



Universidad de Málaga

Escuela de Ingenierías Industriales

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Trabajo Fin de Grado

**Desarrollo de un paquete ROS de alto nivel
para sistemas robóticos actuados por motores
inteligentes**

Grado en Ingeniería Electrónica Industrial

Autor: Maksym Saldat

Tutor: Juan Manuel Gandarias Palacios
Cotutor: Jesús Manuel Gómez de Gabriel

30 de noviembre de 2024

Resumen

Desarrollo de un paquete ROS de alto nivel para sistemas robóticos actuados por motores inteligentes

Autor: Maksym Saldat

Tutor: Juan Manuel Gandarias Palacios

Cotutor: Jesús Manuel Gómez de Gabriel

Departamento: DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Titulación: Grado en Ingeniería Electrónica Industrial

Palabras clave librería; programación; ROS, robótica, actuadores

El presente Trabajo de Fin de Grado (TFG) tiene como objetivo el desarrollo de un paquete de alto nivel de Robot Operating System (ROS), programado en C++, que facilite la integración y el manejo de los servomotores inteligentes de la marca Dynamixel. La motivación principal de este proyecto surge de las limitaciones y problemáticas existentes en las herramientas actuales para controlar estos motores, las cuales requieren un conocimiento profundo sobre su construcción y funcionamiento.

En la actualidad, las soluciones disponibles para trabajar con servomotores Dynamixel incluyen principalmente librerías de bajo nivel que permiten únicamente la escritura y lectura de los registros del motor, y sistemas de control basados en plataformas como Arduino. Estas herramientas, aunque funcionales, presentan una curva de aprendizaje muy elevada, lo que dificulta su uso por parte de usuarios no expertos en la materia. Además, estas soluciones no proporcionan una integración sencilla con ROS, que es una plataforma ampliamente utilizada en robótica para la gestión y control de sistemas complejos.

El paquete desarrollado en este TFG busca abordar estas carencias proporcionando una interfaz de alto nivel que permita a los usuarios controlar y gestionar los servomotores Dynamixel de manera eficiente y sin necesidad de conocimientos avanzados sobre su funcionamiento interno. Este paquete no solo facilitará la programación y control de los motores, sino que también mejorará la integración con ROS, permitiendo a los desarrolladores centrarse en las aplicaciones finales en lugar de en los detalles técnicos del hardware.

Abstract

Development of a High-Level ROS Package for Robotic Systems Actuated by Smart Motors

Author: Maksym Saldat

Supervisor: Juan Manuel Gandarias Palacios

Cosupervisor: Jesús Manuel Gómez de Gabriel

Departament: DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Degree: Grado en Ingeniería Electrónica Industrial

Keywords: librería; programación; ROS, robótica, actuadores

The present Bachelor's Thesis aims to develop a high-level package for the Robot Operating System (ROS), programmed in C++, to facilitate the integration and management of intelligent Dynamixel servomotors. The main motivation for this project stems from the limitations and issues with existing tools for controlling these motors, which typically require deep knowledge of their construction and operation.

Currently, available solutions for working with Dynamixel servomotors primarily include low-level libraries that allow only for writing and reading motor registers, and control systems based on platforms like Arduino. While functional, these tools have a steep learning curve, making them challenging for non-expert users. Moreover, these solutions do not offer seamless integration with ROS, which is widely used in robotics for managing and controlling complex systems.

The package developed in this thesis addresses these shortcomings by providing a high-level interface that enables users to efficiently control and manage Dynamixel servomotors without requiring advanced knowledge of their internal workings. This package will not only simplify motor programming and control but also enhance integration with ROS, allowing developers to focus on final applications rather than the technical details of the hardware.

Dedicatoria o cita

"No se trata de si la guerra es real o no, la victoria no es posible. No se trata de ganar la guerra, sino de que esta sea constante."

George Orwell, "1984"

Agradecimientos

Quisiera destacar que, además del esfuerzo requerido para un trabajo como este, es necesario estar rodeado de personas que te impulsen a ser cada vez mejor y que te animen a enfrentar los problemas que, a menudo, preferimos evitar.

Teniendo la oportunidad para ello, quiero expresar mi más profundo agradecimiento a mi familia, cuyo apoyo incondicional me ha permitido alcanzar logros que hace nueve años hubieran sido impensables. Ellos me han enseñado a ver la vida desde una perspectiva diferente y me han acompañado en el día a día. Gracias, Andriy, Mariya y Diana.

