempre la velocidad de referencia, respetando así la consigna máxima impuesta por el joystick:

```
double Ri = radio_giro - (ANCHOVIA_MM / 2.0);
double Ro = radio_giro + (ANCHOVIA_MM / 2.0);
double ratio = Ri / Ro;

if (joystick_turn < 0x7F)
vel_izq *= ratio;
else if (joystick_turn > 0x7F)
vel_dcha *= ratio;
```

Código 5.11: Ajuste de velocidades para el giro

Por último, las velocidades calculadas se redondean y se envían a los controladores de motor mediante la función `send_can_message()`, que evita redundancias si la velocidad no ha cambiado respecto al último envío:

```
unsigned char final_301 = (unsigned char)(vel_izq + 0.5);
unsigned char final_302 = (unsigned char)(vel_dcha + 0.5);

send_can_message(s, COBID_CONTROLADORA_1, final_301, &ult_vel_301);
send_can_message(s, COBID_CONTROLADORA_2, final_302, &ult_vel_302);
```

Código 5.12: Envío de consignas de velocidad por CAN

Este sistema garantiza que el vehículo trace curvas coherentes con el modelo Ackermann, adaptando la velocidad de la rueda trasera interior según la geometría del giro y respetando la dirección y sentido de marcha.

Todas las funciones, junto con el programa completo, se pueden consultar en el Anexo A.1, donde se incluye el código `envio_can_ACK.c`, además de en el repositorio de GitHub mencionado anteriormente [37].

5.3.3. Versión final - Modularización e inclusión de botones

Finalmente y tras haber comprobado un funcionamiento correcto de los programas básicos de apartados anteriores, se hace una reorganización completa del proyecto con el objetivo de modularizarlo en distintos nodos aprovechando la arquitectura basada en topics de ROS 2. Esta modularización mejora tanto la legibilidad como el mantenimiento del código, y facilita además ampliaciones futuras.

Además, se incorporan nuevas funcionalidades como el uso de botones del mando Scanreco Mini para activar distintos modos de operación, y un comportamiento de giro sobre sí mismo, que será plenamente funcional una vez que el vehículo disponga de un motor independiente en cada rueda.

En las siguientes secciones se desarrollan las funcionalidades de cada nodo, su relación mediante topics y los archivos necesarios para la compilación y ejecución del sistema completo.

Asignación de botones del mando Scanreco Mini

En esta sección se describen las funcionalidades asignadas a los distintos botones del mando Scanreco Mini. En la Figura 5.3 se muestra la distribución de botones del mando utilizado para el control del vehículo junto a sus respectivos nombres.

Figura 5.3: Distribución de botones en el Scanreco Mini. Fuente: [38]

Los botones que se han seleccionado, junto a la función que desempeñan son:

- **S2**: Se utiliza para el encendido y apagado del motor. Es una palanca con retorno al centro: al desplazarla hacia la izquierda se manda la señal de arranque de los motores, mientras que hacia la derecha se solicita la parada.
- **S5**: Permite seleccionar el sentido de la marcha. Tiene dos posiciones fijas: hacia la izquierda se activa la marcha adelante, y hacia la derecha la marcha atrás.
- **S1**: Se encarga de seleccionar el modo de giro. Tiene tres posiciones; en la central se aplica el modelo de dirección tipo Ackermann, al moverlo hacia la izquierda se activa el modo de giro sobre sí mismo pensado para maniobras en espacios reducidos. Por último, la posición derecha no tiene función asignada.
- **S11**: Controla el limitador de velocidad. Cada pulsación hacia la izquierda reduce progresivamente el límite de velocidad al 85 %, 70 %, 50 %, 35 % y 20 % de la velocidad máxima. El estado del limitador se indica mediante un LED verde situado en el mando ("Led Micro"), que parpadea una, dos, tres, cuatro o cinco veces en función del nivel activo. Un único movimiento hacia la derecha desactiva el limitador, apagando el LED.
- **S13**: Se utiliza para encender el mando, siempre que previamente se haya rearmado la seta de emergencia que actúa como botón de apagado del mismo.

A continuación, en la Figura 5.4, se resume e indica la tipología de cada uno de los botones mencionados anteriormente:

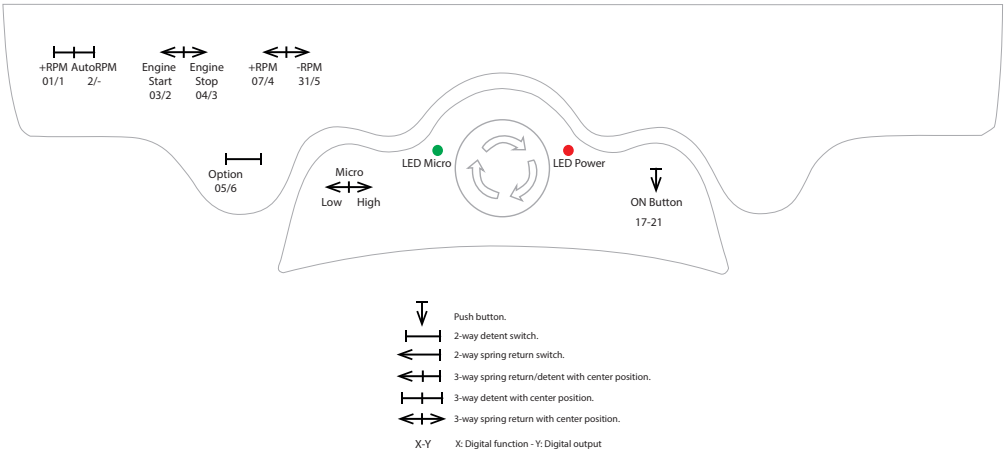


Figura 5.4: Tipología de cada botón. Fuente: [38]

Recepción de mensajes del bus: `receptor.c`

Este nodo se encarga de procesar los mensajes recibidos por el bus CAN a través de `can0` y publica la información relevante (joysticks y botones) en distintos topics.

Topic	Tipo	Descripción
joystick_acc	UInt8	Valor de aceleración del joystick izquierdo.
joystick_turn	UInt8	Valor de giro del joystick derecho.
sentido	UInt8	Sentido de la marcha (adelante/atrás).
giro	UInt8	Modo de conducción (Ackermann/giro).
arranque	UInt8	Estado de arranque/parada del sistema.
vmax	UInt8	Nivel de limitación de velocidad.

Tabla 5.1: Topics en los que publica `receptor.c`. Fuente: Elaboración propia.

Para ello, primero crea un socket tipo RAW:

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
bind(s, (struct sockaddr *) &addr, sizeof(addr));
```

Código 5.13: Creación del socket CAN `can0`

Una vez establecida la comunicación, el nodo procesa los mensajes recibidos con los identificadores `0x186` y `0x286`. Estos valores se corresponden con los indicados en el manual incluido en el Anexo E, según el cual los mensajes generados por un nodo CAN con ID 6 se transmiten en las direcciones `0x180` + ID = `0x186` y `0x260` + ID = `0x286`.

El identificador `0x186` contiene los datos de los joysticks. En concreto, se extraen dos valores: la aceleración (byte 1) y el giro (byte 3), que se publican en los topics `joystick_acc` y `joystick_turn`, respectivamente:

```
if (frame.can_id == COB_ID_JOYSTICK) {
msg_acc.data = frame.data[1];
msg_turn.data = frame.data[3];
rcl_publish(&pub_joystick_acc, &msg_acc, NULL);
rcl_publish(&pub_joystick_turn, &msg_turn, NULL);
}
```

Código 5.14: Procesamiento de mensajes del joystick (ID `0x186`)

El identificador `0x286` contiene la información de los botones en los distintos bits del byte 0, además del limitador de velocidad máxima en el byte 5. Del primero, se extraen los siguientes valores:

- El sentido de la marcha, indicado en el bit 5 (0 = atrás, 1 = adelante), que posteriormente se publica en el topic `sentido`.

- El modo de desplazamiento, según el bit 0 (0 = dirección tipo Ackermann, 1 = giro sobre sí mismo), que se publica en el topic `giro`.
- Orden de arranque/parada del sistema:
  - Si el bit 1 está activo, se interpreta como arranque y se publica el valor 1 en el topic `arranque`.
  - Si el que tiene valor 1 es el bit 2 (por limitación física del propio botón no pueden coincidir nunca), se publica el valor 2 en el mismo topic, que indica que se ha solicitado la parada.

```
uint8_t control_byte = frame.data[0];
msg_sentido.data = (control_byte >> 5) & 0x01;
msg_giro.data = control_byte & 0x01;
rcl_publish(&pub_sentido, &msg_sentido, NULL);
rcl_publish(&pub_giro, &msg_giro, NULL);

if ((control_byte >> 1) & 0x01) {
msg_arranque.data = 1;
rcl_publish(&pub_arranque, &msg_arranque, NULL);
} else if ((control_byte >> 2) & 0x01) {
msg_arranque.data = 2;
rcl_publish(&pub_arranque, &msg_arranque, NULL);
}
```

Código 5.15: Extracción de valores del byte de control

Por último, el valor de velocidad máxima (byte 5) se publica en el topic `vmax`:

```
msg_vmax.data = frame.data[5];
rcl_publish(&pub_vmax, &msg_vmax, NULL);
```

Código 5.16: Publicación del valor `vmax`

Los posibles valores de este byte son los siguientes:

Hex		Significado (Dec)
`0xC0`	⇒	Sin limitador (192)
`0xC1`	⇒	85 % de velocidad máxima (193)
`0xC2`	⇒	70 % de velocidad máxima (194)
`0xC3`	⇒	50 % de velocidad máxima (195)
`0xC4`	⇒	35 % de velocidad máxima (196)
`0xC5`	⇒	20 % de velocidad máxima (197)

Tabla 5.2: Valores de `vmax` posibles. Fuente: Elaboración propia.

Todos los mensajes se publican como enteros sin signo de 8 bits (`UInt8`), y además se imprimen en el terminal.

Para garantizar un cierre limpio del nodo, se instala un manejador de señal `SIGINT` que simplemente cambia una variable `running` a 0, lo que provoca la salida del bucle principal. Al finalizar, se cierran el socket y el nodo ROS ("`rcl_node_fini(&node)`") para liberar todos los recursos.

```
// Manejo de senal SIGINT (Ctrl+C) para terminar el bucle principal
void handle_sigint(int sig) {
  (void) sig; // Ignorar el argumento
  running = 0;
}

  // Cerrar socket CAN
  close(s);

  // Finalizar nodo ROS
  rcl_node_fini( & node);
```

Código 5.17: Manejador de señales y liberación de recursos

Cálculo del movimiento: `ackermann.c`

Este nodo calcula las velocidades que se deben aplicar en cada motor según el modo de conducción, el valor de los joysticks, el sentido de la marcha y la limitación de velocidad.

Topic	Tipo	Descripción
Suscripciones		
`joystick_acc`	UInt8	Aceleración del joystick.
`joystick_turn`	UInt8	Dirección del joystick.
`sentido`	UInt8	Sentido de marcha.
`giro`	UInt8	Modo de conducción (Ackermann/giro).
`vmax`	UInt8	Limitador de velocidad.
Publicaciones		
`motor_1`	Int32	Velocidad del motor trasero izquierdo.
`motor_2`	Int32	Velocidad del motor trasero derecho.
`sentido_ackermann`	UInt8	Sentido de desplazamiento (modo Ackermann).
`giro_izq`	UInt8	Sentido de giro del motor izquierdo (modo giro).
`giro_dcha`	UInt8	Sentido de giro del motor derecho (modo giro).

Tabla 5.3: Topics a los que se suscribe y donde publica `ackermann.c`. Fuente: Elaboración propia.

Al ser el nodo más extenso y complejo del sistema, en la Figura 5.5 se representa un diagrama de flujo del mismo que resume su funcionamiento y facilita su comprensión y, posteriormente, se explica parte por parte como el resto.

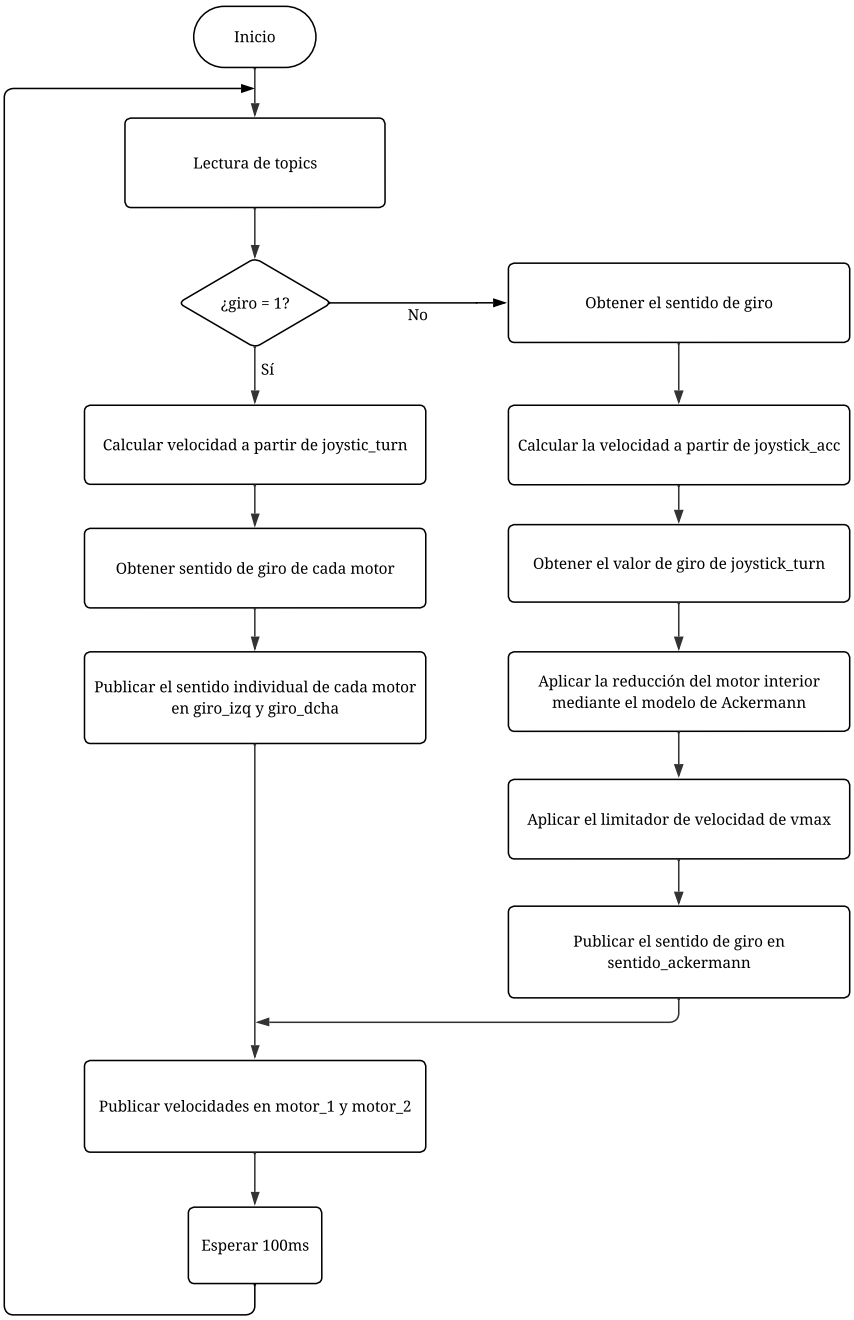


Figura 5.5: Diagrama de flujo del nodo `ackermann.c`. Fuente: Elaboración propia

A continuación, se explica el funcionamiento de cada una de las funciones del nodo descrito en el diagrama, comenzando por la creación de un temporizador ROS con periodo de 100 ms para ejecutar la lógica de control de forma periódica:

```
1  rcl_timer_t timer;
2  rclc_timer_init_default( & timer, & support, RCL_MS_TO_NS(100),
       timer_callback);
3  rclc_executor_add_timer( & executor, & timer);
4
5  // Bucle principal del nodo
6  while (true) {
7    rclc_executor_spin_some( & executor, RCL_MS_TO_NS(100));
8  }
```

Código 5.18: Timer del nodo

Según los datos recibidos, calcula y publica:

- Las velocidades individuales de los dos motores traseros en los topics `motor_1` (izquierdo) y `motor_2` (derecho).
- El sentido de desplazamiento (solo en modo Ackermann) en el topic `sentido_ackermann`.
- El sentido de giro individual (solo en modo giro sobre sí mismo) en los topics `giro_izq` y `giro_dcha`.

En el propio código se definen los parámetros físicos del vehículo, como la distancia entre ejes, el ancho de vía y el ángulo máximo de giro de la rueda interior, para realizar los cálculos de velocidad diferencial. Estos datos son fácilmente editables en caso de que en un futuro se diseñe la nueva dirección y se corte el chasis para reducir su longitud, como está previsto.

```
1  #define ANCHOVIA_MM 1640.0 // Distancia entre ruedas izquierda y derecha (
       mm)
2  #define DISTANCIAEJES_MM 1830.0 // Distancia entre ejes delantero y
       trasero (mm)
3  #define GIRO_MAX_angulo_grad 30.0 // Angulo maximo de giro (grados)
```

Código 5.19: Definición de parámetros físicos del vehículo

A partir del valor del joystick de giro se estima el ángulo de la rueda delantera interior, que se traduce en radios de giro diferentes para cada rueda trasera, como ya sucedía en el código básico (`envio_can_ACK.c`) de apartados anteriores. Una función auxiliar convierte el valor del joystick de aceleración en una velocidad proporcional, este valor depende del sentido de la marcha y se calcula de la misma forma que ya se expresó en la gráfica de la Figura 5.2:

```
unsigned char mapeo_joystick_velocidad(unsigned char valor_joystick,
    unsigned char sentido_val) {
  if (sentido_val == 1) { // Adelante
    if (valor_joystick <= 0x7F) return 0x00;
    if (valor_joystick >= 0xFE) return 0xFD;
    return (unsigned char)(((valor_joystick - 0x7F) * 0xFD) / (0xFE - 0x7F
        ));
  } else { // Atras
    if (valor_joystick <= 0x00) return 0xFD;
    if (valor_joystick >= 0x7F) return 0x00;
    return (unsigned char)(0xFD - ((valor_joystick * 0xFD) / 0x7F));
  }
}
```

Código 5.20: Mapeo del valor del joystick a velocidad

Además, el valor de velocidad se ajusta en función de la posición del selector del mando asociado a `vmax`, mediante un factor de reducción (Tabla 5.2):

```
double escala_vmax(int vmax_val) {
  switch (vmax_val) {
  case 192: return 1.0;
  case 193: return 0.85;
  case 194: return 0.70;
  case 195: return 0.50;
  case 196: return 0.35;
  case 197: return 0.20;
  default: return 1.0;
  }
}
```

Código 5.21: Escalado de la velocidad máxima

Este nodo distingue entre dos modos de funcionamiento del vehículo:

- Modo Ackermann (`giro` = 0):

  En este modo se utiliza el desplazamiento del joystick de giro publicado en `joystick_turn`. A partir de dicho valor se calcula el radio de giro del vehículo y, utilizando las dimensiones físicas (distancia entre ejes y ancho de vía), se obtienen los radios de las trayectorias que siguen cada una de las ruedas traseras, tal y como sucedía en `envio_can_ACK.c`.

  Como cada rueda debe recorrer un arco de circunferencia distinto, sus velocidades deben adaptarse para evitar deslizamientos. La rueda exterior mantiene la velocidad

(mapeada) marcada por el joystick de aceleración, mientras que la interior se ajusta proporcionalmente al cociente entre radios.

Ambas velocidades se ven finalmente escaladas en función de la posición del selector de velocidad máxima del mando, lo que permite limitar la velocidad del vehículo en ciertas situaciones.

- Modo giro sobre sí mismo (`giro` = 1):

  Cuando se activa este modo, el vehículo debe girar sobre su eje central sin desplazamiento. Para ello, ambos motores traseros giran a la misma velocidad pero en sentidos opuestos: uno hacia adelante y el otro hacia atrás; este modo no es completamente funcional en el momento en el que se desarrolla este proyecto, pero lo será cuando el vehículo conste de un motor en cada rueda.

  La dirección del giro (hacia la izquierda o hacia la derecha) se deduce comparando el valor del joystick de giro con el valor central (`0x7F`). Si el valor es inferior, el giro es hacia la izquierda, si es superior hacia la derecha. Además, el propio joystick de giro es sobre el que se mapea la velocidad de los motores, no se usa el de aceleración en este modo y se sigue la misma función del Código 5.20.

  En este modo también se publica el sentido de giro individual de cada motor (en los topics `giro_izq` y `giro_dcha`), ya que es necesario que se envíen sus valores a las controladoras para que éstas ajusten el sentido (se manda un `0x07` a la que gira hacia adelante y un `0x0B` a la que va hacia atrás).

Actuación sobre el bus: `emisor_can.c`

Este nodo se suscribe a distintos topics en los que publican los nodos `receptor.c` y `ackermann.c`, y traduce las consignas deseadas en cada momento en tramas CAN que envía por la interfaz `can0`. Tras crear el socket y registrar un manejador de `SIGINT` para cierre ordenado, se configuran las suscripciones y el executor ROS.

Topic	Tipo	Descripción
`motor_1`	Int32	Velocidad calculada para el motor trasero izquierdo.
`motor_2`	Int32	Velocidad calculada para el motor trasero derecho.
`sentido_ackermann`	UInt8	Dirección de avance en modo Ackermann.
`giro`	UInt8	Modo de funcionamiento (Ackermann o giro sobre eje).
`giro_izq`	UInt8	Consigna de giro para el motor izquierdo en giro sobre eje.
`giro_dcha`	UInt8	Consigna de giro para el motor derecho en giro sobre eje.
`arranque`	UInt8	Orden de arranque (1) o parada (2) del sistema.

Tabla 5.4: Topics a los que se suscribe `emisor_can.c`. Fuente: Elaboración propia.

La función auxiliar `send_can_frame` se encarga de la construcción y el envío de la trama CAN, añadiendo un segundo byte (`0x05`) cuando se trata de velocidad (para cumplir con el diccionario del Anexo D):

```
void send_can_frame(uint16_t can_id, uint8_t data_byte, bool add_0x05) {
  struct can_frame frame;
  frame.can_id = can_id;
  if (add_0x05) {
    frame.can_dlc = 2;
    frame.data[0] = data_byte;
    frame.data[1] = 0x05;
  } else {
    frame.can_dlc = 1;
    frame.data[0] = data_byte;
  }
  // Envio del frame por el socket CAN y registro en el terminal
  int nbytes = write(socket_can, & frame, sizeof(struct can_frame));
  if (nbytes < 0) {
    perror("Error enviando CAN frame");
  } else {
    if (add_0x05)
      printf("Enviado CAN ID: %X, Data: %02X 05\n", can_id, data_byte);
    else
      printf("Enviado CAN ID: %X, Data: %02X\n", can_id, data_byte);
  }
}
```

Código 5.22: Función auxiliar `send_can_frame()`

El funcionamiento del nodo se basa en la función `process_can()`, que implementa la siguiente lógica:

- Modo giro sobre eje (`giro == 1`): Envía sentidos de giro opuestos (procedentes de `ackermann.c`) sólo si cambia cada consigna:

```
if (giro == 1) {
  if (giro_izq != last_giro_izq) {
    send_can_frame(0x201, giro_izq, false);
    last_giro_izq = giro_izq;
  }
  if (giro_dcha != last_giro_dcha) {
    send_can_frame(0x202, giro_dcha, false);
    last_giro_dcha = giro_dcha;
  }
}
```

Código 5.23: Giro sobre eje en `process_can()`

- Modo Ackermann (`giro == 0`): Envía el mismo sentido a ambas controladoras (0x201 y 0x202):

```
else {
  if (sentido_ackermann != last_sentido) {
    send_can_frame(0x201, sentido_ackermann, false);
    send_can_frame(0x202, sentido_ackermann, false);
    last_sentido = sentido_ackermann;
  }
}
```

Código 5.24: Ackermann en `process_can()`

- Velocidades de motores (`0x301`, `0x302` con byte extra `0x05`): Envía cada velocidad si cambia y actualiza su estado:

```
if (valor_motor_1 != ultimo_motor_1) {
  send_can_frame(0x301, (uint8_t)(valor_motor_1 & 0xFF), true);
  ultimo_motor_1 = valor_motor_1;
}
if (valor_motor_2 != ultimo_motor_2) {
  send_can_frame(0x302, (uint8_t)(valor_motor_2 & 0xFF), true);
  ultimo_motor_2 = valor_motor_2;
}
```

Código 5.25: Velocidades de motores en `process_can()`

- Arranque/parada: Al cambiar el valor de `estado_arranque`, primero se detienen los motores (envío de `0x00` a `201` y `202`). Después:
  1. Arranque (`estado_arranque == 1`):
     - Se envía la contraseña `2B005002DF4B0000` a `0x601` y `0x602` para desbloquear las controladoras.
     - Se envía `00 00` a `0x301` y `0x302` para inicializar las controladoras.
     - Finalmente, se envía el mensaje `2F00280000000000` a `0x601` y `0x602` para ponerlas en estado operacional.
  2. Parada (`estado_arranque == 2`):
     - Se envía el mensaje `2F00280001000000` a `0x601` y `0x602`, lo que pone las controladoras en estado preoperacional y abre los contactores.

```
if (estado_arranque != ultimo_estado_arranque) {
  ultimo_estado_arranque = estado_arranque;

  // Detener motores de direccion/giro
  send_can_frame(0x201, 0x00, false);
  send_can_frame(0x202, 0x00, false);

  // Arranque: contrasena + velocidad cero
```

```
  if (estado_arranque == 1) {
    // Contrasena (8 bytes)
    struct can_frame frame_pass = {
      frame_pass.can_dlc = 8,
      frame_pass.data = {0x2B,0x00,0x50,0x02,0xDF,0x4B,0x00,0x00}
    };
    frame_pass.can_id = 0x601; write(socket_can, &frame_pass, sizeof(
        frame_pass));
    frame_pass.can_id = 0x602; write(socket_can, &frame_pass, sizeof(
        frame_pass));

    // Velocidades cero (2 bytes)
    struct can_frame frame_zero = { .can_dlc=2, .data={0x00,0x00} };
    frame_zero.can_id = 0x301; write(socket_can, &frame_zero, sizeof(
        frame_zero));
    frame_zero.can_id = 0x302; write(socket_can, &frame_zero, sizeof(
        frame_zero));
  }

  // SDO cambio de estado (byte[4] = 0x00 arranque, 0x01 parada)
  struct can_frame frame = {
    frame.can_dlc = 8,
    frame.data = {0x2F,0x00,0x28,0x00,
                (estado_arranque == 1) ? 0x00 : 0x01,
                0x00,0x00,0x00}
  }
    frame.can_id = 0x601;
    write(socket_can, & frame, sizeof(frame));
    frame.can_id = 0x602;
    write(socket_can, & frame, sizeof(frame));
}
```

Código 5.26: Arranque/parada en `process_can()`

Cada bloque actualiza inmediatamente los últimos valores para evitar reenvíos redundantes. Además, al ejecutarse tras cada `rclc_executor_spin_some()` (cada 100 ms), se garantiza una transmisión periódica y reactiva.

Finalmente, como en los nodos anteriores, se instala un manejador `SIGINT` para interrumpir el bucle y, en el bloque de limpieza, se cierra el socket CAN y se liberan recursos:

```
void handle_sigint(int sig) {
  (void)sig;
  running = 0; // Sale del bucle principal
}
```

```
5
6 close(socket_can);
7 rcl_node_fini(&node);
8 rclc_support_fini(&support);
```

Código 5.27: Manejador de señal y liberación de recursos

Relación entre nodos mediante topics

La Figura 5.6 muestra de forma global cómo se interconectan los tres nodos del sistema a través de los topics ROS 2:

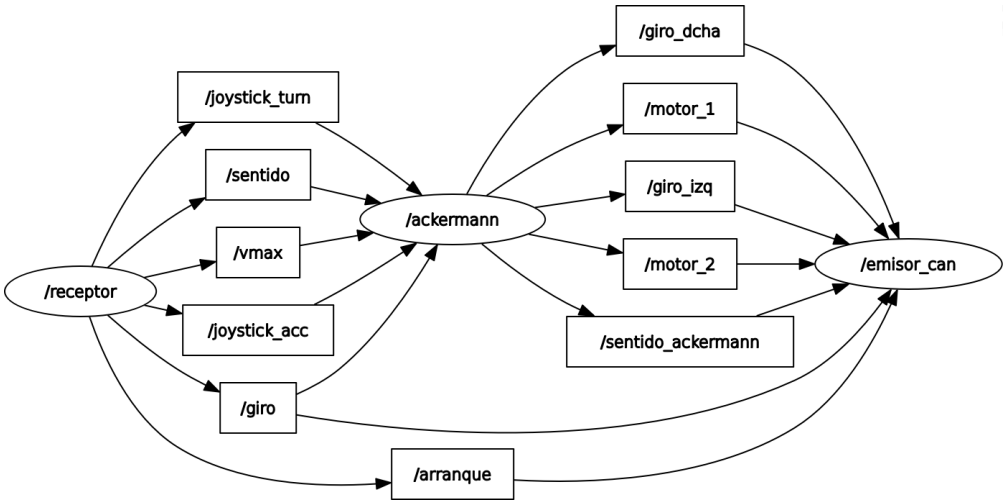


Figura 5.6: Diagrama generado con `rqt_graph`. Fuente: Elaboración propia

- El nodo `receptor` actúa como captador de datos: publica los valores de los joysticks (`joystick_acc`, `joystick_turn`), el sentido y modo de conducción (`sentido`, `giro`), el estado de arranque/parada (`arranque`) y el limitador de velocidad (`vmax`).
- El nodo `ackermann` se suscribe a estos topics, realiza el cálculo diferencial o de giro sobre sí mismo, y publica las consignas resultantes para los motores traseros: velocidades (`motor_1`, `motor_2`), sentido en modo Ackermann (`sentido_ackermann`) y sentidos de giro en modo rotación (`giro_izq`, `giro_dcha`).
- El nodo `emisor_can` recoge esas consignas (cinco topics de velocidad/sentido junto al topic `arranque`) y las traduce en tramas CAN enviadas por la interfaz `can0`.

Construcción mediante `CmakeLists.txt`

Para compilar, enlazar e instalar los tres nodos ROS 2 (`receptor.c`, `ackermann.c` y `emisor_can.c`), se utiliza el fichero `CMakeLists.txt` que se describe a continuación.

Configuración mínima de CMake y nombre del paquete:

```
cmake_minimum_required(VERSION 3.8)
project(tfg_jljg_completo)
```

Define la versión mínima de CMake (3.8) necesaria y establece el nombre del paquete ROS 2 como `tfg_jljg_completo`.

Búsqueda de dependencias:

```
find_package(ament_cmake REQUIRED)
find_package(rcl REQUIRED)
find_package(rclc REQUIRED)
find_package(std_msgs REQUIRED)
```

Localiza e incluye los módulos clave:

- `ament_cmake`: sistema de construcción de ROS 2.
- `rcl`, `rclc`: APIs de cliente para nodos en C/C++.
- `std_msgs`: tipos de mensaje estándar (UInt8, Int32, ...).

Inclusión de directorios de cabeceras:

```
include_directories(include)
```

Permite al compilador buscar archivos de cabecera (`.h`) dentro de la carpeta `include/` del proyecto.

Definición de ejecutables ROS 2:

```
add_executable(ackermann src/ackermann.c)
add_executable(receptor src/receptor.c)
add_executable(emisor_can src/emisor_can.c)
```

Declara cada nodo como un ejecutable, especificando su archivo fuente en `src/`.

Enlace con dependencias de ROS 2:

```
ament_target_dependencies(ackermann rclc rcl std_msgs)
ament_target_dependencies(receptor rclc rcl std_msgs)
ament_target_dependencies(emisor_can rcl rclc std_msgs)
```

Asocia los ejecutables con las librerías de ROS 2 y los tipos de mensaje necesarios para resolver símbolos y estructuras de datos.

Inclusión de la librería matemática (necesaria para cálculos):

```
target_link_libraries(ackermann m)
```

Incluye la librería al nodo `ackermann` para operaciones trigonométricas (funciones `sin`, `tan`, etc.).

Instalación de ejecutables:

```
install(TARGETS
  ackermann
  receptor
  emisor_can
  DESTINATION lib/${PROJECT_NAME})
```

Instala los ejecutables para que puedan ser lanzados tras compilar.

Instalación de archivos de lanzamiento:

```
install(DIRECTORY launch/
  DESTINATION share/${PROJECT_NAME}/launch)
```

Copia el directorio `launch/`, con el script `.launch.py`, dentro de `/tfg_jljg_completo/launch`.

Cierre de la definición del paquete:

```
ament_package()
```

Finaliza la configuración para el sistema de construcción `ament_cmake`.

Archivo de lanzamiento `sistema.launch.py`

Una vez compilados e instalados los nodos, se puede lanzar todo el sistema de forma conjunta mediante un archivo `launch` en Python. El fichero `sistema.launch.py`, situado en el directorio `launch/`, contiene lo siguiente:

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='tfg_jljg_completo',
            executable='receptor',
```

```
9             name='receptor'
10        ),
11        Node(
12            package='tfg_jljg_completo',
13            executable='ackermann',
14            name='ackermann'
15        ),
16        Node(
17            package='tfg_jljg_completo',
18            executable='emisor_can',
19            name='emisor_can'
20        ),
21    ])
```

Código 5.28: Archivo de lanzamiento `sistema.launch.py`

Gracias a este archivo, todo el sistema puede ponerse en funcionamiento con un solo comando:

```
ros2 launch tfg_jljg_completo sistema.launch.py
```

Este método permite automatizar el arranque del sistema; es posible configurar el sistema operativo del ordenador para que ejecute este comando al iniciarse de una forma similar a la descrita en el Anexo C. De este modo, al arrancar, los tres nodos se ejecutan sin necesidad de intervención manual.

5.4. Corrección de errores

Durante las pruebas del sistema, el receptor de radio muestra a través del display los errores 08.01 (*CAN passive*), 08.02 (*CAN I/O Buffer overflow*) y 08.06 (*CAN Transmit COB-ID Collision*). La descripción completa de estos códigos puede consultarse en el manual de códigos de error [39] incluido en el Anexo H.

Para resolver esta situación, se hacen los siguientes ajustes:

- Se modifica el nodo `emisor_can.c` para que únicamente envíe consignas cuando éstas cambien respecto al valor anterior. De esta forma se reduce la carga sobre el bus CAN. La versión final es la mostrada en el apartado anterior, no se incluye la versión que generaba estos errores.
- Se crimpan correctamente cuatro cables que estaban sueltos:
  - Los dos correspondientes a las líneas CAN (CAN_H y CAN_L) entre el PC y la controladora 1.

    - Los dos que conectan el receptor Scanreco G3B al bus CAN.

- Se ajusta la configuración del archivo `system.launch.py` para que, durante la ejecución normal, los tres terminales implicados no muestren mensajes por pantalla, lo cual ayuda a evitar errores relacionados con el buffer de salida.

CAPÍTULO 6

Pruebas y validación del sistema

En este capítulo se describen una serie de pruebas realizadas al sistema desarrollado en el capítulo anterior, llevadas a cabo en el taller del Área de Experimentación *LAENTIEC* [40] con el montaje de la Figura 6.1.

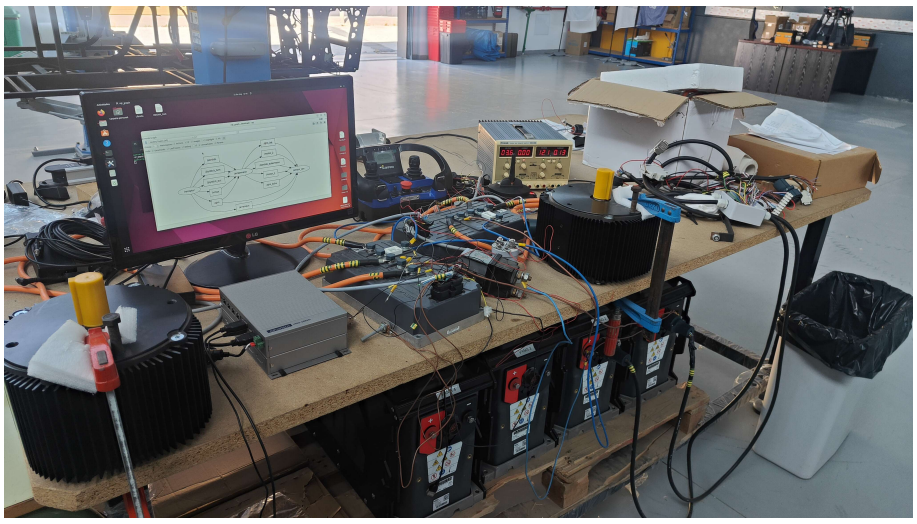


Figura 6.1: Montaje utilizado en el taller durante las pruebas. Fuente: Elaboración propia

La validación comienza con una comprobación de la velocidad de los motores sin ninguna limitación. A partir de esta referencia, se activa el limitador de velocidad al mínimo (20 %) con el objetivo de comprobar su funcionamiento. Una vez verificado, el resto de experimentos se realizan con los motores en este estado para reducir vibraciones y ruido.

Las pruebas evalúan la comunicación por CAN sin errores, el correcto cálculo de velocidades según el modelo descrito en la Subsección 5.3.2 y el funcionamiento de los distintos botones que permiten seleccionar los modos de funcionamiento.

Estructura de las capturas

Antes de comenzar con el análisis de los ensayos, se describe la estructura de las capturas de pantalla utilizadas. En cada una de ellas se muestran siete terminales organizados en dos filas; cuatro en la parte superior y tres en la inferior. De izquierda a derecha y de arriba a abajo, cada ventana muestra:

- `echo` del topic `joystick_acc`.
- `echo` del topic `joystick_turn`.
- `echo` del topic `vmax`, cuyos valores posibles se recogen en la Tabla 5.2.
- `echo` del topic `giro`.
- Ejecución del programa `receptor_can.c` [37], filtrando los mensajes CAN con ID `0x181`, correspondientes a la velocidad del motor izquierdo.
- Ejecución del mismo programa, filtrando los mensajes con identificador `0x182`, correspondientes a la velocidad del motor derecho.
- `echo` del topic `sentido`.

Esta disposición se mantiene en todas las capturas de pantalla de esta sección.

6.1. Velocidad máxima sin limitador

En la Figura 6.2 se solicita al sistema la velocidad máxima mediante el joystick de aceleración, sin activarse ningún limitador.

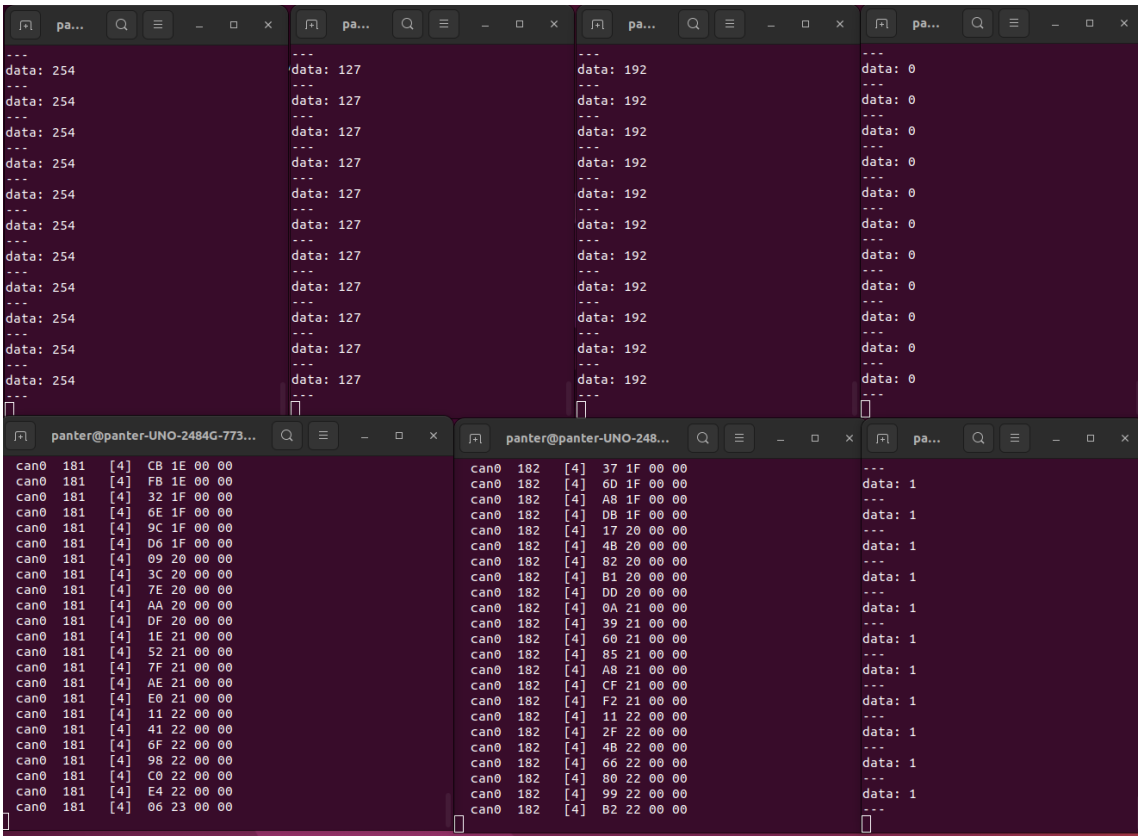


Figura 6.2: Velocidad máxima sin limitador. Fuente: Elaboración propia

Los valores de velocidad de cada motor se muestran en los ID `0x181` y `0x182`, que corresponden respectivamente a las velocidades de los motores izquierdo y derecho. Están transmitidos en formato hexadecimal y es importante tener en cuenta que el orden de los bytes está invertido con respecto al orden habitual de lectura (*little-endian*), por lo que, para obtener el valor en revoluciones por minuto (RPM), es necesario invertir los bytes antes de realizar la conversión a decimal:

- Motor izquierdo: `0x22C0` = 8 896 RPM
- Motor derecho: `0x2280` = 8 832 RPM

6.2. Velocidad máxima con limitador del 20 %

Se activa el limitador de velocidad al 20 % (`vmax = 197`) y se envía la misma orden de aceleración máxima desde el joystick; el resultado se muestra en la Figura 6.3:

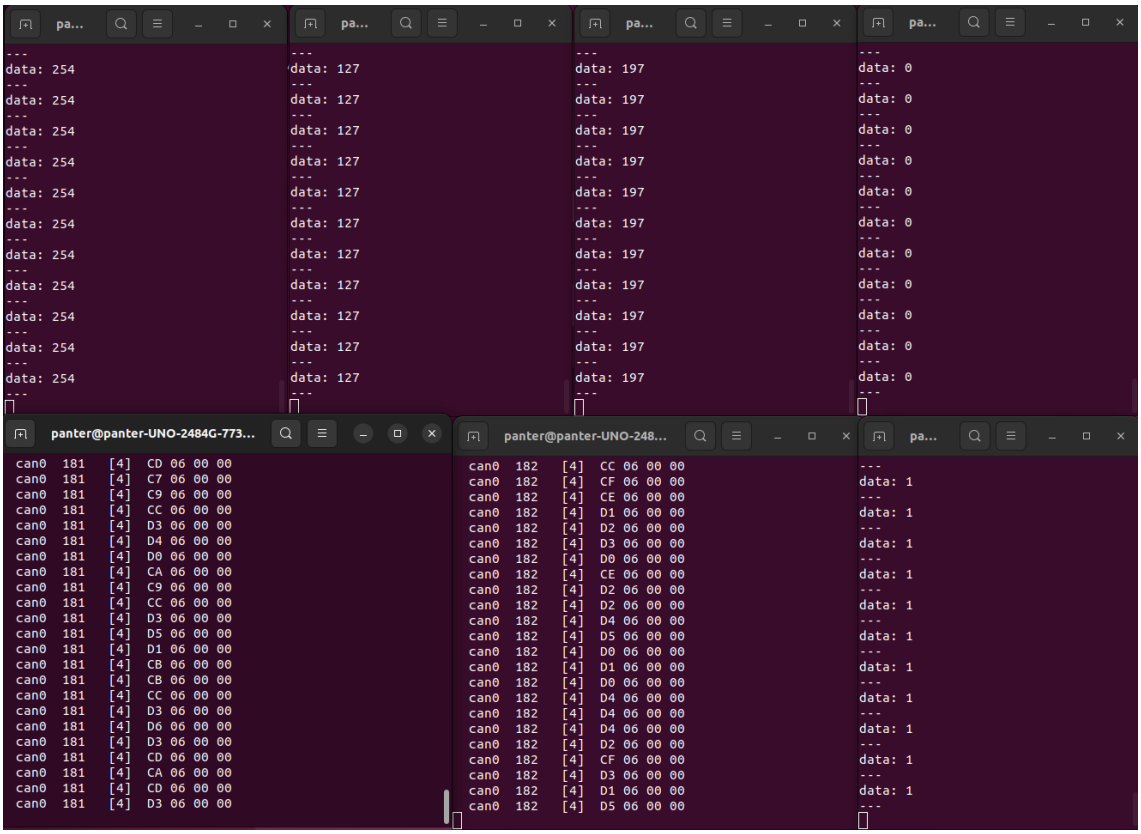


Figura 6.3: Velocidad máxima con limitador del 20 %. Fuente: Elaboración propia

De nuevo, se recogen los valores enviados por el bus CAN y se expresan en RPM:

- Motor izquierdo: `0x06D1` = 1 745 RPM
- Motor derecho: `0x06CD` = 1 741 RPM

Se comprueba si el limitador se aplica correctamente:

$$\text{Reducción}_{\text{izquierdo}} = \frac{\text{RPM}_{\text{limitado}}}{\text{RPM}_{\text{máximo}}} \cdot 100 = \frac{1\,745}{8\,896} \cdot 100 = 19{,}62\,\%$$

$$\text{Reducción}_{\text{derecho}} = \frac{\text{RPM}_{\text{limitado}}}{\text{RPM}_{\text{máximo}}} \cdot 100 = \frac{1\,741}{8\,832} \cdot 100 = 19{,}71\,\%$$

La reducción se sitúa en torno al 20 % en ambos casos, validando el funcionamiento de la limitación de velocidad, por lo que se usa en el resto de experimentos.

6.3. Marcha hacia adelante con giro Ackermann

En esta prueba se utiliza el modo de desplazamiento Ackermann (topic `giro = 0`) y se solicita un movimiento hacia adelante (topic `sentido = 1`) con un giro a la izquierda (topic `joystick_turn < 127`), utilizando una consigna de aceleración determinada por el joystick (`joystick_acc`). La Figura 6.4 muestra una captura del sistema durante este experimento.

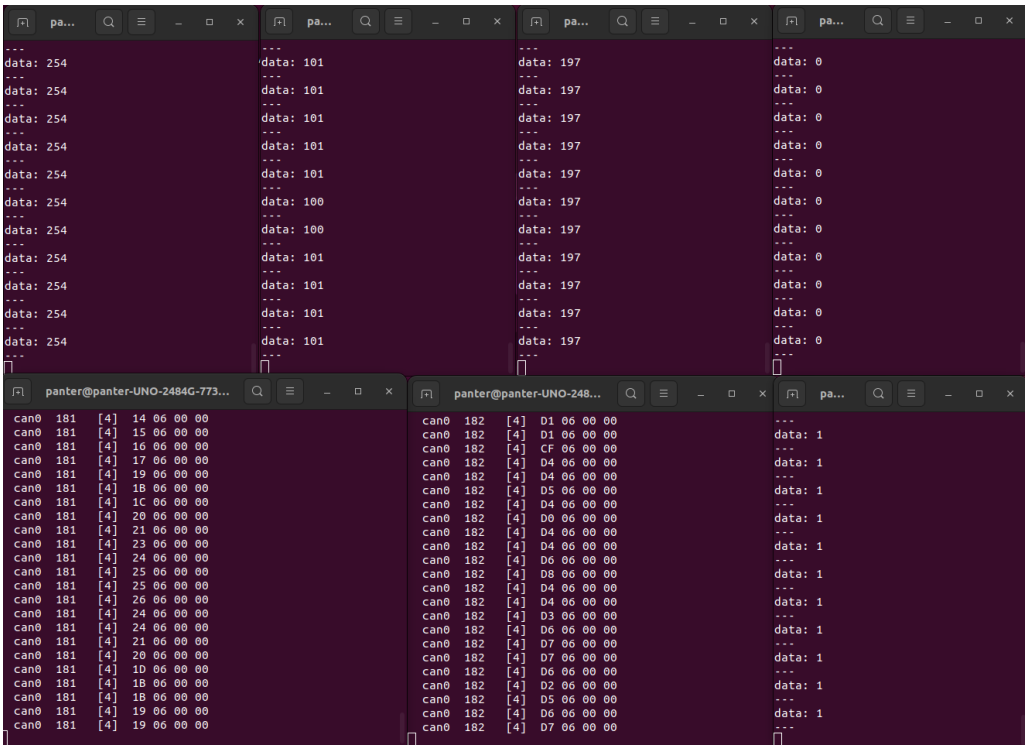


Figura 6.4: Marcha hacia adelante con giro a la izquierda. Fuente: Elaboración propia

El objetivo principal de esta prueba es validar que la lógica de cálculo de velocidades diferenciales para las ruedas traseras se ajusta correctamente al modelo de Ackermann. Para ello, se comparan los valores obtenidos desde el bus CAN con los valores teóricos que deberían resultar del modelo (Subsección 5.3.2).

A partir del valor leído en el topic `joystick_turn`, que en esta prueba es `100`, se calcula el ángulo de giro correspondiente. Dado que el ángulo máximo de giro de la rueda

delantera interior es de 30° (0,5236 rad) cuando `joystick_turn = max`, el valor leído se interpola linealmente entre 0 y 127 para obtener el ángulo θ:

$$\theta = \theta_{\max} \cdot \left(1 - \frac{\texttt{joystick\_turn}}{127}\right)$$

Sustituyendo:

$$\theta = 0{,}5236 \cdot \left(1 - \frac{100}{127}\right) = 0{,}1113 \text{ rad}$$

Con este ángulo, y usando la distancia entre ejes $L = 1\,830$ mm:

$$R = \frac{L}{\tan(\theta)} = \frac{1\,830}{\tan(0,1113)} = 16\,374{,}10 \text{ mm}$$

A partir de aquí se obtienen los radios de giro de las ruedas traseras:

$$R_{\text{int}} = R - \frac{1\,640}{2} = 15\,554{,}1 \text{ mm}, \quad R_{\text{ext}} = R + \frac{1\,640}{2} = 17\,194{,}1 \text{ mm}$$

En la figura se puede observar que las velocidades transmitidas por el bus CAN en los ID `0x181` y `0x182` son las siguientes:

- Motor izquierdo: `0x0626` = 1 574 RPM
- Motor derecho: `0x06D4` = 1 748 RPM

Finalmente, por definición del modelo, se sabe que:

$$v_{\text{int}} = v \cdot \frac{R_{\text{int}}}{R_{\text{ext}}}$$

Teniendo los radios calculados y la velocidad exterior, se comprueba si la interior coincide con la que se está mandando:

$$v_{\text{int}} = 1\,748 \cdot \frac{15\,554,1}{17\,194,1} = 1\,581 \approx 1\,574 \text{ RPM}$$

Se observa que la velocidad del motor izquierdo (interior) es inferior a la del derecho (exterior), lo cual es coherente con un giro hacia la izquierda. Además, la relación entre ambas velocidades se aproxima a la esperada según el modelo de Ackermann, lo que valida un cálculo de consignas correcto.

6.4. Marcha atrás con giro Ackermann

Esta prueba se realiza en modo Ackermann (topic `giro = 0`), con sentido de marcha hacia atrás (topic `sentido = 0`) y giro hacia la derecha (topic `joystick_turn > 127`). La Figura 6.5 muestra una imagen del sistema durante esta prueba.

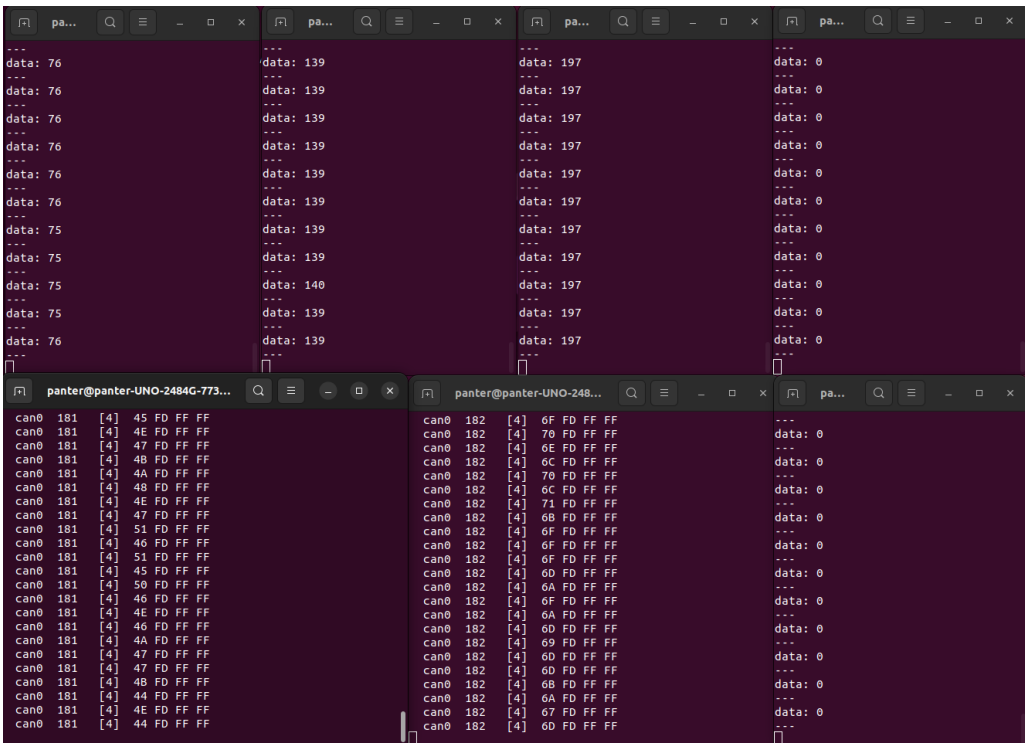


Figura 6.5: Marcha atrás con giro a la derecha. Fuente: Elaboración propia

El objetivo es validar que el cálculo de velocidades diferenciales sigue siendo coherente con el modelo de Ackermann incluso invirtiendo el sentido de marcha. El ángulo de giro se obtiene interpolando entre 127 (recto) y 255 (máximo giro a la derecha), tomando como referencia un ángulo máximo de 30° (0.5236 rad):

$$\theta = \theta_{\max} \cdot \frac{\texttt{joystick\_turn} - 127}{127}$$

$$\theta = 0{,}5236 \cdot \frac{139 - 127}{127} = 0{,}0495 \text{ rad}$$

A partir del ángulo, se calcula el radio de giro desde el eje trasero:

$$R = \frac{L}{\tan(\theta)} = \frac{1\,830}{\tan(0{,}0495)} = 36\,939{,}50 \text{ mm}$$

Y de ahí, los radios para las ruedas traseras:

$$R_{\text{int}} = R - \frac{1\,640}{2} = 36\,119{,}50 \text{ mm}, \quad R_{\text{ext}} = R + \frac{1\,640}{2} = 37\,759{,}50 \text{ mm}$$

Las velocidades recibidas desde el bus CAN para cada motor están codificadas en complemento a dos de 16 bits. Esto implica que, al estar en marcha atrás, los valores enviados por las controladoras son cercanos a `0xFFFFFFFF`, y su conversión da lugar a valores negativos.

Para facilitar la interpretación de los datos, se representarán como valores positivos tomando su valor absoluto. Esta conversión se realiza restando cada valor recibido a `0xFFFF` y sumando 1, es decir:

$$\text{RPM (positiva)} = \texttt{0xFFFF} - \text{valor\_hex} + 1$$

Aplicando esta transformación, se obtienen las siguientes velocidades:

- Motor izquierdo (exterior): `0xFD47` $\Rightarrow$ 0xFFFF − 0xFD47 + 1 = 0x02B9 = **697** RPM
- Motor derecho (interior): `0xFD69` $\Rightarrow$ 0xFFFF − 0xFD69 + 1 = 0x0297 = **663** RPM

Con los radios calculados y la velocidad exterior, se comprueba si la interior coincide con la que se está mandando de la misma forma que en el experimento anterior:

$$v_{\text{int}} = v \cdot \frac{R_{\text{int}}}{R_{\text{ext}}} = 697 \cdot \frac{36\,119{,}50}{37\,759{,}50} = 667 \text{ RPM} \approx 663 \text{ RPM}$$

Los resultados son coherentes con lo esperado, lo que valida los cálculos de asignación de velocidades diferenciales también en marcha atrás y para giros hacia la derecha.

6.5. Giro sobre sí mismo (derecha)

En esta prueba se activa el modo de giro sobre sí mismo (topic `giro = 1`), en el cual las ruedas giran en sentidos opuestos para permitir que el vehículo rote en su lugar. En este modo, las consignas de `joystick_acc` y `sentido` son ignoradas, de forma que el

movimiento del vehículo depende únicamente de la consigna de giro enviada desde el joystick.

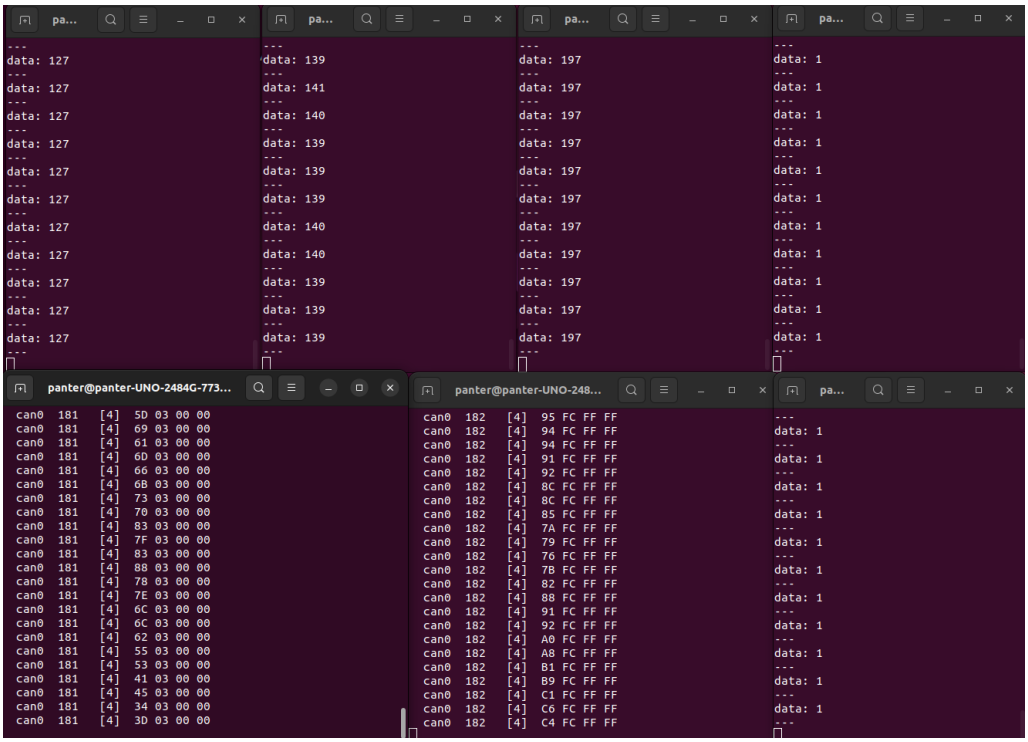


Figura 6.6: Giro sobre sí mismo hacia la derecha. Fuente: Elaboración propia

En esta prueba, se envía una consigna de giro hacia la derecha con un valor de `joystick_turn = 140`, superior al valor central `127`, lo que indica una rotación hacia la derecha sobre el propio eje vertical del vehículo.

Los valores de velocidad correspondientes al movimiento hacia atrás se representan como en el experimento anterior, y su valor absoluto decimal (RPM) se calcula como en dicho apartado:

$$\text{RPM (positiva)} = \texttt{0xFFFF} - \text{valor\_hex} + 1$$

Aplicando esta transformación, se obtienen las siguientes velocidades:

- Motor izquierdo: `0x0341` = **833** RPM (movimiento hacia delante)
- Motor derecho: `0xFCC1` $\Rightarrow$ 0xFFFF − 0xFCC1 + 1 = 0x033F = **831** RPM (movimiento hacia atrás)

Esto significa que el motor izquierdo gira hacia adelante, mientras que el motor derecho lo hace hacia atrás; este comportamiento genera un giro del vehículo sobre sí mismo hacia la derecha, tal como se esperaba.

Como las velocidades son simétricas, se valida el funcionamiento del modo de giro sobre el eje.

6.6. Giro sobre sí mismo (izquierda)

Este experimento también se realiza en modo de giro sobre sí mismo (topic `giro = 1`), pero en este caso con una consigna de giro hacia la izquierda, indicada por un valor de `joystick_turn = 101`, inferior al valor central de `127`. Como en el caso anterior, las señales de aceleración y sentido se ignoran.

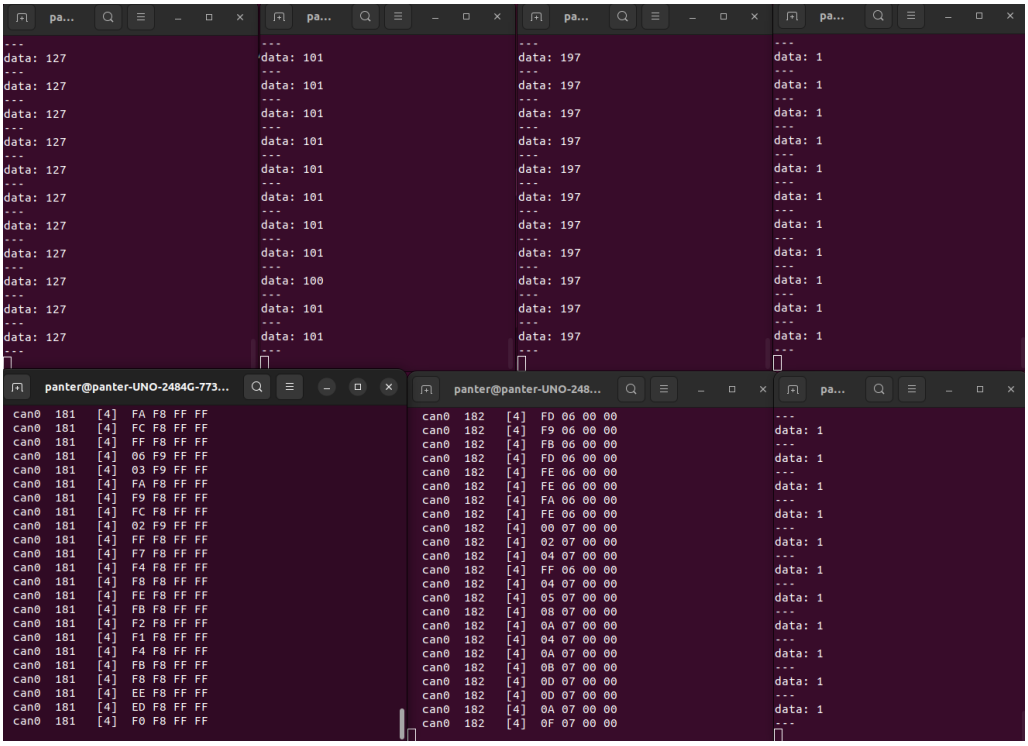


Figura 6.7: Giro sobre sí mismo hacia la izquierda. Fuente: Elaboración propia

Las velocidades de cada motor recibidas por el bus CAN son las siguientes:

- Motor izquierdo: `0xF8EE`
- Motor derecho: `0x070D`

Para facilitar su interpretación, se transforman las velocidades negativas a valores positivos mediante la operación:

$$\text{RPM (positiva)} = \texttt{0xFFFF} - \text{valor\_hex} + 1$$

Aplicando esta conversión:

- Motor izquierdo: $\texttt{0xF8EE} \Rightarrow 0x\text{FFFF} - 0\text{xF8EE} + 1 = \mathbf{1\,810}$ RPM (movimiento hacia atrás)
- Motor derecho: $\texttt{0x070D} = \mathbf{1\,805}$ RPM (movimiento hacia delante)

Tal y como ocurre en el experimento anterior, las velocidades son simétricas y de sentido opuesto.

6.7. Prueba de arranque y parada

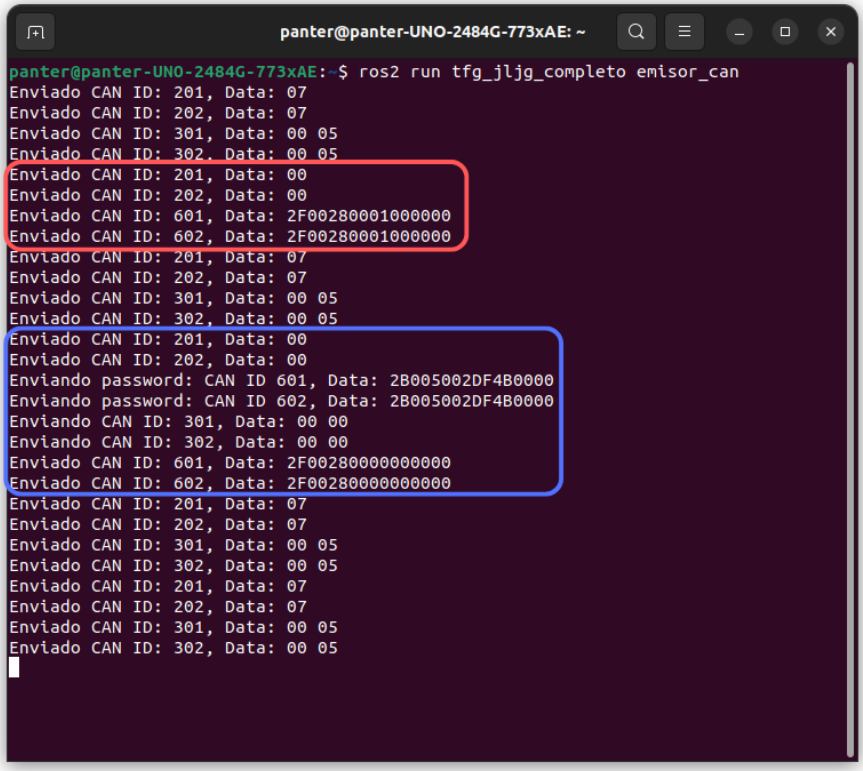


Figura 6.8: Mensajes CAN en arranque y parada del sistema. Fuente: Elaboración propia

En esta prueba se verifica el comportamiento del sistema al solicitar tanto la parada como el arranque del vehículo. La Figura 6.8 corresponde a una captura del nodo encargado de enviar los mensajes durante este experimento (`emisor_can.c`), donde se observan claramente los mensajes enviados por el bus CAN en cada fase. La zona enmarcada en rojo corresponde a la parada, y la zona azul al arranque.

Parada del sistema

Durante la parada, se envían los siguientes mensajes:

- A los ID `201` y `202` se envía un `00`.
- A los ID `601` y `602` se envía `2F00280001000000` que, como se detalla en el Anexo D, corresponde a la escritura del mensaje que pone las controladoras en estado **preoperacional**. Esto provoca la apertura de los contactores, interrumpiendo la alimentación.

Arranque del sistema

En el arranque, los mensajes enviados son los siguientes:

- A los ID `201` y `202` se envía un `00`.
- A los ID `601` y `602` se envía el mensaje `2B005002DF4B0000`: contraseña necesaria para desbloquear las controladoras.
- A los ID `301` y `302` se envía `0000`, que permite inicializar las controladoras cuando se mande la orden de paso a operacional.
- De nuevo, a los ID `601` y `602` se envía `2F00280000000000`, que cambia el estado de las controladoras a **operacional**, lo que provoca el cierre de los contactores y habilita la tracción.

Este intercambio de mensajes garantiza que el paso entre estados (parada y arranque) se realiza de forma segura, respetando la lógica interna de las controladoras.

CAPÍTULO 7

Resultados del trabajo

En este capítulo se hace una valoración global de los resultados obtenidos tras el desarrollo del sistema de control del vehículo. A partir de los ensayos presentados en el capítulo anterior, se analiza el cumplimiento de los objetivos planteados, así como la validez de las decisiones de diseño adoptadas.

Valoración final

El sistema ha demostrado ser funcional y coherente con los modelos definidos. En particular, se han alcanzado los siguientes logros:

- Integración efectiva del ordenador industrial Advantech UNO-2484G con el sistema operativo Ubuntu 22.04, el módulo CAN PCM-26D2CA-BE y ROS 2.
- Implementación de una arquitectura modular de nodos ROS 2 que permite la lectura de datos del joystick, el cálculo de consignas de velocidad según el modelo de Ackermann y el envío de mensajes por el bus.
- Interpretación y transformación correcta de los valores transmitidos por el sistema de radiocontrol Scanreco, permitiendo el control remoto del vehículo en tiempo real.
- Validación experimental de los distintos modos de funcionamiento del sistema: avance y retroceso, giros con diferencial de velocidades y giros sobre el eje vertical del vehículo.
- Gestión efectiva de los estados de las controladoras, incluyendo el arranque, parada y transición entre modos `operacional` y `pre-operacional` mediante mensajes CANopen.
- Coherencia en las tramas CAN emitidas y recibidas, confirmando la correcta comunicación entre los diferentes componentes del sistema.

El sistema alcanzado sirve de base sólida para futuras ampliaciones, algunas de las cuales se citan en el Capítulo 8. La integración de los distintos elementos hardware y software ha resultado satisfactoria y el sistema presenta un comportamiento correcto que concuerda con el apartado teórico.

CAPÍTULO 8

Ideas de trabajo futuro

A continuación, se presentan una serie de posibles trabajos que podrían desarrollarse en el futuro, con el objetivo de aumentar las prestaciones del Panter. Con estas propuestas no solo se pretende mejorar las capacidades actuales del sistema, también abrir nuevas vías para su evolución, adaptándose a futuras necesidades:

- **Implementación de un display externo** que muestre información importante en tiempo real, como la velocidad individual de cada rueda, la dirección de movimiento, el modo de giro, el nivel de carga de la batería o la detección de posibles fallos. Esta mejora haría que fuese completamente portátil sin renunciar a prestaciones ya que, actualmente, para poder ver esta información es necesario conectar un monitor al PC y mostrarla en terminales.
- **Inclusión de las baterías en el bus CAN**, la integración completa del BMS (Battery Management System) en el bus CAN permitiría monitorizar parámetros como el estado de carga (SOC), la tensión por celda, la temperatura o el estado de salud. Esto permitiría un control preciso de la parte eléctrica, mejorando la eficiencia
- **Seguimiento de trayectorias**: mediante un sistema de navegación por GPS, Galileo o similar, se propone que el Panter sea capaz de seguir rutas predefinidas.
- **Sistema navegación autónoma**: otra posible línea de trabajo, relacionada con la anterior, consiste en integrar sensores (ultrasonidos, cámaras, LiDAR) que permitan detectar obstáculos. Esta mejora sería especialmente útil de cara a la navegación autónoma uniéndose al seguimiento de trayectorias ya que se podría adaptar el recorrido para superar obstáculos.

Bibliografía

[1] R. Onieva Molina, *Configuración e implantacion de controladoras en un vehículo eléctrico*, Dept. Ingeniería de Sis. y Aut., Trabajo de fin de grado, Universidad de Málaga, Málaga, 2024.

[2] D. Chrzanowski, *Diseño del cableado de un vehículo de rescate*, Dept. Ingeniería de Sis. y Aut., Trabajo de fin de grado, Universidad de Málaga, Málaga, 2024.

[3] Movelco, *Panter 4x4: Manual de usuario*, Sin fecha, España: Movelco.

[4] U. de Málaga. «Las XV Jornadas sobre seguridad y emergencias celebran un ejercicio práctico de simulación de rescate.» Recuperado el 24 de noviembre de 2024. (2023), dirección: https://www.uma.es/sala-de-prensa/noticias/--las-xv-jornadas-sobre-seguridad-y-emergencias-celebran-un-ejercicio-practico-de-simulacion-de-rescate/?set_language=es.

[5] M. Magnets. «IPM vs SPM Electric Motors.» Accedido: 25 de marzo de 2025. (2025), dirección: https://mpcomagnetics.com/blog/ipm-vs-spm-electric-motors/.

[6] S. Macenski, T. Foote, B. Gerkey et al., «Robot Operating System 2: Design, architecture, and uses in the wild,» *Science Robotics*, vol. 7, n.º 66, eabm6074, 2022. DOI: 10.1126/scirobotics.abm6074. dirección: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[7] A. Koubaa, ed., *Robot Operating System (ROS): The Complete Reference (Volume 5)* (Studies in Computational Intelligence). Springer, 2020, vol. 895, ISBN: 978-3-030-45955-0. DOI: 10.1007/978-3-030-45956-7. dirección: https://doi.org/10.1007/978-3-030-45956-7.

[8] S. Murtaza. «ROS2 and CARLA Setup Guide.» Accedido: 14 de abril de 2025, LearnOpenCV. (nov. de 2022), dirección: https://learnopencv.com/ros2-and-carla-setup-guide/.

[9] M. D. Natale, H. Zeng, P. Giusto y A. Ghosal, *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*. Springer, 2012, ISBN: 978-1-4614-0313-5. DOI: 10.1007/978-1-4614-0314-2. dirección: https://doi.org/10.1007/978-1-4614-0314-2.

[10] F. B. D. «¿Qué es el CAN Bus?» Accedido: 14 de abril de 2025. (2025), dirección: https://usuaris.tinet.cat/fbd/comunicaciones/canbus/can.html.

[11] E. JRW. «CAN Bus differential signal.» Imagen propia del autor, licenciada bajo CC BY-SA 4.0. Accedido: 14 de abril de 2025. (2025), dirección: https://commons.wikimedia.org/w/index.php?curid=55237229.

[12] Kit-Elec-Shop. «Interface IXXAT USB to CAN V1 Compact (Obsolète).» Accedido: 14 de abril de 2025. (2025), dirección: https://www.kit-elec-shop.com/fr/accessoires-variateurs/2334-interface-ixxat-usb-to-can-v1-compact-obsolete.html.

[13] Asociación Española de Normalización (AENOR), *UNE-EN 50325-4: Industrial communications subsystem based on ISO 11898 (CAN) for controller-device interfaces. Part 4: CANopen*, Norma técnica, Acceso mediante licencia institucional de la Universidad de Málaga, 2021.

[14] CAN in Automation (CiA), *CAN in Automation – The international users' and manufacturers' group for CAN*, https://www.can-cia.org/, Accedido en mayo de 2025, 2024.

[15] Scanreco. «Scanreco: Mandos a distancia inalámbricos profesionales.» Accedido el 15 de abril de 2025. (2025), dirección: https://scanreco.com/es/.

[16] Play y D. S.L., *PDM18 PMAC Motor*, Acceso privado. Accedido: 26 de noviembre de 2024. dirección: https://drive.google.com/file/d/1buztceZMFc377j6UTmEK0grcEG4HvSpM/view?usp=drive_link.

[17] Motenergy. «MEPMBRMO – Brushless Motor.» Accedido: 2024-11-26. (2024), dirección: http://www.motenergy.com/mepmbrmo.html.

[18] Play y D. S.L., *PDM18 curves*, Acceso privado. Accedido: 26 de marzo de 2025. dirección: https://drive.google.com/file/d/1AedA1pdlJI03pfJ2iUgMiNsYX6_TPKoy/view?usp=drive_link.

[19] S. P. EQUIPMENT. «Vanguard 48V 5kWh Lithium Ion Battery.» Accedido: 2024-12-05. (2024), dirección: https://scag.com/battery/vanguard-48v-5kwh-lithium-ion-battery/.

[20] Vanguard, *Fi Series – Fixed Battery Packs*, Acceso privado. Accedido: 26 de noviembre de 2024. dirección: https://drive.google.com/file/d/1tHE10MBExJN-xgg8ZmA516a6EOvHEWeU/view?usp=drive_link.

[21] Sevtronic. «Sevcon 634A85203 Gen4 72-80V 550A.» Accedido: 26 de noviembre de 2024. (), dirección: https://sevtronic.com/variadores-gen4/1222-sevcon634a85203-gen4-7280v-550a.html.

[22] A. International. «Albright SW200.» Accedido: 30 de marzo de 2025. (2025), dirección: https://www.albrightinternational.com/products/sw200/.

[23] Advantech. «UNO-2484G-9S51 Compact Industrial PC.» Accedido: 12 de abril de 2025. (2024), dirección: https://buy.advantech.eu/Compact-Tower-Systems/Embedded-Box-Computers-Automation-Industrial-Computers-UNO/UNO-2484G-9S51/system-21637.htm.

[24] L. Advantech Co., *UNO-2484G Datasheet*, Accedido: 13 de mayo de 2025, 2023. dirección: https://advdownload.blob.core.windows.net/productfile/PIS/UNO-2484G/file/UNO-2484G_DS(060523)20230605143131.pdf.

[25] Advantech Co., Ltd. «Advantech - Soluciones Industriales.» Imagen obtenida desde la web oficial. Accedido: 16 de abril de 2025. (2025), dirección: https://www.advantech.com/es-es/.

[26] L. Advantech Co., *PCM-26D2CA Datasheet*, Accedido: 13 de mayo de 2025, 2025. dirección: https://advdownload.advantech.com/productfile/PIS/PCM-26D2CA/file/PCM-26D2CA_DS(022625)20250226100430.pdf.

[27] Canonical Ltd. «Ubuntu 22.04 LTS (Jammy Jellyfish) – Release Downloads.» Accedido el 24 de abril de 2025. (2022), dirección: https://releases.ubuntu.com/jammy/.

[28] Western Digital. «Western Digital: SSD SATA.» Accedido en abril de 2025. (), dirección: %5Curl%7Bhttps://www.westerndigital.com/es-la%7D.

[29] ifm electronic gmbh. «ifm - Automatización eficiente.» Accedido: abril 2025. (2025), dirección: https://www.ifm.com/es/es.

[30] WinSCSI, *Manual de WinSCSI*, Acceso privado. Accedido: 26 de abril de 2025. dirección: https://drive.google.com/file/d/1YLoU0VnOf4671t23JpbTlNQYQ-q6yruW/view?usp=drive_link.

[31] Sevcon, *Gen4 Motor Controller Product Manual V3.4*, Accedido: 26 de noviembre de 2024. dirección: https://www.thunderstruck-ev.com/images/Gen4%20Product%20Manual%20V3%204.pdf.

[32] Advantech Co., Ltd., *PCM-26D2CA User Manual*, Accedido: 2025-05-15, 2020. dirección: https://www.elmark.com.pl/uploaded/karty_produktow/advantech/pcm-26d2ca/pcm-26d2ca_instrukcja-uzytkownika.pdf.

[33] «Arquitectura del bus CAN.» Acceso privado. Accedido el 20 de mayo de 2025. (2025), dirección: %5Curl%7Bhttps://drive.google.com/file/d/16DdH29fSoUTZgXnuie5aXXPMCDGNDe8v%7D.

[34] Scanreco, *Configuración de Scanreco*, Acceso privado. Accedido: 26 de abril de 2025. dirección: https://drive.google.com/file/d/1XOF7JnivZ3gC8tOfp_gkheUqL2UDo-NU/view?usp=drive_link.

[35] J. L. Jaramillo. «Configuración del Scanreco G3B.» Repositorio GitHub. (2025), dirección: https://github.com/jljaramillog/scanrecoG3B_config.

[36] «Software DVT para configuración de controladoras Sevcon.» Enlace privado de Google Drive. (2025), dirección: https://drive.google.com/file/d/1y1NbUqEYXG7CcQ0wS9zqTQsFEvOI4tQ2/view?usp=drive_link.

[37] J. L. Jaramillo. «Workspace del proyecto.» Repositorio GitHub. (2025), dirección: https://github.com/jljaramillog/tfg_jljg_ws.

[38] Scanreco AB. «Manual del mando Scanreco Mini.» Acceso privado. Accedido el 13 de mayo de 2025. (2023), dirección: %5Curl%7Bhttps://drive.google.com/file/d/1tBm3nrMAwTAhb0ozIClqZrJ8hvWwtT1h%7D.

[39] Scanreco, *G2B-G3B Error Codes Manual*, Accedido: 2025-05-15, 2025. dirección: https://www.lkwsmejkal.cz/download/PRISLUSENSTVI/SCANRECO%20-%20dalkove%20ovladani/rc400%20g2b-g3%20error%20codes.pdf.

[40] Universidad de Málaga. «Área de Experimentación LAENTIEC.» Accedido el 21 de mayo de 2025. (2025), dirección: https://www.uma.es/robotics-and-mechatronics/cms/menu/robotica-y-mecatronica/area-de-experimentacion/.

[41] Akeo. «Rufus: The Reliable USB Formatting Utility.» Accedido el 24 de abril de 2025. (2024), dirección: https://rufus.ie/es/.

[42] Advantech. «Driver Download - PCM-26D2CA-BE CAN Module.» Accessed on 2025-04-25. (2025), dirección: https://www.advantech.com/en/support/details/driver?id=GF-GRSC (visitado 25-04-2025).

ANEXOS

APÉNDICE A

Códigos desarrollados

En este anexo se incluyen todos los códigos en C desarrollados para el proyecto separados en dos secciones, la de los códigos básicos y la de la versión final con todas las funciones descritas en la memoria. Para utilizar los paquetes completos con los correspondientes `CMakelists.txt` y el `launcher` en caso del paquete `tfg_jljg_completo`, se recomienda la descarga del workspace completo desde el repositorio del proyecto: `https://github.com/jljaramillog/tfg_jljg_ws/` [37].

A.1. Códigos básicos para interactuar con CAN

receptor_can.c

```
1  /**
2   * Programa para escuchar mensajes en el bus CAN usando 'candump', con
        opcion de multiples filtros por COB-ID.
3   *
4   * El usuario puede introducir hasta 5 COB-IDs separados por espacios (
        ejemplo: 080 701 181).
5   * Si no se introduce nada, se muestran todos los mensajes del bus CAN.
6   */
7
8  // ======================
9  // INCLUDES
10 // ======================
11
12 #include <stdio.h> // Entrada/salida estandar
13 #include <stdlib.h> // Funcion system()
14 #include <string.h> // Manipulacion de cadenas
15
16 // ======================
17 // CONSTANTES
18 // ======================
```

```
19
20 #define CAN_INTERFACE "can0" // Nombre de la interfaz CAN
21 #define MAX_FILTROS 5 // Numero maximo de COB-IDs que se pueden introducir
22 #define BUFFER_TAM 100 // Tamano del buffer de entrada
23
24 // ========================
25 // FUNCION PRINCIPAL
26 // ========================
27
28 int main() {
29     char entrada[BUFFER_TAM]; // Linea de entrada del usuario
30     char command[256]; // Comando final que se ejecutara con system()
31     char filtros[128] = ""; // Parte del comando con los filtros CAN
32
33     printf("Introduce hasta %d COB-IDs separados por espacio (ejemplo: 080
           701 181), o ENTER para sin filtro:\n", MAX_FILTROS);
34     fgets(entrada, sizeof(entrada), stdin); // Leer linea completa
35
36     // Eliminar salto de linea final
37     entrada[strcspn(entrada, "\n")] = 0;
38
39     // Comprobar si se ha introducido algun filtro
40     if (strlen(entrada) > 0) {
41         int count = 0;
42         char *token = strtok(entrada, " ");
43         while (token != NULL && count < MAX_FILTROS) {
44             char filtro_str[20];
45             snprintf(filtro_str, sizeof(filtro_str), "%s,%s:7FF ",
                   CAN_INTERFACE, token);
46             strcat(filtros, filtro_str);
47             token = strtok(NULL, " ");
48             count++;
49         }
50
51         // Construir comando final con filtros multiples
52         snprintf(command, sizeof(command), "candump %s", filtros);
53         printf("Aplicando filtros a los siguientes COB-IDs: %s\n", filtros)
               ;
54     } else {
55         // Sin filtro
56         snprintf(command, sizeof(command), "candump %s", CAN_INTERFACE);
57         printf("Mostrando todos los mensajes CAN.\n");
58     }
59
```

```
60      // Ejecutar el comando resultante
61      system(command);
62
63      return 0;
64  }
```

Código A.1: Nodo en C que lee del bus CAN y muestra en el terminal, permitiendo filtrado

envio_can_BASICO.c

```
1   /**
2    * Programa para enviar tramas CAN manualmente mediante consola.
3    *
4    * El usuario debe introducir los mensajes en formato: ID#DATA (por
       ejemplo: 123#11.22.33).
5    * Internamente, se construye y ejecuta el comando 'cansend' sobre la
       interfaz especificada.
6    * Es util para pruebas iniciales o envio manual de comandos.
7    */
8
9   // ======================
10  // INCLUDES
11  // ======================
12
13  #include <stdio.h> // Entrada/salida estandar
14  #include <stdlib.h> // Funciones como system()
15  #include <string.h> // Manipulacion de cadenas (strcmp, strcspn)
16
17  // ======================
18  // CONSTANTES
19  // ======================
20
21  #define CAN_INTERFACE "can0" // Interfaz CAN por defecto (puede
      modificarse segun configuracion)
22
23  // ========================
24  // FUNCION PRINCIPAL
25  // ========================
26
27  int main() {
28      char input[100]; // Buffer para entrada del usuario
29
30      printf("CAN Sender iniciado. Introduce un mensaje en formato: COB_ID#
          DATA\n");
```

```
31
32      while (1) {
33          // Solicitar entrada al usuario
34          printf("\nIntroduce un mensaje CAN (o escribe 'exit' para salir): "
                );
35          fgets(input, sizeof(input), stdin); // Leer linea de entrada
36
37          // Eliminar el salto de linea final (\n)
38          input[strcspn(input, "\n")] = 0;
39
40          // Comprobar si el usuario desea salir
41          if (strcmp(input, "exit") == 0) {
42              break;
43          }
44
45          // Construir el comando cansend completo
46          char command[150];
47          snprintf(command, sizeof(command), "cansend %s %s", CAN_INTERFACE,
                 input);
48
49          // Ejecutar el comando y mostrar resultado
50          int ret = system(command);
51          if (ret != 0) {
52              printf("Error\n"); // Error al ejecutar el comando
53          } else {
54              printf("Mensaje enviado: %s\n", input); // Confirmacion de
                    envio
55          }
56      }
57
58      return 0; // Fin del programa
59  }
```

Código A.2: Nodo en C que permite publicar en el bus

envio_can_ACK.c

```
1  /**
2   * Este programa en C recibe datos desde el bus CAN y calcula las
        velocidades de motores
3   * siguiendo una cinematica de tipo Ackermann. Interactua directamente con
        la interfaz CAN
4   * y envia comandos a dos controladoras de motor (301 y 302) en funcion
        del valor del joystick
```

```
5  * y del estado de avance/retroceso recibido.
6  *
7  * Entradas desde el bus CAN:
8  * - 0x186 (joystick): Contiene dos bytes relevantes para velocidad y giro
       .
9  * - 0x201 y 0x202 (estado): Byte unico que indica sentido de marcha para
       motor 1 y 2.
10 *
11 * Salidas al bus CAN:
12 * - 0x301 y 0x302: Comandos de velocidad (byte 0) y modo (byte 1 = 0x05).
13 */
14
15 // ======================
16 // INCLUDES
17 // ======================
18
19 #include <stdio.h> // Entrada/salida estandar
20 #include <stdlib.h> // Funciones generales (exit, etc.)
21 #include <string.h> // Manipulacion de cadenas
22 #include <unistd.h> // Funciones de POSIX (close)
23 #include <fcntl.h> // Control de archivos
24 #include <sys/ioctl.h> // Control de dispositivos
25 #include <linux/can.h> // Estructura CAN (can_frame)
26 #include <sys/socket.h> // Funciones de sockets
27 #include <sys/types.h> // Tipos de sockets
28 #include <arpa/inet.h> // Utilidades de red
29 #include <net/if.h> // Informacion de interfaz de red
30 #include <signal.h> // Manejo de senales (Ctrl+C)
31 #include <math.h> // Funciones matematicas
32
33 // ======================
34 // DEFINICIONES CAN
35 // ======================
36
37 #define INTERFAZ_CAN "can0" // Nombre de la interfaz CAN
38
39 // Identificadores CAN (COB-ID)
40 #define COBID_JOYSTICK 0x186 // Entrada del joystick
41 #define COBID_CONTROLADORA_1 0x301 // Motor izquierdo
42 #define COBID_CONTROLADORA_2 0x302 // Motor derecho
43 #define COBID_ESTADO_1 0x201 // Estado del motor 1
44 #define COBID_ESTADO_2 0x202 // Estado del motor 2
45
46 // ======================
```

```
47 // PARAMETROS FISICOS
48 // ======================
49
50 #define DISTANCIAEJES_MM 1830.0 // Distancia entre ejes (mm)
51 #define ANCHOVIA_MM 1640.0 // Distancia entre ruedas (mm)
52 #define GIRO_MAX_GRAD 30.0 // Angulo maximo de giro (grados)
53 #define GRAD_A_RAD (M_PI / 180.0) // Conversion de grados a radianes
54
55 // ======================
56 // VARIABLES GLOBALES
57 // ======================
58
59 unsigned char estado_301 = 0x07; // Estado actual del motor 1 (por defecto
       : avance)
60 unsigned char estado_302 = 0x07; // Estado actual del motor 2
61 unsigned char ult_vel_301 = 0xFF; // Ultima velocidad enviada al motor 1 (
       para evitar redundancia)
62 unsigned char ult_vel_302 = 0xFF; // Ultima velocidad enviada al motor 2
63 int s; // Descriptor del socket CAN
64
65 // ============================
66 // MANEJO DE SENALES
67 // ============================
68
69 /**
70  * Captura Ctrl+C (SIGINT) y cierra el socket antes de salir.
71  */
72 void cerrar_programa(int signo) {
73     printf("\nCerrando programa...\n");
74     if (s >= 0) close(s);
75     exit(0);
76 }
77
78 // ============================
79 // FUNCIONES DEL SOCKET
80 // ============================
81
82 /**
83  * Inicializa y enlaza el socket CAN.
84  * Devuelve el descriptor del socket o -1 en caso de error.
85  */
86 int open_can_socket() {
87     int sock = socket(PF_CAN, SOCK_RAW, CAN_RAW);
88     if (sock < 0) {
```

```
89          perror("Error al abrir el socket CAN");
90          return -1;
91      }
92
93      struct ifreq ifr;
94      strncpy(ifr.ifr_name, INTERFAZ_CAN, sizeof(ifr.ifr_name) - 1);
95      if (ioctl(sock, SIOCGIFINDEX, &ifr) < 0) {
96          perror("Error al obtener el indice de la interfaz CAN");
97          close(sock);
98          return -1;
99      }
100
101     struct sockaddr_can addr = { .can_family = AF_CAN, .can_ifindex = ifr.
            ifr_ifindex };
102     if (bind(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
103         perror("Error al enlazar el socket CAN");
104         close(sock);
105         return -1;
106     }
107
108     return sock;
109 }
110
111 // ============================
112 // FUNCIONES DE MAPEADO
113 // ============================
114
115 /**
116  * Convierte el valor del joystick (0-255) en velocidad (0x00-0xFD),
117  * teniendo en cuenta el sentido (adelante o atras).
118  */
119 unsigned char mapeo_joystick_velocidad(unsigned char valor_joystick,
        unsigned char estado) {
120     if (estado == 0x07) { // Adelante
121         if (valor_joystick <= 0x7F) return 0x00;
122         if (valor_joystick >= 0xFE) return 0xFD;
123         return (unsigned char)(((valor_joystick - 0x7F) * 0xFD) / (0xFE - 0
                x7F));
124     } else if (estado == 0x0B) { // Atras
125         if (valor_joystick <= 0x00) return 0xFD;
126         if (valor_joystick >= 0x7F) return 0x00;
127         return (unsigned char)(0xFD - ((valor_joystick * 0xFD) / 0x7F));
128     }
129     return 0x00; // Estado desconocido o sin movimiento
```

```
130 }
131
132 /**
133  * Envia un mensaje CAN si la velocidad ha cambiado respecto a la ultima
        enviada.
134  */
135 void send_can_message(int sock, unsigned int cobid, unsigned char
        velocidad, unsigned char *ult_velocidad) {
136     if (*ult_velocidad == velocidad) return; // No enviar si no hay cambio
137     *ult_velocidad = velocidad;
138
139     struct can_frame frame = {
140         .can_id = cobid,
141         .can_dlc = 2,
142         .data = { velocidad, 0x05 } // El segundo byte es fijo
143     };
144
145     if (write(sock, &frame, sizeof(frame)) == sizeof(frame)) {
146         printf("CAN -> 0x%X: [%02X %02X]\n", cobid, frame.data[0], frame.
              data[1]);
147     } else {
148         perror("Error al enviar mensaje CAN");
149     }
150 }
151
152 // ============================
153 // CALCULO DE VELOCIDADES
154 // ============================
155
156 /**
157  * Realiza el calculo tipo Ackermann segun los valores del joystick y
        envia las velocidades.
158  * joystick_vel: valor de aceleracion
159  * joystick_turn: valor de giro
160  */
161 void calcular_y_enviar_velocidades(unsigned char joystick_vel, unsigned
        char joystick_turn) {
162     // Mapeo del joystick a velocidad base
163     unsigned char base_speed_301 = mapeo_joystick_velocidad(joystick_vel,
          estado_301);
164     unsigned char base_speed_302 = mapeo_joystick_velocidad(joystick_vel,
          estado_302);
165     double vel_izq = (double)base_speed_301;
166     double vel_dcha = (double)base_speed_302;
```

```
167
168     // Calculo del angulo de direccion
169     double angulo_grad = 0.0;
170     if (joystick_turn < 0x7F)
171         angulo_grad = -GIRO_MAX_GRAD * ((0x7F - joystick_turn) / 127.0);
172     else if (joystick_turn > 0x7F)
173         angulo_grad = GIRO_MAX_GRAD * ((joystick_turn - 0x7F) / 127.0);
174
175     // Calculo del radio de giro (Ackermann)
176     double radio_giro = INFINITY;
177     if (angulo_grad != 0.0) {
178         double angulo_rad = angulo_grad * GRAD_A_RAD;
179         radio_giro = fabs(DISTANCIAEJES_MM / tan(angulo_rad));
180     }
181
182     // Ajuste de velocidades por curva
183     if (radio_giro != INFINITY) {
184         double Ri = radio_giro - (ANCHOVIA_MM / 2.0);
185         double Ro = radio_giro + (ANCHOVIA_MM / 2.0);
186         double ratio = Ri / Ro;
187
188         if (joystick_turn < 0x7F)
189             vel_izq *= ratio;
190         else if (joystick_turn > 0x7F)
191             vel_dcha *= ratio;
192     }
193
194     // Redondeo y envio
195     unsigned char final_301 = (unsigned char)(vel_izq + 0.5);
196     unsigned char final_302 = (unsigned char)(vel_dcha + 0.5);
197
198     send_can_message(s, COBID_CONTROLADORA_1, final_301, &ult_vel_301);
199     send_can_message(s, COBID_CONTROLADORA_2, final_302, &ult_vel_302);
200 }
201
202 // ============================
203 // FUNCION PRINCIPAL
204 // ============================
205
206 int main() {
207     signal(SIGINT, cerrar_programa); // Ctrl+C
208
209     s = open_can_socket();
210     if (s < 0) return 1;
```

```
211
212     struct can_frame frame;
213
214     while (1) {
215         int nbytes = read(s, &frame, sizeof(frame));
216         if (nbytes < 0) {
217             perror("Error al leer del bus CAN");
218             continue;
219         }
220
221         // Procesamiento de mensajes CAN entrantes
222         switch (frame.can_id) {
223             case COBID_ESTADO_1:
224                 estado_301 = frame.data[0];
225                 printf("Estado 301: 0x%02X\n", estado_301);
226                 break;
227
228             case COBID_ESTADO_2:
229                 estado_302 = frame.data[0];
230                 printf("Estado 302: 0x%02X\n", estado_302);
231                 break;
232
233             case COBID_JOYSTICK:
234                 printf("Joystick: Vel=0x%02X Giro=0x%02X\n", frame.data[1],
                        frame.data[3]);
235                 calcular_y_enviar_velocidades(frame.data[1], frame.data[3]);
236                 break;
237         }
238     }
239
240     return 0;
241 }
```

Código A.3: Nodo en C que lee del bus CAN y calcula y envia velocidades segun joysticks

A.2. Códigos finales del proyecto

receptor.c

```c
/*
 * Nodo que escucha el bus CAN por la interfaz can0 y publica en
 * topics los valores relevantes para el control del vehiculo
 * (joystick, vmax, sentido, arranque, etc.).
 *
 * Publica en los topics: joystick_acc, joystick_turn, vmax, sentido,
 * giro, arranque.
 * Escucha CAN IDs: 0x186 (joysticks), 0x286 (control).
 */

// ======================
// INCLUDES
// ======================

// Librerias estandar de C
#include <stdio.h> // Entrada/salida estandar (printf, perror...)
#include <stdlib.h> // Funciones utiles (malloc, free, exit...)
#include <string.h> // Manipulacion de cadenas (strcpy, memset...)
#include <unistd.h> // Funciones del sistema (read, write...)
#include <errno.h> // Manejo de errores mediante errno

// Librerias para la configuracion y uso de sockets CAN
#include <net/if.h> // Informacion sobre interfaces de red (can0, etc.)
#include <sys/ioctl.h> // Control de dispositivos (ioctl)
#include <sys/socket.h> // API para trabajar con sockets
#include <linux/can.h> // Estructuras para mensajes CAN
#include <linux/can/raw.h> // Tipo de socket CAN_RAW para acceso directo
#include <signal.h> // Para capturar SIGINT (Ctrl+C)

// Librerias de ROS 2 (RCL y RCLC)
#include "rcl/rcl.h" // API principal de ROS 2 en C
#include "rclc/rclc.h" // Extension client library para C

// Tipos de mensajes estandar de ROS 2
#include "std_msgs/msg/u_int8.h" // Mensaje entero sin signo de 8 bits


// ======================
// CONSTANTES
// ======================
```

```
41
42 // Identificadores CAN (COB-ID) de los dispositivos conectados
43 #define COB_ID_JOYSTICK 0x186 // Mensajes provenientes de los joysticks
44 #define COB_ID_CONTROL 0x286 // Mensajes de control (vmax, sentido, giro,
       arranque)
45
46 // ========================
47 // VARIABLES GLOBALES
48 // ========================
49
50 volatile sig_atomic_t running = 1; // Control de bucle principal (1 =
       seguir, 0 = salir)
51
52 // ========================
53 // MANEJADOR DE SEALES
54 // ========================
55
56 // Manejo de seal SIGINT (Ctrl+C) para terminar el bucle principal
57 void handle_sigint(int sig) {
58   (void) sig; // Ignorar el argumento
59   running = 0;
60 }
61
62 // ========================
63 // FUNCIoN PRINCIPAL
64 // ========================
65
66 int main(int argc, char * argv[]) {
67   // Capturar Ctrl+C para salir limpiamente
68   signal(SIGINT, handle_sigint);
69
70   // Estructuras basicas de ROS 2
71   rcl_allocator_t allocator = rcl_get_default_allocator();
72   rclc_support_t support;
73   rcl_node_t node;
74
75   // Inicializacion del soporte y nodo ROS 2
76   rclc_support_init( & support, argc, argv, & allocator);
77   rclc_node_init_default( & node, "receptor", "", & support);
78
79   // Declaracion de publishers para publicar en distintos topics
80   rcl_publisher_t pub_joystick_acc;
81   rcl_publisher_t pub_joystick_turn;
82   rcl_publisher_t pub_vmax;
```

```
83  rcl_publisher_t pub_sentido;
84  rcl_publisher_t pub_giro;
85  rcl_publisher_t pub_arranque;
86
87  // Crear publishers
88  rclc_publisher_init_default( & pub_arranque, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "arranque");
89  rclc_publisher_init_default( & pub_joystick_acc, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "joystick_acc");
90  rclc_publisher_init_default( & pub_joystick_turn, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "joystick_turn");
91  rclc_publisher_init_default( & pub_vmax, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "vmax");
92  rclc_publisher_init_default( & pub_sentido, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "sentido");
93  rclc_publisher_init_default( & pub_giro, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "giro");
94
95  // ========================
96  // CONFIGURAR SOCKET CAN
97  // ========================
98
99  int s;
100 struct sockaddr_can addr;
101 struct ifreq ifr;
102 struct can_frame frame; // Estructura que representa un mensaje CAN
103
104 // Crear socket CAN tipo RAW
105 s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
106 if (s < 0) {
107   perror("Error abriendo el socket CAN");
108   return 1;
109 }
110
111 // Asociar el socket a la interfaz can0
112 strcpy(ifr.ifr_name, "can0");
113 ioctl(s, SIOCGIFINDEX, & ifr); // Obtener indice de can0
114
115 addr.can_family = AF_CAN;
116 addr.can_ifindex = ifr.ifr_ifindex;
117
118 // Vincular socket con la interfaz can0
119 bind(s, (struct sockaddr * ) & addr, sizeof(addr));
120
```

```
121  printf("Leyendo mensajes CAN en can0...\n");
122
123  // ========================
124  // BUCLE PRINCIPAL
125  // ========================
126
127  while (running) {
128    // Leer una trama CAN del bus
129    int nbytes = read(s, & frame, sizeof(struct can_frame));
130    if (nbytes < 0) {
131      perror("Error leyendo del socket");
132      break;
133    }
134
135    // ========================
136    // JOYSTICK: ID 0x186
137    // ========================
138    if (frame.can_id == COB_ID_JOYSTICK) {
139      std_msgs__msg__UInt8 msg_acc;
140      std_msgs__msg__UInt8 msg_turn;
141
142      // Extraer aceleracion (byte 1) y giro (byte 3)
143      msg_acc.data = frame.data[1];
144      msg_turn.data = frame.data[3];
145
146      // Publicar valores en ROS
147      rcl_publish( & pub_joystick_acc, & msg_acc, NULL);
148      rcl_publish( & pub_joystick_turn, & msg_turn, NULL);
149
150      printf("Publicado acelerador=%d, giro=%d\n", msg_acc.data, msg_turn.
           data);
151    }
152
153    // ========================
154    // CONTROL: ID 0x286
155    // ========================
156    if (frame.can_id == COB_ID_CONTROL) {
157      std_msgs__msg__UInt8 msg_vmax;
158      std_msgs__msg__UInt8 msg_sentido;
159      std_msgs__msg__UInt8 msg_giro;
160      std_msgs__msg__UInt8 msg_arranque;
161
162      // Byte 5: vmax
163      msg_vmax.data = frame.data[5];
```

```
164
165       // Byte 0: byte de control con varios botones, cada uno en un bit
166       uint8_t control_byte = frame.data[0];
167
168       // Bit 5: sentido (adelante/atras)
169       msg_sentido.data = (control_byte >> 5) & 0x01;
170
171       // Bit 0: modo giro (0 = Ackermann, 1 = giro sobre eje)
172       msg_giro.data = control_byte & 0x01;
173
174       // Publicar vmax, sentido y giro
175       rcl_publish( & pub_vmax, & msg_vmax, NULL);
176       rcl_publish( & pub_sentido, & msg_sentido, NULL);
177       rcl_publish( & pub_giro, & msg_giro, NULL);
178
179       printf("Publicado vmax=%d, sentido=%d, giro=%d\n", msg_vmax.data,
              msg_sentido.data, msg_giro.data);
180
181       // Bits 1 y 2: control de arranque/parada
182       if ((control_byte >> 1) & 0x01) {
183         msg_arranque.data = 1; // Start
184         rcl_publish( & pub_arranque, & msg_arranque, NULL);
185         printf("Publicado arranque: START\n");
186       } else if ((control_byte >> 2) & 0x01) {
187         msg_arranque.data = 2; // Stop
188         rcl_publish( & pub_arranque, & msg_arranque, NULL);
189         printf("Publicado arranque: STOP\n");
190       }
191     }
192   }
193
194   // =======================
195   // FINALIZACION
196   // =======================
197
198   // Cerrar socket CAN
199   close(s);
200
201   // Finalizar nodo ROS
202   rcl_node_fini( & node);
203
204   return 0;
205 }
```

Código A.4: Nodo en C que lee del bus CAN y publica en distintos topics

ackermann.c

```
1  /**
2   * Este nodo se suscribe a los topics joystick_acc, joystick_turn, vmax,
3   * sentido y giro.
4   *
5   * Publica comandos de velocidad (motor_1, motor_2) y de sentido
6   * individual para el modo giro (giro_izq, giro_dcha), o un sentido
7   * comun para Ackermann (sentido_ackermann).
8   */
9
10 // =======================
11 // INCLUDES
12 // =======================
13
14 #include <stdio.h> // Entrada/salida estandar
15 #include <math.h> // Funciones matematicas
16
17 // ROS 2 en C
18 #include "rcl/rcl.h" // API principal de ROS 2 en C
19 #include "rclc/rclc.h" // Extension client library para C
20 #include "rclc/executor.h" // Ejecutores para callbacks de ROS 2
21
22 // Tipos de mensajes estandar
23 #include "std_msgs/msg/int32.h" // Mensaje entero con signo de 32 bits
24 #include "std_msgs/msg/u_int8.h" // Mensaje entero sin signo de 8 bits
25
26
27 // =======================
28 // CONSTANTES
29 // =======================
30
31 // Parametros fisicos del vehiculo
32 #define ANCHOVIA_MM 1640.0 // Distancia entre ruedas izquierda y derecha (
       mm)
33 #define DISTANCIAEJES_MM 1830.0 // Distancia entre ejes delantero y
       trasero (mm)
34 #define GIRO_MAX_angulo_grad 30.0 // angulo maximo del volante (grados)
35
36 // =======================
37 // VARIABLES ROS
38 // =======================
39
40 // Suscripciones a topics
41 rcl_subscription_t sub_joystick_acc;
```

```
42 rcl_subscription_t sub_joystick_turn;
43 rcl_subscription_t sub_vmax;
44 rcl_subscription_t sub_sentido;
45 rcl_subscription_t sub_giro;
46
47 // Declaracion de publishers para publicar en distintos topics
48 rcl_publisher_t pub_motor_1;
49 rcl_publisher_t pub_motor_2;
50 rcl_publisher_t pub_sentido_ackermann;
51 rcl_publisher_t pub_giro_izq;
52 rcl_publisher_t pub_giro_dcha;
53
54 // Variables globales para los datos recibidos
55 int joystick_acc = 0;
56 int joystick_turn = 0;
57 int valor_vmax = 192;
58 int sentido = 1;
59 int giro = 0;
60
61 // ============================
62 // FUNCIONES AUXILIARES
63 // ============================
64
65 /* Convierte el valor del joystick en velocidad, dependiendo del sentido
66  * valor_joystick: Valor del joystick (0-255).
67  * sentido_val: 1 si es avance, 0 si es retroceso.
68  * Devuelve la velocidad normalizada.
69  */
70 unsigned char mapeo_joystick_velocidad(unsigned char valor_joystick,
       unsigned char sentido_val) {
71   if (sentido_val == 1) { // Adelante
72     if (valor_joystick <= 0x7F) return 0x00;
73     if (valor_joystick >= 0xFE) return 0xFD;
74     return (unsigned char)(((valor_joystick - 0x7F) * 0xFD) / (0xFE - 0x7F
          ));
75   } else { // Atras
76     if (valor_joystick <= 0x00) return 0xFD;
77     if (valor_joystick >= 0x7F) return 0x00;
78     return (unsigned char)(0xFD - ((valor_joystick * 0xFD) / 0x7F));
79   }
80 }
81
82 /* Escala de velocidad maxima segun el valor de vmax
83  * Devuelve el factor de escala para la velocidad segun el valor del
```

```
84  * selector vmax.
85  * vmax_val: Valor recibido del selector de velocidad maxima.
86  */
87 double escala_vmax(int vmax_val) {
88   switch (vmax_val) {
89   case 192:
90     return 1.0;
91   case 193:
92     return 0.85;
93   case 194:
94     return 0.70;
95   case 195:
96     return 0.50;
97   case 196:
98     return 0.35;
99   case 197:
100    return 0.20;
101  default:
102    return 1.0;
103  }
104 }
105
106 // ========================
107 // CALLBACKS ROS
108 // ========================
109
110 void cb_joystick_acc(const void * msgin) {
111   joystick_acc = ((std_msgs__msg__UInt8 * ) msgin) -> data;
112 }
113 void cb_joystick_turn(const void * msgin) {
114   joystick_turn = ((std_msgs__msg__UInt8 * ) msgin) -> data;
115 }
116 void cb_vmax(const void * msgin) {
117   valor_vmax = ((std_msgs__msg__UInt8 * ) msgin) -> data;
118 }
119 void cb_sentido(const void * msgin) {
120   sentido = ((std_msgs__msg__UInt8 * ) msgin) -> data;
121 }
122 void cb_giro(const void * msgin) {
123   giro = ((std_msgs__msg__UInt8 * ) msgin) -> data;
124 }
125
126 // ========================
127 // LOGICA DEL TIMER
```

```
128 // ======================
129
130 /* Se ejecuta periodicamente y calcula las velocidades y direcciones
131  * Callback del temporizador que ejecuta la logica de control cada 100ms.
132  * Calcula las velocidades motoras y direcciones basandose en el tipo de
133  * movimiento (Ackermann o giro sobre eje).
134  */
135 void timer_callback(rcl_timer_t * timer, int64_t last_call_time) {
136   (void) timer;
137   (void) last_call_time;
138
139   std_msgs__msg__Int32 motor_1_msg;
140   std_msgs__msg__Int32 motor_2_msg;
141
142   if (giro == 1) {
143     // Giro sobre si mismo: ruedas giran en sentidos opuestos, velocidad
          obtenida del joystick de giro
144     unsigned char sentido_izq, sentido_dcha;
145     unsigned char velocidad = 0x00;
146
147     if (joystick_turn > 0x7F) {
148       sentido_izq = 0x07; // izquierda adelante
149       sentido_dcha = 0x0B; // derecha atras
150       velocidad = mapeo_joystick_velocidad((unsigned char) joystick_turn,
            1);
151     } else if (joystick_turn < 0x7F) {
152       sentido_izq = 0x0B; // izquierda atras
153       sentido_dcha = 0x07; // derecha alante
154       velocidad = mapeo_joystick_velocidad((unsigned char) joystick_turn,
            0);
155     } else {
156       sentido_izq = 0x00;
157       sentido_dcha = 0x00;
158       velocidad = 0x00;
159     }
160
161     motor_1_msg.data = velocidad;
162     motor_2_msg.data = velocidad;
163
164     // Publicar sentidos individuales
165     std_msgs__msg__UInt8 giro_izq_msg = {
166       .data = sentido_izq
167     };
168     std_msgs__msg__UInt8 giro_dcha_msg = {
```

```
169       .data = sentido_dcha
170     };
171     rcl_publish( & pub_giro_izq, & giro_izq_msg, NULL);
172     rcl_publish( & pub_giro_dcha, & giro_dcha_msg, NULL);
173
174   } else {
175     // Movimiento tipo Ackermann: diferente velocidad por rueda en curva
176
177     unsigned char velocidad_izqda_raw = mapeo_joystick_velocidad((unsigned
          char) joystick_acc, sentido);
178     unsigned char velocidad_dcha_raw = mapeo_joystick_velocidad((unsigned
         char) joystick_acc, sentido);
179     double velocidad_izqda = (double) velocidad_izqda_raw;
180     double velocidad_dcha = (double) velocidad_dcha_raw;
181
182     // Calculo del angulo de direccion
183     double giro_volante = (double) joystick_turn;
184     double angulo_grad = 0.0;
185     double radio_giro = INFINITY;
186
187     if (giro_volante < 0x7F) // giro hacia la izquierda
188       angulo_grad = -GIRO_MAX_angulo_grad * ((0x7F - giro_volante) / 127.0)
            ; // mapeo
189     else if (giro_volante > 0x7F) // giro hacia la derecha
190       angulo_grad = GIRO_MAX_angulo_grad * ((giro_volante - 0x7F) / 127.0);
            // mapeo
191
192     if (angulo_grad != 0.0) {
193       double angulo_rad = angulo_grad * M_PI / 180.0;
194       radio_giro = fabs(DISTANCIAEJES_MM / tan(angulo_rad));
195     }
196
197     // Ajustar velocidades para Ackermann
198     if (radio_giro != INFINITY) {
199       double Ri = radio_giro - (ANCHOVIA_MM / 2.0);
200       double Ro = radio_giro + (ANCHOVIA_MM / 2.0);
201       double ratio = Ri / Ro;
202
203       if (giro_volante < 0x7F)
204         velocidad_izqda *= ratio;
205       else if (giro_volante > 0x7F)
206         velocidad_dcha *= ratio;
207     }
208
```

```
209     // Aplicar limitador vmax
210     double escalado = escala_vmax(valor_vmax);
211     velocidad_izqda *= escalado;
212     velocidad_dcha *= escalado;
213 
214     motor_1_msg.data = (int) velocidad_izqda;
215     motor_2_msg.data = (int) velocidad_dcha;
216 
217     // Publicar sentido
218     std_msgs__msg__UInt8 sentido_msg = {
219       .data = (sentido == 1) ? 0x07 : 0x0B
220     };
221     rcl_publish( & pub_sentido_ackermann, & sentido_msg, NULL);
222   }
223 
224   // Publicar velocidades de motores
225   rcl_publish( & pub_motor_1, & motor_1_msg, NULL);
226   rcl_publish( & pub_motor_2, & motor_2_msg, NULL);
227 }
228 
229 // =========================
230 // FUNCION PRINCIPAL
231 // =========================
232 
233 int main(int argc, char * argv[]) {
234   // Inicializar soporte y nodo ROS
235   rcl_allocator_t allocator = rcl_get_default_allocator();
236   rclc_support_t support;
237   rclc_support_init( & support, argc, argv, & allocator);
238   rcl_node_t node;
239   rclc_node_init_default( & node, "ackermann", "", & support);
240 
241   // Crear publishers
242   rclc_publisher_init_default( & pub_motor_1, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32), "motor_1");
243   rclc_publisher_init_default( & pub_motor_2, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32), "motor_2");
244   rclc_publisher_init_default( & pub_sentido_ackermann, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "sentido_ackermann
        ");
245   rclc_publisher_init_default( & pub_giro_izq, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "giro_izq");
246   rclc_publisher_init_default( & pub_giro_dcha, & node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "giro_dcha");
```

```
247
248   // Crear suscripciones
249   rclc_subscription_init_default( & sub_joystick_acc, & node,
          ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "joystick_acc");
250   rclc_subscription_init_default( & sub_joystick_turn, & node,
          ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "joystick_turn");
251   rclc_subscription_init_default( & sub_vmax, & node,
          ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "vmax");
252   rclc_subscription_init_default( & sub_sentido, & node,
          ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "sentido");
253   rclc_subscription_init_default( & sub_giro, & node,
          ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "giro");
254
255   // Crear temporizador para el bucle de control
256   rcl_timer_t timer;
257   rclc_timer_init_default( & timer, & support, RCL_MS_TO_NS(100),
          timer_callback);
258
259   // Executor para manejar callbacks
260   rclc_executor_t executor;
261   rclc_executor_init( & executor, & support.context, 7, & allocator);
262   rclc_executor_add_subscription( & executor, & sub_joystick_acc, & (
          std_msgs__msg__UInt8) {}, & cb_joystick_acc, ON_NEW_DATA);
263   rclc_executor_add_subscription( & executor, & sub_joystick_turn, & (
          std_msgs__msg__UInt8) {}, & cb_joystick_turn, ON_NEW_DATA);
264   rclc_executor_add_subscription( & executor, & sub_vmax, & (
          std_msgs__msg__UInt8) {}, & cb_vmax, ON_NEW_DATA);
265   rclc_executor_add_subscription( & executor, & sub_sentido, & (
          std_msgs__msg__UInt8) {}, & cb_sentido, ON_NEW_DATA);
266   rclc_executor_add_subscription( & executor, & sub_giro, & (
          std_msgs__msg__UInt8) {}, & cb_giro, ON_NEW_DATA);
267   rclc_executor_add_timer( & executor, & timer);
268
269   // Bucle principal del nodo
270   while (true) {
271     rclc_executor_spin_some( & executor, RCL_MS_TO_NS(100));
272   }
273
274   return 0;
275 }
```

Código A.5: Nodo en C que lee de distintos topics y publica velocidades calculadas mediante Ackermann en otros

emisor_can.c

```
1  /**
2   * Nodo que escucha distintos topics y transmite consignas por el bus CAN
       a
3   * traves de la interfaz can0.
4   * Se encarga de manejar los motores, direccion, giro y arranque/parada
5   * del sistema.
6   *
7   * Publica por CAN en los IDs 0x201, 0x202 (direccion), 0x301, 0x302
8   * (velocidades), 0x601, 0x602 (control/arranque).
9   */
10
11 // ======================
12 // INCLUDES
13 // ======================
14
15 // Librerias estandar de C
16 #include <stdio.h> // Entrada/salida estandar (printf, perror...)
17 #include <stdlib.h> // Funciones utiles (malloc, free, exit...)
18 #include <string.h> // Manipulacion de cadenas (strcpy, memset...)
19 #include <unistd.h> // Funciones del sistema (read, write...)
20 #include <errno.h> // Manejo de errores mediante errno
21
22 // Librerias para la configuracion y uso de sockets CAN
23 #include <net/if.h> // Informacion sobre interfaces de red (e.g., can0)
24 #include <sys/ioctl.h> // Control de dispositivos (ioctl)
25 #include <sys/socket.h> // API para trabajar con sockets
26 #include <linux/can.h> // Estructuras para mensajes CAN
27 #include <linux/can/raw.h> // Tipo de socket CAN_RAW para acceso directo
28 #include <signal.h> // Manejo de seales como SIGINT (Ctrl+C)
29 #include <stdbool.h> // Soporte para tipo booleano
30
31 // Librerias de ROS 2 (RCL y RCLC)
32 #include "rcl/rcl.h" // API principal de ROS 2 en C
33 #include "rclc/rclc.h" // Extension client library para C
34 #include "rclc/executor.h" // Ejecutores para callbacks de ROS 2
35
36 // Tipos de mensajes estandar de ROS 2
37 #include "std_msgs/msg/int32.h" // Mensaje entero con signo de 32 bits
38 #include "std_msgs/msg/u_int8.h" // Mensaje entero sin signo de 8 bits
39
40
41 // ======================
42 // VARIABLES GLOBALES
```

```
43 // ======================
44
45 // Descriptor del socket CAN y estructuras de configuracion
46 int socket_can;
47 struct sockaddr_can addr; // Direccion de socket para interfaz CAN
48 struct ifreq ifr; // Estructura para obtener indice de interfaz (e.g.,
       can0)
49
50 // Control de ejecucion: se pone en 0 al recibir SIGINT (Ctrl+C)
51 volatile sig_atomic_t running = 1;
52
53 // Variables que almacenan los comandos actuales y ultimos enviados para
       evitar envios redundantes
54 int valor_motor_1 = -1, ultimo_motor_1 = -1; // Velocidad motor 1
55 int valor_motor_2 = -1, ultimo_motor_2 = -1; // Velocidad motor 2
56
57 uint8_t sentido_ackermann = 0x07, last_sentido = 0xFF; // Direccion de
       movimiento general (adelante o atras)
58
59 uint8_t giro = 0; // Modo de movimiento: 0 = Ackermann, 1 = giro sobre eje
60 uint8_t giro_izq = 0x07, giro_dcha = 0x07; // Sentidos individuales de
       cada motor (modo giro)
61 uint8_t last_giro_izq = 0xFF, last_giro_dcha = 0xFF; // ultimos valores
       enviados de giro
62
63 uint8_t estado_arranque = 0; // Estado actual del arranque (0=nada, 1=
       start, 2=stop)
64 uint8_t ultimo_estado_arranque = 0; // ultimo estado publicado
65
66 // ======================
67 // FUNCIONES
68 // ======================
69
70 // Manejo de seal SIGINT (Ctrl+C) para terminar el bucle principal
71 void handle_sigint(int sig) {
72   (void) sig; // Ignorar el argumento
73   running = 0;
74 }
75
76 /*
77  * Envia un mensaje CAN por el socket con un ID y uno o dos bytes de datos
        .
78  * can_id: ID del mensaje CAN (ej. 0x301).
79  * data_byte: Byte principal a enviar.
```

```
80  * add_0x05: Si es true, se anade un segundo byte con valor 0x05. Sirve
81  * para las consignas de acelerador
82  */
83 void send_can_frame(uint16_t can_id, uint8_t data_byte, bool add_0x05) {
84   struct can_frame frame;
85   frame.can_id = can_id;
86
87   if (add_0x05) { // Consignas de aceleracion
88     frame.can_dlc = 2;
89     frame.data[0] = data_byte;
90     frame.data[1] = 0x05;
91   } else { // Consignas de control
92     frame.can_dlc = 1;
93     frame.data[0] = data_byte;
94   }
95
96   // Envio del frame por el socket CAN y registro en el terminal
97   int nbytes = write(socket_can, & frame, sizeof(struct can_frame));
98   if (nbytes < 0) {
99     perror("Error enviando CAN frame");
100  } else {
101    if (add_0x05)
102      printf("Enviado CAN ID: %X, Data: %02X 05\n", can_id, data_byte);
103    else
104      printf("Enviado CAN ID: %X, Data: %02X\n", can_id, data_byte);
105  }
106 }
107
108 // ================================
109 // CALLBACKS PARA SUSCRIPCIONES ROS
110 // ================================
111
112 void cb_motor_1(const void * msgin) {
113   valor_motor_1 = ((std_msgs__msg__Int32 * ) msgin) -> data;
114 }
115
116 void cb_motor_2(const void * msgin) {
117   valor_motor_2 = ((std_msgs__msg__Int32 * ) msgin) -> data;
118 }
119
120 void cb_sentido_ackermann(const void * msgin) {
121   sentido_ackermann = ((std_msgs__msg__UInt8 * ) msgin) -> data;
122 }
123
```

```
124 void cb_giro(const void * msgin) {
125   giro = ((std_msgs__msg__UInt8 * ) msgin) -> data;
126 }
127
128 void cb_giro_izq(const void * msgin) {
129   giro_izq = ((std_msgs__msg__UInt8 * ) msgin) -> data;
130 }
131
132 void cb_giro_dcha(const void * msgin) {
133   giro_dcha = ((std_msgs__msg__UInt8 * ) msgin) -> data;
134 }
135
136 void cb_arranque(const void * msgin) {
137   estado_arranque = ((std_msgs__msg__UInt8 * ) msgin) -> data;
138
139   // Si se ha recibido comando de arranque, reiniciar los estados
        anteriores; cierre de contactores
140   if (estado_arranque == 1) {
141     ultimo_motor_1 = -1;
142     ultimo_motor_2 = -1;
143     last_sentido = 0xFF;
144   }
145 }
146
147 // ======================
148 // PROCESAMIENTO PRINCIPAL
149 // ======================
150
151 /**
152  * Funcion principal de logica: compara estados actuales y anteriores y
153  * envia por CAN si hubo cambios para no saturar.
154  * Maneja logica de direccion, velocidad y arranque.
155  */
156 void process_can() {
157   // Enviar comandos de sentido de giro si esta seleccionada la rotacion
        sobre si mismo
158   if (giro == 1) {
159     if (giro_izq != last_giro_izq) {
160       send_can_frame(0x201, giro_izq, false);
161       last_giro_izq = giro_izq;
162     }
163     if (giro_dcha != last_giro_dcha) {
164       send_can_frame(0x202, giro_dcha, false);
165       last_giro_dcha = giro_dcha;
```

```
166     }
167   } else {
168     // Enviar direccion general en caso de movimiento con Ackermann
169     if (sentido_ackermann != last_sentido) {
170       send_can_frame(0x201, sentido_ackermann, false);
171       send_can_frame(0x202, sentido_ackermann, false);
172       last_sentido = sentido_ackermann;
173     }
174   }
175
176   // Enviar velocidad motor 1 si ha cambiado
177   if (valor_motor_1 != ultimo_motor_1) {
178     send_can_frame(0x301, (uint8_t)(valor_motor_1 & 0xFF), true);
179     ultimo_motor_1 = valor_motor_1;
180   }
181
182   // Enviar velocidad motor 2 si ha cambiado
183   if (valor_motor_2 != ultimo_motor_2) {
184     send_can_frame(0x302, (uint8_t)(valor_motor_2 & 0xFF), true);
185     ultimo_motor_2 = valor_motor_2;
186   }
187
188   // Si cambia el estado de arranque, enviar secuencia de control
189   if (estado_arranque != ultimo_estado_arranque) {
190     ultimo_estado_arranque = estado_arranque;
191
192     // Detener ambos motores, necesario para cerrar contactores
193     send_can_frame(0x201, 0x00, false);
194     send_can_frame(0x202, 0x00, false);
195
196     // Si se arranca, enviar contrasena de desbloqueo y poner velocidades
          a cero
197     if (estado_arranque == 1) {
198       struct can_frame frame_pass;
199       frame_pass.can_dlc = 8;
200       frame_pass.data[0] = 0x2B;
201       frame_pass.data[1] = 0x00;
202       frame_pass.data[2] = 0x50;
203       frame_pass.data[3] = 0x02;
204       frame_pass.data[4] = 0xDF;
205       frame_pass.data[5] = 0x4B;
206       frame_pass.data[6] = 0x00;
207       frame_pass.data[7] = 0x00;
208
```

```
209       frame_pass.can_id = 0x601;
210       write(socket_can, & frame_pass, sizeof(frame_pass));
211       printf("Enviando password: CAN ID 601, Data: 2B005002DF4B0000\n");
212
213       frame_pass.can_id = 0x602;
214       write(socket_can, & frame_pass, sizeof(frame_pass));
215       printf("Enviando password: CAN ID 602, Data: 2B005002DF4B0000\n");
216
217       // Enviar velocidad cero a ambos motores
218       struct can_frame frame_zero;
219       frame_zero.can_dlc = 2;
220       frame_zero.data[0] = 0x00;
221       frame_zero.data[1] = 0x00;
222
223       frame_zero.can_id = 0x301;
224       write(socket_can, & frame_zero, sizeof(frame_zero));
225       printf("Enviando CAN ID: 301, Data: 00 00\n");
226
227       frame_zero.can_id = 0x302;
228       write(socket_can, & frame_zero, sizeof(frame_zero));
229       printf("Enviando CAN ID: 302, Data: 00 00\n");
230     }
231
232     // Enviar comando de arranque (operacional) o parada (preoperacional)
          segun estado
233     struct can_frame frame;
234     frame.can_dlc = 8;
235     frame.data[0] = 0x2F;
236     frame.data[1] = 0x00;
237     frame.data[2] = 0x28;
238     frame.data[3] = 0x00;
239     frame.data[4] = (estado_arranque == 1) ? 0x00 : 0x01;
240     frame.data[5] = 0x00;
241     frame.data[6] = 0x00;
242     frame.data[7] = 0x00;
243
244     frame.can_id = 0x601;
245     write(socket_can, & frame, sizeof(frame));
246     printf("Enviado CAN ID: 601, Data: 2F002800%02X000000\n", frame.data
          [4]);
247
248     frame.can_id = 0x602;
249     write(socket_can, & frame, sizeof(frame));
```

```
250     printf("Enviado CAN ID: 602, Data: 2F002800%02X000000\n", frame.data
            [4]);
251   }
252 }
253
254 // =========================
255 // FUNCION PRINCIPAL
256 // =========================
257
258 int main(int argc, char * argv[]) {
259   // Capturar Ctrl+C para salir limpiamente
260   signal(SIGINT, handle_sigint);
261
262   // Crear socket CAN
263   socket_can = socket(PF_CAN, SOCK_RAW, CAN_RAW);
264   if (socket_can < 0) {
265     perror("Error abriendo socket CAN");
266     return 1;
267   }
268
269   // Asignar interfaz CAN (can0)
270   strcpy(ifr.ifr_name, "can0");
271   ioctl(socket_can, SIOCGIFINDEX, & ifr);
272
273   addr.can_family = AF_CAN;
274   addr.can_ifindex = ifr.ifr_ifindex;
275
276   if (bind(socket_can, (struct sockaddr * ) & addr, sizeof(addr)) < 0) {
277     perror("Error al hacer bind del socket CAN");
278     return 1;
279   }
280
281   // Inicializar ROS 2
282   rcl_allocator_t allocator = rcl_get_default_allocator();
283   rclc_support_t support;
284   rcl_node_t node;
285   rclc_support_init( & support, argc, argv, & allocator);
286   rclc_node_init_default( & node, "emisor_can", "", & support);
287
288   // Crear suscripciones
289   rcl_subscription_t sub_motor_1, sub_motor_2, sub_sentido, sub_giro,
          sub_giro_izq, sub_giro_dcha, sub_arranque;
290   rclc_subscription_init_default( & sub_motor_1, & node,
          ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32), "motor_1");
```

```
291  rclc_subscription_init_default( & sub_motor_2, & node,
         ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32), "motor_2");
292  rclc_subscription_init_default( & sub_sentido, & node,
         ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "sentido_ackermann
         ");
293  rclc_subscription_init_default( & sub_giro, & node,
         ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "giro");
294  rclc_subscription_init_default( & sub_giro_izq, & node,
         ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "giro_izq");
295  rclc_subscription_init_default( & sub_giro_dcha, & node,
         ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "giro_dcha");
296  rclc_subscription_init_default( & sub_arranque, & node,
         ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, UInt8), "arranque");
297
298  // Executor para manejar callbacks
299  rclc_executor_t executor;
300  rclc_executor_init( & executor, & support.context, 7, & allocator);
301  rclc_executor_add_subscription( & executor, & sub_motor_1, & (
         std_msgs__msg__Int32) {}, & cb_motor_1, ON_NEW_DATA);
302  rclc_executor_add_subscription( & executor, & sub_motor_2, & (
         std_msgs__msg__Int32) {}, & cb_motor_2, ON_NEW_DATA);
303  rclc_executor_add_subscription( & executor, & sub_sentido, & (
         std_msgs__msg__UInt8) {}, & cb_sentido_ackermann, ON_NEW_DATA);
304  rclc_executor_add_subscription( & executor, & sub_giro, & (
         std_msgs__msg__UInt8) {}, & cb_giro, ON_NEW_DATA);
305  rclc_executor_add_subscription( & executor, & sub_giro_izq, & (
         std_msgs__msg__UInt8) {}, & cb_giro_izq, ON_NEW_DATA);
306  rclc_executor_add_subscription( & executor, & sub_giro_dcha, & (
         std_msgs__msg__UInt8) {}, & cb_giro_dcha, ON_NEW_DATA);
307  rclc_executor_add_subscription( & executor, & sub_arranque, & (
         std_msgs__msg__UInt8) {}, & cb_arranque, ON_NEW_DATA);
308
309  // Bucle principal: ejecuta callbacks y procesa logica de envio
310  while (running) {
311    rclc_executor_spin_some( & executor, RCL_MS_TO_NS(100));
312    process_can();
313  }
314
315  // ======================
316  // FINALIZACION
317  // ======================
318
319  // Limpieza final
320  close(socket_can);
```

```
321
322   // Finalizar nodo ROS
323   rcl_node_fini( & node);
324   rclc_support_fini( & support);
325   return 0;
326 }
```

Código A.6: Nodo en C que lee de distintos topics y publica en bus CAN

APÉNDICE B

Puesta en marcha del ordenador

Este anexo se describe el procedimiento seguido para la instalación y configuración de todo el entorno software necesario para el desarrollo del sistema, incluyendo tanto el sistema operativo como los controladores y herramientas necesarias, así como el entorno ROS 2 *Humble*. Con estas configuraciones se deja el sistema preparado para su uso como controlador del sistema completo.

B.1. Instalación del sistema operativo

Para la instalación de Ubuntu 22.04 en el PC se hace uso de una memoria USB booteable generada mediante el software Rufus [41], a partir de una imagen ISO del sistema obtenida del sitio web oficial [27]. Durante el proceso de creación del medio, se configura la partición como `GPT` y el sistema de destino como `UEFI (no CSM)` (Figura B.2), siguiendo las recomendaciones de la página oficial del sistema operativo para un funcionamiento correcto con el PC.

Figura B.1: Software para crear la memoria USB booteable. Fuente [41]

Para poder iniciar la instalación del sistema operativo, es necesario conectar el USB y acceder a la BIOS del equipo para establecerlo como la opción primaria de arranque. En el caso del Advantech UNO-2484G, esto se realiza al encender el equipo y pulsar repetidamente la tecla `ESC` mientras está arrancando. Una vez abierto el menú de configuración, hay que ir hasta la sección *Boot*, donde se establece la unidad USB como primera opción.

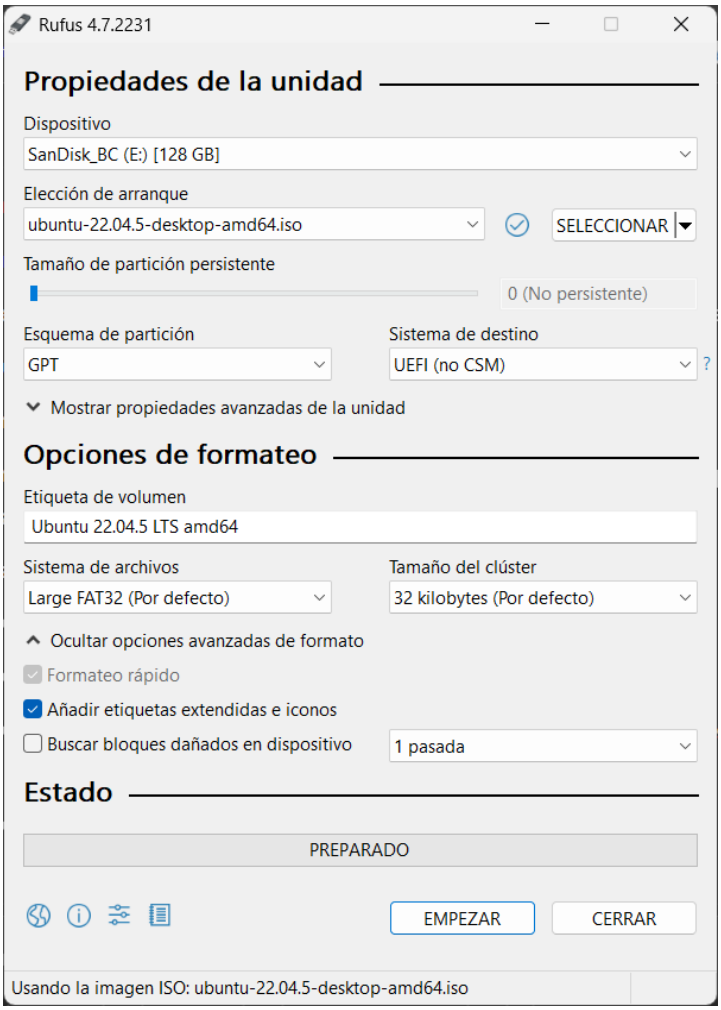


Figura B.2: Configuración de Rufus utilizada. Fuente: Elaboración propia

Tras haber hecho lo anterior y reiniciar el equipo, comienza automáticamente el proceso de instalación, donde se opta por una instalación mínima con el objetivo de reducir la cantidad de software preinstalado (navegadores, aplicaciones multimedia y de oficina, etc.) y favorecer un entorno más ligero y con menos consumo de recursos para el desarrollo del proyecto.

Finalizada la instalación, se procede a actualizar los paquetes del sistema mediante los siguientes comandos:

```
sudo apt update
sudo apt upgrade
```

Código B.1: Actualización del sistema tras la instalación

Con ello se asegura un entorno actualizado y preparado para empezar a funcionar.

B.2. Instalación de ROS 2 *Humble*

A continuación, se describen los pasos necesarios para la instalación de la distribución *Humble Hawksbill* de ROS 2 [6], compatible con Ubuntu 22.04 [27].

En primer lugar, es preciso configurar los repositorios de ROS 2 en el sistema. Para ello, se instalan los paquetes auxiliares `curl`, `gnupg2`, `lsb-release` y `software-properties-common` mediante el comando:

```
sudo apt install -y curl gnupg2 lsb-release software-properties-common
```

Posteriormente, se descarga y configura la clave GPG necesaria para verificar la autenticidad de los paquetes:

```
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.
    asc | sudo gpg --dearmor -o /usr/share/keyrings/ros-archive-keyring.
    gpg
```

Una vez añadida la clave, se incorpora el repositorio de ROS 2 al sistema de gestión de paquetes:

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings
    /ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(
    lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/ros2.list >
    /dev/null
```

Con los repositorios configurados, se actualizan los paquetes del sistema y se instala la distribución completa de ROS 2 *Humble*:

```
sudo apt update
sudo apt install -y ros-humble-desktop
```

Para facilitar el uso de ROS 2 en cada nueva sesión de terminal, se incluye la carga automática del entorno en el archivo `.bashrc`:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Además, resulta necesario instalar las herramientas de compilación y las extensiones comunes de `colcon`, así como configurar el sistema de gestión de dependencias de ROS 2, `rosdep`:

```
sudo apt install python3-colcon-common-extensions python3-rosdep build-
    essential
sudo rosdep init
rosdep update
```

Con esto, la instalación habrá finalizado.

B.3. Configuración de los puertos CAN

Para configurar los puertos CAN, se descargan desde la página oficial de Advantech los controladores necesarios para el módulo NKP-SJA1000 [42] que monta el PCM-26D2CA-BE:

- `advsocketcan`: controlador base para tarjetas CAN de Advantech.
- `advcan_sja1000`: controlador específico para tarjetas con el chip SJA1000 como la que se utiliza en este caso.

Ambos controladores se descargan en una carpeta comprimida, al descomprimirla se accede al directorio de la nueva carpeta desde un terminal y se ejecutan ambos con:

```
make
sudo make install
```

Código B.2: Compilación e instalación de los drivers

Tras esto, los módulos se cargan en el sistema mediante:

```
sudo modprobe advsocketcan.ko
sudo modprobe advcan_sja1000.ko
```

Código B.3: Carga de módulos

El sistema ya reconoce las interfaces CAN como `can0` y `can1`, para completar la configuración, es necesario cargar también los módulos estándar del stack SocketCAN de Linux, que permiten la comunicación mediante sockets:

```
sudo modprobe can
sudo modprobe can_raw
sudo modprobe can_dev
sudo modprobe can_bcm
sudo modprobe can_gw
```

Código B.4: Carga de módulos SocketCAN

A continuación, se comprueba que las interfaces se han creado correctamente con el siguiente comando:

```
ip link show
```

Si las interfaces `can0` y `can1` aparecen, habrá finalizado el proceso de configuración inicial del módulo CAN.

APÉNDICE C

Scripts de inicialización del puerto can0

Este anexo describe el proceso para automatizar la configuración de la interfaz CAN al arrancar el sistema mediante un script de inicialización. A continuación, se detallan los pasos y el contenido del script desarrollado.

Script de configuración de CAN

Script que configura los módulos necesarios para la comunicación CAN y establece la interfaz `can0`, se llamará `inicio_can.sh` y se ubicará en la ruta `/usr/local/bin/` para que sea accesible globalmente.

```
#!/bin/bash

# Se cargan los modulos estandar de CAN
modprobe can
modprobe can_raw
modprobe can_dev
modprobe can_bcm
modprobe can_gw

# Se selecciona el directorio donde se encuentran los drivers
cd /home/panter/Escritorio/driver-msi

# Por ultimo, se cargan los controladores especificos de Advantech
insmod advcan_sja1000.ko
insmod advsocketcan.ko

# Se inicializa la interfaz con una velocidad de 500 kbps
ip link set can0 up type can bitrate 500000
```

Código C.1: Script de configuración de la interfaz CAN

Permisos de ejecución

Para poder ejecutar el script, es necesario concederle permisos de ejecución con el siguiente comando:

```
sudo chmod +x /usr/local/bin/inicio_can.sh
```

Código C.2: Dar permisos de ejecucion al script

Automatización con systemd

Para que el script se ejecute automáticamente al iniciar el sistema, se crea un servicio `systemd` que permite que no sea necesaria la intervención manual.

El primer paso consiste en crear un archivo de servicio en `/etc/systemd/system/`. Este archivo contiene lo siguiente:

```
[Unit]
Description=Configurar interfaces CAN
After=network.target

[Service]
ExecStart=/usr/local/bin/inicio_can.sh
Restart=always
User=root

[Install]
WantedBy=multi-user.target
```

Código C.3: Contenido del archivo de servicio

En el archivo de servicio, se definen los parámetros para asegurarse de que el script se ejecute automáticamente tras el arranque del sistema. Incluye las siguientes funciones:

- `Description`: Se describe el servicio.
- `After=network.target`: Garantiza que el servicio se ejecute solo después de que la red esté configurada.
- `ExecStart`: Ubicación y script que se ejecutara.
- `Restart=always`: El servicio se reiniciara automáticamente si falla.
- `User=root`: El script se ejecutara con privilegios de superusuario (necesarios para los módulos estándar y los drivers en Código C.1).

- `WantedBy=multi-user.target`: Nivel de ejecución en el que se debe activar el servicio.

Una vez creado el archivo, es necesario habilitarlo para que se inicie automáticamente al arrancar el sistema:

```
sudo systemctl enable inicio_can.service
```

Código C.4: Habilitar el servicio systemd

Se puede iniciar el servicio de manera manual utilizando el siguiente comando:

```
sudo systemctl start inicio_can.service
```

Código C.5: Iniciar el servicio manualmente

Finalmente, es posible comprobar el estado del servicio con:

```
sudo systemctl status inicio_can.service
```

Código C.6: Verificar el estado del servicio

Con esto ya se configura automáticamente el puerto `can0` al arrancar el PC.

APÉNDICE D

Diccionario de consignas CAN

En este anexo se presenta el diccionario de consignas empleadas para la comunicación mediante bus CAN entre el sistema de control del vehículo y las controladoras de los motores, recoge los identificadores, formatos y contenido de cada tipo de mensaje utilizado.

```
----- PASSWORD -----
cansend can0 601#2B005002DF4B0000
cansend can0 602#2B005002DF4B0000

----- MODOS CONTROLADORA 1 -----
cansend can0 601#2F00280000000000 # Pasar a operacional
cansend can0 601#2F00280001000000 # Pasar a preoperacional

----- MODOS CONTROLADORA 2 -----
cansend can0 602#2F00280000000000 # Pasar a operacional
cansend can0 602#2F00280001000000 # Pasar a preoperacional

----- INICIALIZAR IO -----
cansend can0 201#00
cansend can0 301#0000
cansend can0 202#00
cansend can0 302#0000
```

A efectos de simplificación, en el diccionario de consignas de acelerador mostrado a continuación, el identificador `ID` corresponde a:

- `301` para la primera controladora (motor trasero izquierdo).
- `302` para la segunda controladora (motor trasero derecho).

```

----- SIMULAR ACELERACION -----
cansend can0 ID#0005 # 5.00V
cansend can0 ID#0205 # 5.01V
```

```
cansend can0 ID#0505 # 5.02V
cansend can0 ID#0705 # 5.03V
cansend can0 ID#0A05 # 5.04V
cansend can0 ID#0C05 # 5.05V
cansend can0 ID#0F05 # 5.06V
cansend can0 ID#1105 # 5.07V
cansend can0 ID#1405 # 5.08V
cansend can0 ID#1705 # 5.09V
cansend can0 ID#1905 # 5.10V
cansend can0 ID#1C05 # 5.11V
cansend can0 ID#1E05 # 5.12V
cansend can0 ID#2105 # 5.13V
cansend can0 ID#2305 # 5.14V
cansend can0 ID#2605 # 5.15V
cansend can0 ID#2805 # 5.16V
cansend can0 ID#2B05 # 5.17V
cansend can0 ID#2E05 # 5.18V
cansend can0 ID#3005 # 5.19V
cansend can0 ID#3305 # 5.20V
cansend can0 ID#3505 # 5.21V
cansend can0 ID#3805 # 5.22V
cansend can0 ID#3A05 # 5.23V
cansend can0 ID#3D05 # 5.24V
cansend can0 ID#4005 # 5.25V
cansend can0 ID#4205 # 5.26V
cansend can0 ID#4505 # 5.27V
cansend can0 ID#4705 # 5.28V
cansend can0 ID#4A05 # 5.29V
cansend can0 ID#4C05 # 5.30V
cansend can0 ID#4F05 # 5.31V
cansend can0 ID#5105 # 5.32V
cansend can0 ID#5405 # 5.33V
cansend can0 ID#5705 # 5.34V
cansend can0 ID#5905 # 5.35V
cansend can0 ID#5C05 # 5.36V
cansend can0 ID#6105 # 5.37V
cansend can0 ID#6305 # 5.39V
cansend can0 ID#6605 # 5.40V
cansend can0 ID#6805 # 5.41V
cansend can0 ID#6B05 # 5.42V
cansend can0 ID#6E05 # 5.43V
cansend can0 ID#7005 # 5.44V
cansend can0 ID#7305 # 5.45V
cansend can0 ID#7505 # 5.46V
```

```
cansend can0 ID#7805 # 5.47V
cansend can0 ID#7A05 # 5.48V
cansend can0 ID#7D05 # 5.49V
cansend can0 ID#8005 # 5.50V
cansend can0 ID#8205 # 5.51V
cansend can0 ID#8505 # 5.52V
cansend can0 ID#8705 # 5.53V
cansend can0 ID#8A05 # 5.54V
cansend can0 ID#8C05 # 5.55V
cansend can0 ID#8F05 # 5.56V
cansend can0 ID#9105 # 5.57V
cansend can0 ID#9405 # 5.58V
cansend can0 ID#9705 # 5.59V
cansend can0 ID#9905 # 5.60V
cansend can0 ID#9C05 # 5.61V
cansend can0 ID#9E05 # 5.62V
cansend can0 ID#A105 # 5.63V
cansend can0 ID#A305 # 5.64V
cansend can0 ID#A605 # 5.65V
cansend can0 ID#A805 # 5.66V
cansend can0 ID#AB05 # 5.67V
cansend can0 ID#AE05 # 5.68V
cansend can0 ID#B005 # 5.69V
cansend can0 ID#B305 # 5.70V
cansend can0 ID#B505 # 5.71V
cansend can0 ID#B805 # 5.72V
cansend can0 ID#BA05 # 5.73V
cansend can0 ID#BD05 # 5.74V
cansend can0 ID#C005 # 5.75V
cansend can0 ID#C205 # 5.76V
cansend can0 ID#C505 # 5.77V
cansend can0 ID#C705 # 5.78V
cansend can0 ID#CA05 # 5.79V
cansend can0 ID#CC05 # 5.80V
cansend can0 ID#CF05 # 5.81V
cansend can0 ID#D105 # 5.82V
cansend can0 ID#D405 # 5.83V
cansend can0 ID#D705 # 5.84V
cansend can0 ID#D905 # 5.85V
cansend can0 ID#DC05 # 5.86V
cansend can0 ID#DE05 # 5.87V
cansend can0 ID#E105 # 5.88V
cansend can0 ID#E305 # 5.89V
cansend can0 ID#E605 # 5.90V
```

```
cansend can0 ID#E805 # 5.91V
cansend can0 ID#EB05 # 5.92V
cansend can0 ID#EE05 # 5.93V
cansend can0 ID#F005 # 5.94V
cansend can0 ID#F305 # 5.95V
cansend can0 ID#F505 # 5.96V
cansend can0 ID#F805 # 5.97V
cansend can0 ID#FA05 # 5.98V
cansend can0 ID#FD05 # 5.99V
```

APÉNDICE E

Documentación Scanreco

En este anexo se adjunta la documentación proporcionada por el fabricante correspondiente al conjunto Scanreco Mini y Scanreco G3B (mando y receptor respectivamente). Esta documentación incluye los esquemas de conexión y las indicaciones necesarias para su correcta instalación y configuración. Fuente: [38]

TECHNICAL SPECIFICATION

SDC0002ou7437 Remote Control System G3B CAN
Universidad de Málaga
Test - Mini Joystick 433MHz

Revision A-1: 2023-12-18

This technical specification is a complement to the Instruction Manual.

Document Index:

Revision history
System overview
Transmitter
Receiver
System settings

REVISION HISTORY

Rev.	Date	Sign.	Revision note
A-1	2023-12-18	MC	First version

WARNING!

Read and understand the separate Scanreco instruction manual before proceeding with the installation, maintenance, or operation of the remote-control system. Failure to follow the instructions in the manual could result in death or serious injury.

DISCLAIMER

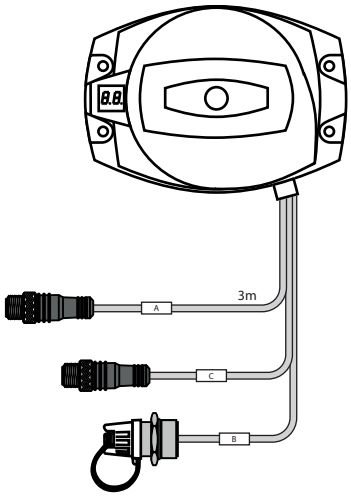
IT IS THE RESPONSIBILITY OF THE SYSTEM INSTALLER TO CORRECTLY IMPLEMENT AND INSTALL THE SCANRECO RADIO REMOTE CONTROL SYSTEM FOR A SPECIFIC MACHINE.

THE SYSTEM INSTALLER ALSO HAS THE RESPONSIBILITY TO MAKE SURE THAT THE SCANRECO RADIO REMOTE CONTROL SYSTEM IS INSTALLED IN ACCORDANCE WITH ALL COUNTRY, FEDERAL, STATE, LOCAL AND PRIVATE SAFETY AND HEALTH REGULATIONS, CODES, AND STANDARDS. SCANRECO DOES NOT TAKE RESPONSIBILITY FOR ANY DAMAGE OR INJURY CAUSED BY INADEQUATE SAFETY IMPLEMENTATIONS.

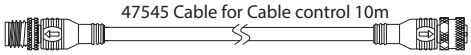
IF THE SCANRECO RADIO REMOTE CONTROL SYSTEM IS USED IN A SAFETY CRITICAL APPLICATION, THE SYSTEM INSTALLER MUST DO THE APPROPRIATE TESTING AND ANALYSIS OF THE FINAL APPLICATION TO PREVENT INJURY TO THE END USER.

SYSTEM OVERVIEW

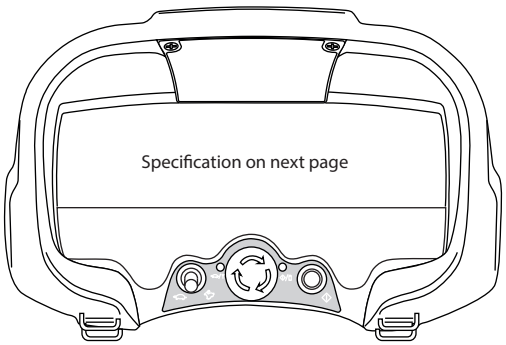
Receiver Unit CANOpen



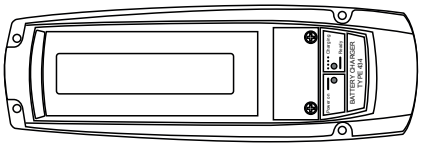
47545 Cable for Cable control 10m



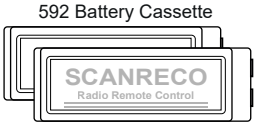
Transmitter Mini Joystick



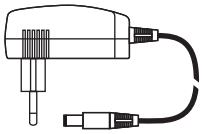
Battery Charger



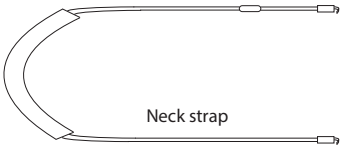
592 Battery Cassette



AC adaptor



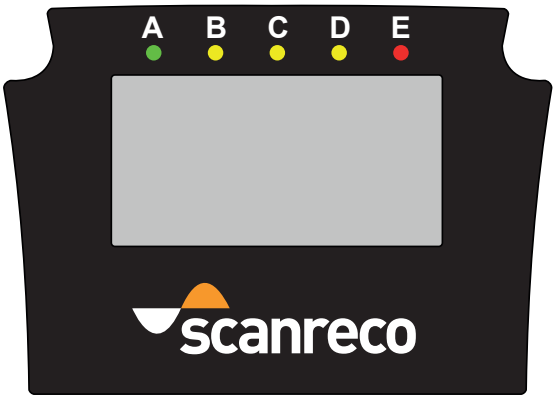
User Manual

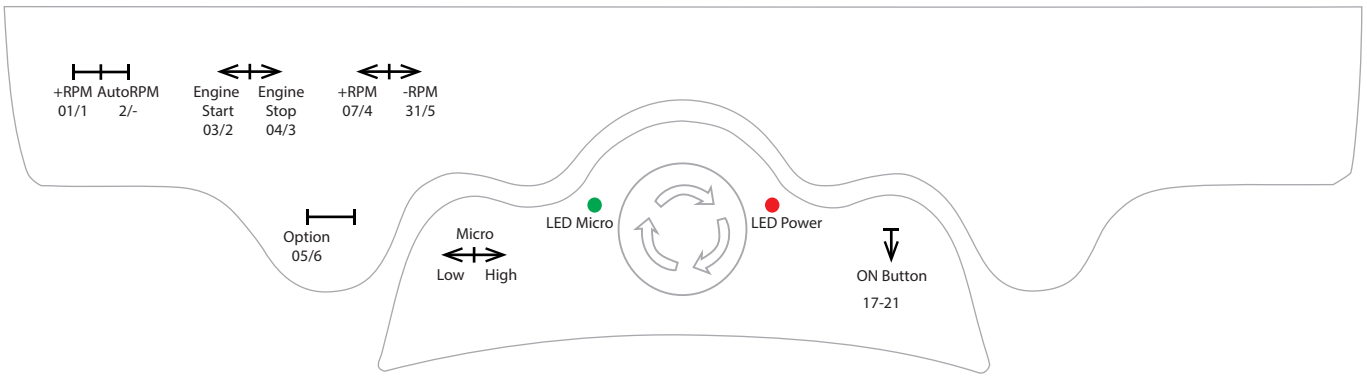


SYSTEM CONTENT

Part nr:	Description:	Qty:	Notes:
SDC0008ou7437	Transmitter	1	
1604ou7437	Receiver	1	
47545	Cable control 10m	1	
434	Battery charger	1	Including 48726 - Net adaptor
592	Battery Cassette	2	
44512	Neck strap	1	
66024	User Manual G2 English	1	
66066	User Manual G3 English	1	

TRANSMITTER





Push button.

2-way detent switch.

2-way spring return switch.

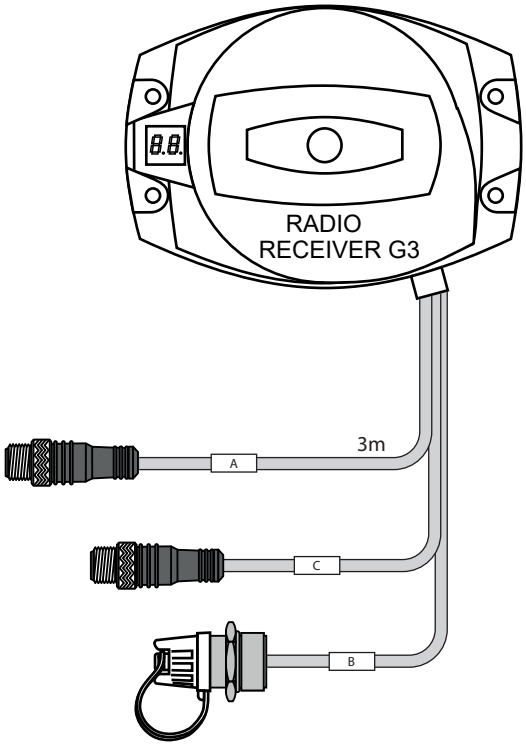
3-way spring return/detent with center position.

3-way detent with center position.

3-way spring return with center position.

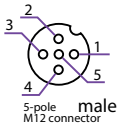
X-Y X: Digital function - Y: Digital output

RECEIVER



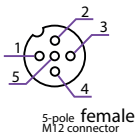
Cable A

Connection	
Pin no.	Colour / Function
1	Brown / DV1+
2	White / Power supply +12/24VDC
3	Blue / GND / CAN_GND
4	Black / CAN_HIGH
5	Grey / CAN_LOW



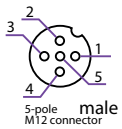
Cable B

Connection	
Pin no.	Colour / Function
1	Brown / Data
2	White / GND
3	Blue / RS 232 TX
4	Black / RS 232 RX
5	Grey / Supply output/CAB+



Cable C

Connection	
Pin no.	Colour / Function
1	Brown / System operational state
2	White / LOOP1_OUT
3	Blue / LOOP1_IN
4	Black / LOOP2_OUT
5	Grey / LOOP2_IN



SYSTEM SETTINGS

General data:

Radio platform:	TR02 433MHz
Transmitter firmware:	Standard
Receiver firmware:	Standard
Application program:	ou7437a1.txt
Graphic file:	Graphics application supplied by the customer

Transmitter settings:

Auto-OFF timer	30 minutes from idle operation on analogue outputs

DV1-output settings:

Auto-OFF timer:	0,5 sec from idle operation on analogue outputs

CAN SETTINGS

Type:	CANopen (Slave)
Baud rate:	250k
CAN ID:	100

TPDO1, COB-ID 0x180 + node id.

Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
Function 1	**Function 2**	**Function 3**	**Function 4**	**Function 5**	**Function 6**	**Function 7**	**Function 8**
0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir	0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir	0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir	0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir	0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir	0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir	0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir	0xFE = 100% A dir 0x7F = Idle 0x00 = 100% B dir

TPDO2, COB-ID 0x280 + node id.

Byte0	Byte1	Byte2	Byte3
0 : 0x01 : DFC01 - +RPM 1 : 0x02 : DFC02 - Engine Start 2 : 0x04 : DFC03 - Engine Stop 3 : 0x08 : DFC04 - RPM+ 4 : 0x10 : DFC05 - RPM- 5 : 0x20 : DFC06 - Option 1 6 : 0x40 : DFC07 - 7 : 0x80 : DFC08 -	0 : 0x01 : DFC09 - 1 : 0x02 : DFC10 - 2 : 0x04 : DFC11 - 3 : 0x08 : DFC12 - 4 : 0x10 : DFC13 - 5 : 0x20 : DFC14 - 6 : 0x40 : DFC15 - 7 : 0x80 : DFC16 -	0xFF = STOP No operational link between transmitter and receiver present. 0x00 = RUN Operational link between transmitter and receiver present.	0 : 0x01 : DFC17 - 1 : 0x02 : DFC18 - 2 : 0x04 : DFC19 - 3 : 0x08 : DFC20 - 4 : 0x10 : DFC21 - ON Button 5 : 0x20 : DFC22 - Cable 6 : 0x40 : DFC23 - 7 : 0x80 : DFC24 -
Byte4	**Byte5**	**Byte6**	**Byte7**
0 : 0x01 : DFC25 - UP 1 : 0x02 : DFC26 - Down 2 : 0x04 : DFC27 - F1 3 : 0x08 : DFC28 - F2 4 : 0x10 : DFC29 - F3 5 : 0x20 : DFC30 -F4 6 : 0x40 : DFC31 -Esc 7 : 0x80 : DFC32 -Enter	0 - Micro Level, 0x00 - 0x05 = level 1-level 5 1 - Micro Level 2 - Micro Level 3 : 0x08 - Micro reduce / OFF 4 : 0x10 - Micro increase / ON 5 : 0x20 - Stop reason, 0x00 = timeout, 0x01 = Stop button 6 - Radio Quality, 0x00 = 0-25%; 0x01 = 25-50% 7 - Radio Quality, 0x02 = 50-75%; 0x03 = 75-100%	0 - Battery Level, 0x00 = 0-25%; 0x01 = 25-50% 1 - Battery Level, 0x02 = 50-75%; 0x03 = 75-100% 2 - 3 - 4 - 5 - 6 - 7 -	0 : 0x01 : DFC33 - 1 : 0x02 : DFC34 - 2 : 0x04 : DFC35 - 3 : 0x08 : DFC36 - 4 : 0x10 : DFC37 - 5 : 0x20 : DFC38 - 6 : 0x40 : DFC39 - 7 : 0x80 : DFC40 -

RPDO2, COB-ID 0x300 + node id.

Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
0xFF	0 : 0x01 - LED A 1 : 0x02 - LED B 2 : 0x04 - LED C 3 : 0x08 - LED D 4 : 0x10 - not used 5 : 0x20 - Buzzer 6 : 0x40 - not used 7 : 0x80 - not used	0x00	0x00	0x00	0x00	0x00	0x00

Please see SCANRECOU-12132 G2B and G2B CANopen.pdf for detailes.

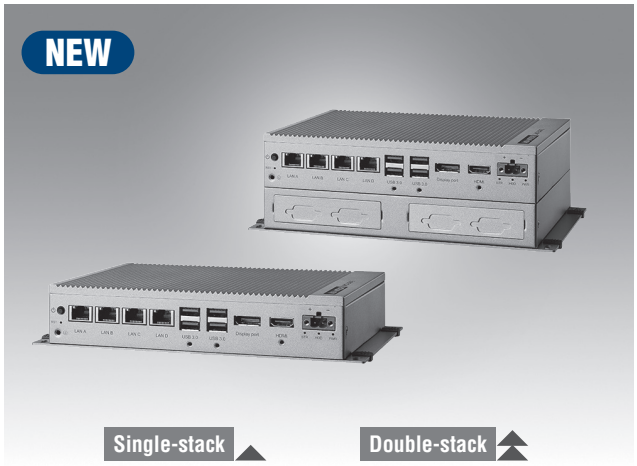
APÉNDICE F

Documentación Advantech UNO-2484G

Se adjunta el datasheet del ordenador industrial utilizado en este vehículo, el Advantech UNO-2484G, contiene también información sobre la extensión EKBE con la que se trabaja. Este documento, obtenido del fabricante, recoge las especificaciones y detalles más relevantes para su inclusión en el sistema. Fuente: [24]

UNO-2484G

Intel® Core™ i7/i5/i3/Celeron Regular-Size Modular Box Platform (MBP) with 4 x GbE, 1 x mPCIe, HDMI, DP



Features

- Intel® Core™ i7/i5/i3/Celeron Processor with 8GB DDR4 built-in memory
- 4 x GbE, 4 x USB 3.0, 1 x HDMI, 1 x DP, 4 x RS232/422/485
- Stackable 2nd layer for up to 4 iDoor extension or customize for domain applications
- Compact fanless design
- Ruggedized by zero cable and lockable I/O design
- Diverse system I/O and isolated digital I/O by iDoor technology
- Supports Fieldbus protocol by iDoor technology
- 3G/GPS/GPRS/Wi-Fi communication by iDoor technology
- Supports 30+ iDOOR combination with four main categories

Introduction

Advantech's new generation UNO-2000 series of Embedded Automation Computers are fanless and highly ruggedized with embedded OS. New UNO-2000 series are modular designs which provide flexible and time-to-market support in variety of applications. The series also includes iDOOR technology which supports automation feature-extensions such as multiple I/O peripherals, Industrial Fieldbus, and smart I/O Communication. New UNO-2000 series including pocket, small, and regular-size form-factors for smart factory applications such as Equipment Connectivity (EC), Process Visualization (PV), Environment Management (EM), and Dispatch Management (DM) solutions.

Specifications

General

- **Certification** CE, FCC, UL, CCC, BSMI
- **Dimensions (W x D x H)** 200 x 140 x 40 (7.8" x 5.6" x 1.6") for single stack UNO-2484G version
  200 x 140 x 70 (7.8" x 5.6" x 2.8") for double stack UNO-2484G version
- **Form Factor** Regular Size with stackable design
- **Enclosure** Aluminum Housing
- **Mounting** Stand, Wall, VESA (Optional), DIN-rail (Optional)
- **Weight (Net)** Single stack: 1.4kg (3.09lbs)
  Double stack: 1.8kg (3.97lbs)
- **Power Requirement** 10 - 36 V_{DC}
- **Power Consumption** 55W (Typical), 95.2W (Max)
- **OS Support** Microsoft® Windows 7, 10, Advantech Linux (UNO-2484G-6 series)
  Microsoft® Windows 10, Advantech Linux (UNO-2484G-7 series)

System Hardware

- **BIOS** AMI 128 Mbit SPI
- **Watchdog Timer** Programmable 255 levels timer interval, from 1 to 255 sec
- **Hardware Security** TPM2.0 (Optional)
- **Processor** 6th Gen Intel Core i7-6600U, 2.6GHz
  i5-6300U, 2.4GHz
  i3-6100U, 2.3GHz
  7th Gen Intel Core i7-7600U, 2.8GHz
  i5-7300U, 2.6GHz
  i3-7100U, 2.4GHz
  Celeron 3965U, 2.0 GHz
- **Memory** i7/i5/i3 model: Built-in 8GB DDR4 2133 MHz (support up to 16G)
  Celeron model: Built-in 4GB DDR4 2133 MHz (support up to 16G)
- **Graphics Engine** Intel® HD Graphics 520/620
- **Ethernet** 4 x RJ45, 10/100/1000 Mbps
- **LED Indicators** LEDs for Power, HDD, LAN (Active status), RTC Battery low
- **Storage** One mSATA (slot shared with mPCIe slot)
  Two drive bay for SATA 2.5" SSD/HDD (Support RAID 0/1)
- **Expansion** 1 x full-size mPCIe slot for single stack version (only support wireless iDOOR)
  4 x full-size mPCIe slots for double stack version (1 slot for wireless iDOOR, 3 slots for all iDOOR modules)

I/O Interfaces

- **Serial Ports** 4 x RS 232/422/485, 50-115.2 kbps speed
- **LAN Ports** 4 x RJ45, 10/100/1000 Mbps
  1000BASE-T Fast Ethernet
- **USB Ports** 4 x USB3.0
- **Displays** 1 x HDMI, 1 x DP
  HDMI 1.4 (4096x2304@24Hz)
  HDMI 1.4 (1920x1080@60Hz)
  DP 1.2 (4096x2304@60Hz)
  DP 1.2 (1920x1080@60Hz)
- **Audio** Line-out
- **Power Connector** 1 x 2 Pins, Terminal Block

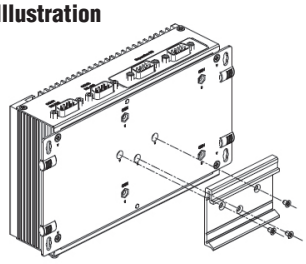
Environment

- **Operating Temperature** -20 ~ 60°C (-4 ~ 140°F) @ 5 ~ 85% RH with 0.7 m/s airflow
- **Storage Temperature** -40 ~ 85°C (-40 ~ 185°F)
- **Relative Humidity** 10 ~ 95% RH @ 40°C, non-condensing
- **Shock Protection** Operating, IEC 60068-2-27, 50G, half sine, 11 ms
- **Vibration Protection** Operating, IEC 60068-2-64, 3 Grms, random, 5 ~ 500 Hz, 1hr/axis (mSATA)
- **Ingress Protection** IP40

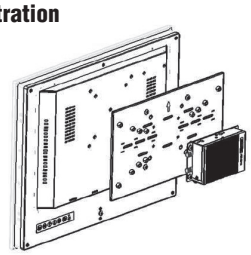
Last updated: 5-Jun-2023

Installation Scenario

DIN-rail Mount Illustration



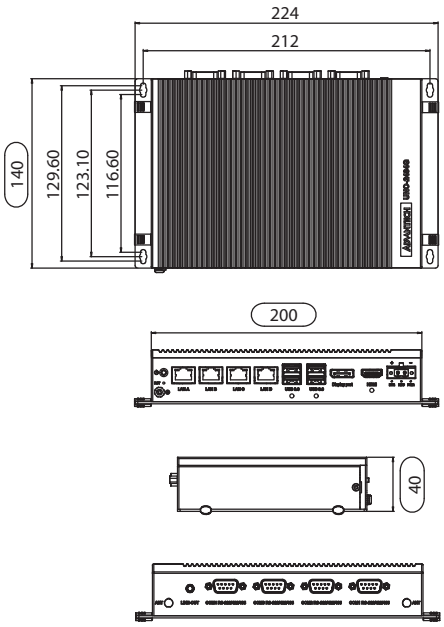
VESA Mount Illustration



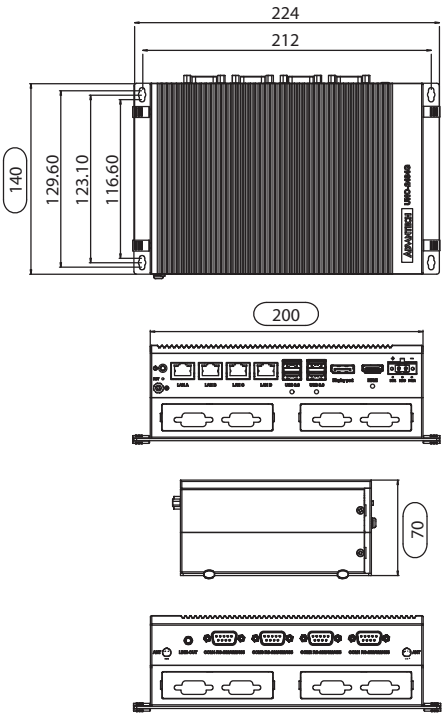
Dimensions

Unit: mm

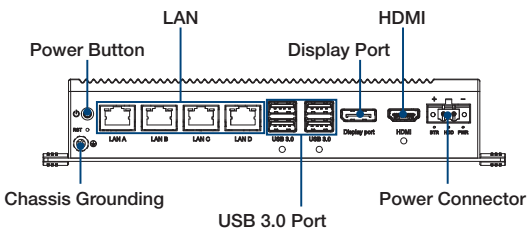
UNO-2484G Single stack version



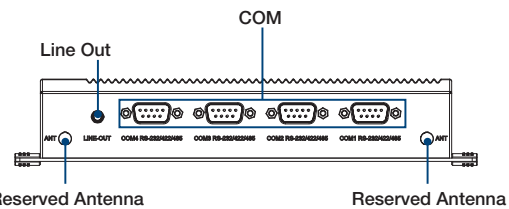
UNO-2484G double stack version



Front I/O View



Rear I/O Veiw



Ordering Information

Ordering Part Number		CPU Generation	CPU	LAN B-D IC	TPM2.0	mPCIe slot	iDoor	Audio	USB	COM	LAN
Single Stack	UNO-2484G-7731BE	Gen 7	i7-7600U	Realtek i8119i	X	1	x	Line Out	4	4	4
	UNO-2484G-7531BE		i5-7300U								
	UNO-2484G-7331BE		i3-7100U								
	UNO-2484G-7C21BE		Celeron 3965U								
	UNO-2484G-6731BE	Gen 6	i7-6600U								
	UNO-2484G-6531BE		i5-6300U								
	UNO-2484G-6331BE		i3-6100U								
Double Stack	UNO-2484G-7732BE	Gen 7	i7-7600U			4	4				
	UNO-2484G-7532BE		i5-7300U								
	UNO-2484G-7332BE		i3-7100U								
	UNO-2484G-7C22BE		Celeron 3965U								
	UNO-2484G-6732BE	Gen 6	i7-6600U								
	UNO-2484G-6532BE		i5-6300U								
	UNO-2484G-6332BE		i3-6100U								

***Below Part Number had been announced end of life**

Ordering Part Number		CPU Generation	CPU	LAN B-D IC	TPM2.0	mPCIe slot	iDoor	Audio	USB	COM	LAN
Single Stack	UNO-2484G-7731AE	Gen 7	i7-7600U	Intel® i210	Yes	1	x	Line Out	4	4	4
	UNO-2484G-7531AE		i5-7300U								
	UNO-2484G-7331AE		i3-7100U								
	UNO-2484G-7C21AE		Celeron 3965U								
	UNO-2484G-6731AE	Gen 6	i7-6600U								
	UNO-2484G-6531AE		i5-6300U								
	UNO-2484G-6331AE		i3-6100U								
Double Stack	UNO-2484G-7732AE	Gen 7	i7-7600U			4	4				
	UNO-2484G-7532AE		i5-7300U								
	UNO-2484G-7332AE		i3-7100U								
	UNO-2484G-7C22AE		Celeron 3965U								
	UNO-2484G-6732AE	Gen 6	i7-6600U								
	UNO-2484G-6532AE		i5-6300U								
	UNO-2484G-6332AE		i3-6100U								

iDoor Modules

Supported by "Double Stack" models. Applied on "Single stack" models with 2nd stack extension kit as well

- **PCM-24S2WF-CE** 802.11 a/b/g/n 2T2R w/ Bluetooth4.0, Half-size mPCIe, 2-port SMA
- **PCM-24R2GL-AE** 2-Port Gigabit Ethernet, mPCIe, RJ45
- **PCM-24D2R2-BE** 2-Port Isolated RS-232 mPCIe, DB9
- **PCM-24D4R4-BE** 4-Port Non-Isolated RS-422/485 mPCIe, DB37
- **PCM-24R1TP-BE** 1-Port Gigabit Ethernet, mPCIe, RJ45
- **PCM-27D24DI-AE** 24-Channel Isolated Digital I/O w/ counter mPCIe, DB37

Please contact Advantech for all compatible iDoor lists

Optional Accessories

- **96PSA-A150W24T2-4** Power Adapter 150W
- **96PSA-A120W24T2-4** Power Adapter 120W
- **1702002600** Power Cable US Plug 1.8 M (Industrial Grade)
- **UNO-REPWR-AE** Remote power & reset kit
- **1702002605** Power Cable EU Plug 1.8 M (Industrial Grade)
- **1702031801** Power Cable UK Plug 1.8 M (Industrial Grade)
- **1700000596** Power Cable China/Australia Plug 1.8 M (Industrial Grade)
- **UNO-2000G-VMKAE** UNO-2000 VESA Mount Kit
- **UNO-2000G-DMKAE** UNO-2484G DIN RAIL Kit
- **UNO-2000-LKAE** Cable lockable kit 10 units/per pack
- **UNO-2484G-VEGA330** UNO-2484G 1st stack bottom plate for VEGA-330

Operating System

UNO-2484G-6 series:

- **20704WE7PS0002** WS7P X64 MUI V4.19 B038 image
- **20704WX1HS0002** WIN10 LTSC V6.08 for i7 version (High End)
- **20704WX1VS0002** WIN10 LTSC V6.08 for i5/i3 version (Value)
- **20703LEB1S0113** Image AdvLinuxTu V2.1.3 (Ubuntu 18.04)
- **20704U20DS0005** Image AdvLinuxTu V3.2.2 (Ubuntu 20.04)

UNO-2484G-7 series:

- **20704WX1HS0003** WIN10 LTSC V6.08 for i7 version (High End)
- **20704WX1VS0003** WIN10 LTSC V6.08 for i5/i3 version (Value)
- **20703LEB1S0113** Image AdvLinuxTu V2.1.3(Ubuntu 18.04)
- **20704U20DS0005** Image AdvLinuxTu V3.2.2(Ubuntu 20.04)

Please note: If some optional modules are offered with the system, additional system certificates may be required in certain regions/countries.
Please contact Advantech for certificate compliance.

Stackable Extension Kits

UNO-2484G-EKBE

Second stack module for supporting iDoor (3 x mPCIe slot) on UNO-2484G-xx31AE

Specifications

General	
Ports	4 x iDOOR (3 x mPCIe only)
Dimensions	200 x 140 x 40 mm
Weight	400g
Environment	
Operating Temperature	-20 ~ 60 °C

Ordering Information

P/N	Description
UNO-2484G-EKBE	UNO-2484G 2nd stack expansion module for 4 iDoor

UNO-2484G-S2AE

Second stack module for supporting iDoor technology on UNO-2484G-xx31AE

Specifications

General	
Ports	2 x external swappable HDD/SSD Storage
Dimensions	200 x 140 x 40 mm
Weight	400g
Environment	
Operating Temperature	-20 ~ 60 °C

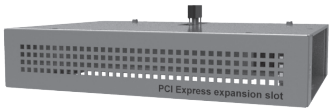
Ordering Information

P/N	Description
UNO-2484G-S2AE	UNO-2484G external swappable HDD extension kit

UNO-2484G-PCIEAE

Second stack module for supporting 1 PCIe x4 card on UNO-2484G-xx31AE

Front view



Side View

Specifications

General	
Ports	1 xPCIex4 slot for half-length PCIe card (W167 x H111 mm)
Dimensions	200 x 140 x 40 mm
Weight	400g
Power consumption	15W (Max)
Environment	
Operating Temperature	0 ~ 50 °C (Depends on expansion card)

Ordering Information

P/N	Description
UNO-2484G-PCIEAE	UNO-2484G second stack PCIex4 expansion module

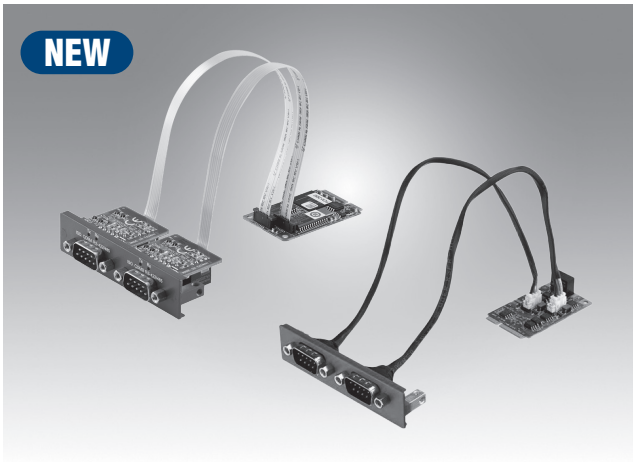
APÉNDICE G

Documentación módulo PDM-26D2CA

Se incluye el datasheet del módulo PDM-26D2CA, instalado en el ordenador para permitir la comunicación mediante bus CAN. Este documento, proporcionado por el fabricante, detalla características técnicas, especificaciones y requisitos de instalación y compatibilidad. Fuente: [26]

PCM-26D2CA

2-Port Isolated CANBus mPCIe, DB9



Features

- Meets Advantech iDoor technology standard
- PCI Express® Mini card specification revision 1.2 compliant
- Operates two separated CAN networks simultaneously
- High speed transmission up to 1 Mbps
- I/O address automatically assigned by PCIe plug & play
- Supports Win10, Linux Ubuntu 18.04/20.04
- Optical isolation protection of 2,500 V_{DC} ensures system reliability
- Includes Windows® DLL library and examples

Introduction

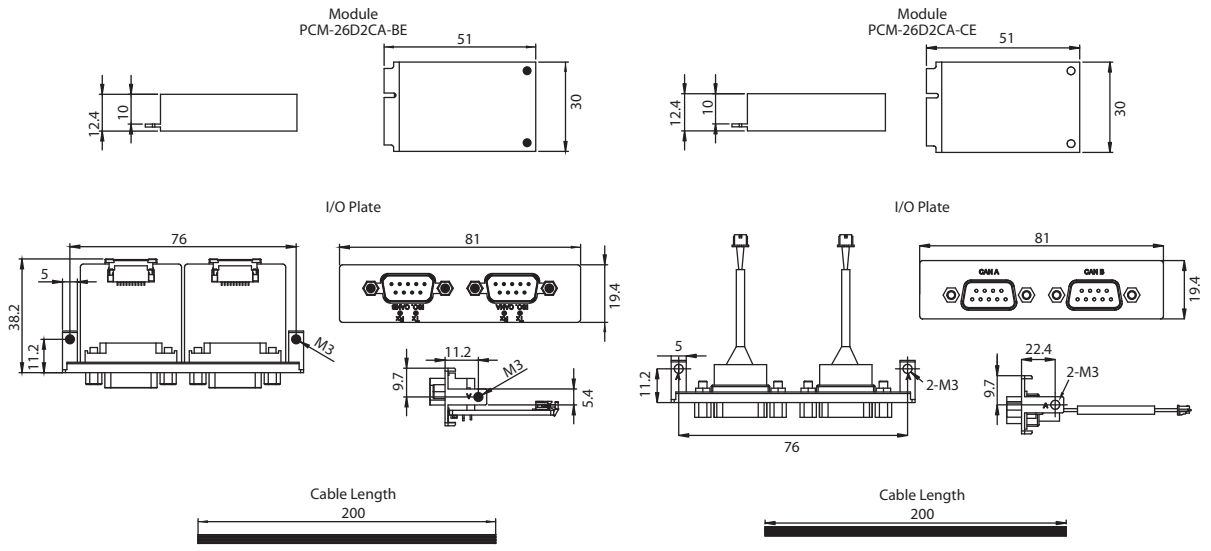
iDoor utilizing mini PCIe format allowing customers the flexibility for I/O and function expansion. The iDoor modules include expansion of wireless communication, industrial fieldbus, I/O and peripherals.

Specifications

		PCM-26D2CA-BE	PCM-26D2CA-CE
General	Bus Type	PCI Express Mini Card Revision 1.2	USB 2.0
	Certification	CE, FCC class A	
	Connectors	2 x Male DB9	
	Dimensions	Module: 51 x 30 x 12.4 mm (2" x 1.18" x 0.49")	
	Power Consumption	Typical : +5V @ 400 mA	
Communications	CAN Controller	NXP SJA-1000	F81604
	CAN Transceiver	NXP_TJA1051T	NXP_TJA1051T
	Protocol	CAN 2.0 A/B	
	Signal Support	CAN_H, CAN_L	
	Speed	1Mbps	500Kbps
	CAN Frequency	16MHz	24MHz
	Termination Resistor	120 Ohm (selected by jumper)	
Protection	Isolation Protection	2,500 V	
	ESD Protection	15 KV	
	EFT Protection	1KV	1KV
	Surge Protection	1,000 V_{DC}	1,000 V_{DC}
Software	CAN Bus Driver	Win10, Linux Ubuntu 18.04/20.04/24.04	
Environment	Humidity (Operating)	5-95% RH, non-condensing	
	Operating Temperature	-20 ~ 60°C (-4 ~ 140°F)	
	Storage Temperature	-40 ~ 85°C (-40 ~ 185°F)	

Dimensions

Unit: mm



Ordering Information

- **PCM-26D2CA-BE** SJA1000 CANBus, DB9 x 2
- **PCM-26D2CA-CE** USB CANBus, DB9 x 2

APÉNDICE H

Códigos de error del Scanreco G3B

Se presenta un listado de códigos de error que pueden aparecer en el display del Scanreco G3B, acompañados de su significado y de posibles opciones de solución. Fuente: [39]

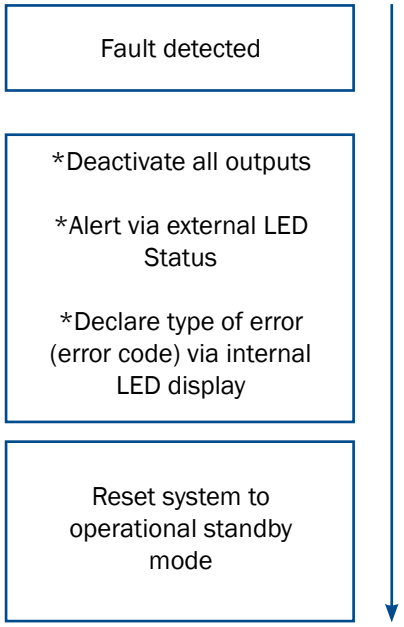
Error code indications

General description

Both the Portable Control Unit and the Central Unit are imbedded with constant fault monitoring, any errors noticed by the system will result in interruption of all operational commands.

Central Unit Error codes

All of the Central Units outputs are fault monitored for short circuits and/or overloads, in the event of an error is detected the Central Unit will alert that an error has occurred via the external LED Status and indicate the appropriate error code via the internal LED Display, the Central Unit will then reset to standby mode awaiting operator action.
Example flow chart on behaviour:



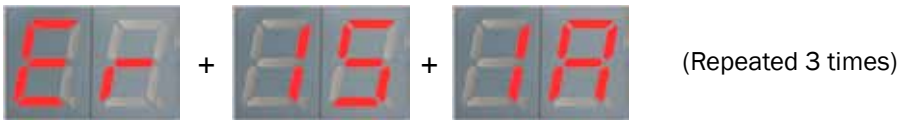
This fault sequence will take an approximately 6 seconds
The external LED STATUS will flash rapidly in red colour declaring that an error has been detected, for more detailed information; the LED display must be monitored (or error code log available from diagnostics menu).

The internal LED display Error codes are displayed in up to 3 sequences, this allows the Central Unit to declare exactly which output that is related to the error (where applicable).

First sequence: Letters E:r is presented declaring an error code

Second sequence: Type of error code

Third sequence: Related output (where applicable)
In example:

 (Repeated 3 times)

The example would imply that there is an short circuit on output 1A

Error codes

2nd	3rd	Description	Cause	Action
01.	01	EEPROM failure.	Incorrect checksum on EEPROM, last stored data will be set.	Reset system, if persistent; Re-load application program.
01.	02	Flash memory failure.	Incorrect checksum on flash memory.	Reset system, if persistent; Re-load application program.
01.	03	Stack memory failure.	Incorrect sizes of data in CANopen protocol, incorrect dataflow or stack overflow.	System will self reset automatically. If persistent; Re-load application program.
01.	04	RAM memory failure.	Incorret RAM and/or hardware identification.	System will self reset automatically. If persistent; Re-load application program.
02.	01	Illegal voltage; DVoutput.	DV-output error; DV-output (DV+) externally supplied	System will self reset. Check DV-output connection. Remove terminal connector and reset system.
02.	02	Short circuit; DV-output.	DV-output error; DV output (DV+) short circuited or overloaded.	System will self reset. Check DV-output connection. Remove terminal connector and reset system.
02.	03	Safety switch error	Safety switch output read back error, incorrect voltage (High instead of Low)	System will self reset. Remove all terminal connectors and reset system.
02.	04	Safety switch error	Safety switch output read back error, incorrect voltage (Low instead of High)	System will self reset. Remove all terminal connectors and reset system.
02.	05	CAN Safety loop error	Incorrect status of CAN safety loop	System will self reset. Check CAN safety loop connection. Reset system.
03.	00	Illegal voltage; Digital output	Digital output (1-14) illegal voltage, expected low signal but read as high (Could be any of the available 14	System will self reset. Check digital output connections. Remove terminal connector and reset

2nd	3rd	Description	0utputs	system
04.	00	Short circuit; Digital output	Digital output (1-14) short circuited or overloaded (Could be any of the available 14 outputs)	System will self reset. Check digital output connections. Remove terminal connector and reset system.
05.	00	Error input triggered (Danfoss CU only).	Error signal for Danfoss valve triggered (Could be any of the available 8 inputs)	System will self reset. Check analogue output connections. Remove terminal connector and reset system.
06.	x	Illegal voltage analogue output	Wrong voltage on analogue output (3rd sequence declares related output; 1A,1B....).	System will self reset. Check analogue output connections. Remove terminal connector and reset system.
07.	x	Illegal voltage analogue output	Wrong current on analogue output (3rd sequence declares related output; 1A,1B....).	System will self reset. Check connections. Remove terminal connector and reset system.
08.	01	CAN Passive	CAN bus in passive mode.	System will self reset. Check CAN connections. Check other nodes on bus and reset system.
08.	02	CAN I/O Buffer overflow	CAN overrun; either the CAN input or CAN output buffer are full	System will self reset. Reset system, re-initiate via CAN controller.
08.	03	CAN physical layer error	Bad communication/transmission	System will self reset. Check CAN connections. Check other nodes on bus and reset system.
08.	04	CAN PDO length exceeded	PDO length is to long	System will self reset. Reset system, re-initiate via CAN controller.
08.	05	CAN PDO length error	PDO length is too short	System will self reset. Reset system, re-initiate via CAN controller.
08.	06	CAN Transmit COB-ID collision	To many collisions on CANbus	System will self reset. Check CAN connections. Check other nodes on bus and reset system, re-initiate via CAN controller.
10.	n/a	PCU failure; Emergency stop	Error transmitted from PCU: Illegal signal from PCU emergency stop switch	System will self reset. Check emergency stop switch on PCU
11.	n/a	PCU failure; Analogue input	Error transmitted from PCU: Analogue input active on start-up	System will self reset. Ensure all analogue inputs on PCU are at zero/neutral position. Restart PCU.
13.	n/a	PCU failure; Analogue input	Error transmitted from PCU: Signal redundancy test; illegal signal from analogue input.	System will self reset; Diagnose PCU via TEST MODE

14.	01	IDprogramming failure	ID-code and/or parameter settings not accepted.	System will self reset. Verify ID-programming procedure. Reset application program.
14.	02	Program failure	Programmable logic parameter error	System will self reset. Reset application program.
15.	x	PWM output failure	Analogue output short circuited or overloaded. (3rd sequence declares related output; 1A,1B....).	System will self reset. Check analogue output connections. Remove terminal connector and reset system.
16.	x	PWM output failure	Analogue output not connected (Programmable feature). (3rd sequence declares related output; 1A,1B....).	System will self reset. Check analogue output connections. Remove terminal connector and reset system.
17.	01	Low supply power	Low power supply (Below 8,5 VDC)	System will self reset. Check power supply and supply connections.
17.	02	High supply power	High power supply (Above 36,0 VDC)	System will self reset. Check power supply and supply connections.
98.	n/a	Undefined PCU error	Undefined error in PCU.	Diagnose PCU via TEST MODE
99.	n/a	Undefined CU error	Undefined error in CU.	System will self reset. Remove all terminal connectors Check power supply and supply connections. Reset system.

Portable Control Unit error codes

The Portable Control Unit monitors all analogue and digital inputs for faults and uses the LED POWER and BUZZER to indicate alarms.

Below available error codes:

Indications	Meaning
1	Analogue input 1 not at zero position during start-up
2	Analogue input 2 not at zero position during start-up
3	Analogue input 3 not at zero position during start-up
4	Analogue input 4 not at zero position during start-up
5	Analogue input 5 not at zero position during start-up
6	Analogue input 6 not at zero position during start-up
7	Analogue input 7 not at zero position during start-up
8	Analogue input 8 not at zero position during start-up