Asimismo, quiero expresar mi gratitud a mi novia, Adriana Alcarazo Aguilar, quien, cada día y a cada hora, me ha motivado a ser una mejor persona, un mejor estudiante y un mejor profesional. Parte del mérito de este logro también te pertenece.

Por supuesto, no puedo no mencionar a mi tutor de este proyecto, Juan Manuel Gandarias Palacios. Es un profesional en toda regla, además de ser una persona responsable, organizada y resolutiva. Me ha acompañado durante todo este Trabajo de Fin de Grado, ayudándome a ver los problemas desde distintas perspectivas y mostrándome diferentes caminos para alcanzar los mismos objetivos.

Acrónimos y Notación Matemática

bps	Bits por segundo
SDK	Source development kit (Kit de desarrollo de software)
SO	Sistema operativo
CC	Corriente continua
CA	Corriente alterna
POO	Programación orientada a objetos
ROS	Robot Operating System (Sistema operativa para robots)
GUI	Graphic User Interface (Interfaz gráfica de usuario)

Índice

	Página
Índice de Figuras	xv
1 Introducción	1
1.1. Motivación	2
1.2. Estado del arte sobre actuadores robóticos	3
1.2.1. Motores de Corriente Continua (CC)	3
1.2.2. Motores Paso a Paso	5
1.2.3. Servomotores	5
1.2.4. Actuadores Inteligentes	5
1.3. Objetivos y contribución	7
1.3.1. Requisitos de la librería	7
1.4. Estructura de la memoria	8
2 Contexto y Fundamentos Teóricos del Proyecto	9
2.1. Herramientas de manejo Dynamixel	10
2.1.1. Dynamixel SDK	10
2.1.2. Dynamixel Workbench	10
2.1.3. Dynamixel Wizzard 2.0	11
2.1.4. Problemáticas encontradas	12
2.2. Funcionamiento de ROS	14
2.2.1. Arquitectura	14
2.2.2. Comunicaciones	15
2.2.3. Herramientas de ROS	16
2.2.4. Paquetes y Dependencias	16
2.2.5. Tipos de Datos en ROS	16
2.2.6. Uso de ROS con los Actuadores Dynamixel	17
2.3. Programación orientada a objetos (POO)	17
2.3.1. Conceptos Fundamentales de la POO	17
2.3.2. Ventajas de la POO	18

ÍNDICE

2.3.3. Particularización de POO para la programación de los motores Dynamixel	19
3 Metodología	21
3.1. Elección de C++ y ROS	22
3.2. Dependencias	22
3.3. Estructura	23
3.3.1. Tablas de Control	23
3.3.2. Campos de clase	23
3.3.3. Atributos estáticos de dynamixelMotor	24
3.3.4. Métodos estáticos de dynamixelMotor	26
3.3.5. Métodos de objeto dynamixelMotor	26
4 Experimentos y Resultados	65
4.1. Entorno experimental	66
4.1.1. Setup de Hardware	66
4.1.2. Configuración del PC	69
4.2. Protocolo de experimentación	70
4.2.1. Objetivo de los Experimentos	70
4.2.2. Preparación del Entorno	71
4.2.3. Procedimiento Experimental	71
4.2.4. Recopilación y Análisis de Datos	72
4.2.5. Criterios de Éxito	72
4.3. Experimento 1: actuadores aislados	73
4.3.1. Contexto	73
4.3.2. Ejecución	73
4.3.3. Resultado	75
4.4. Experimento 2: sistema completo	78
4.4.1. Contexto	78
4.4.2. Ejecución	79
4.4.3. Resultado	81
4.5. Discusión de los resultados	82
5 Conclusiones	85
Bibliografía	87

Índice de Figuras

Figura	Página
1.1. Stock operativo de robots industriales a nivel mundial en el periodo 2012-2022 (<i>Fuente: ifr.org</i>)	2
1.2. Estructura interna de un motor CC (<i>Fuente: IQSDirectory</i>)	3
1.3. Aspecto de un servomotor (<i>Fuente: GrupoGAES</i>)	5
1.4. Variedad de motores Dynamixel Series X (<i>Fuente: Robotis</i>)	7
2.1. Dispositivos, SO y lenguajes de programación compatibles con Dynamixel SDK (<i>Fuente: Robotis</i>)	11
2.2. Interfaz gráfica de Dynamixel Wizzard 2.0 (<i>Fuente: Robotis</i>)	11
2.3. Relación Valor-bps para los modelos Dynamixel XL330-M077 (a) y Dynamixel XM430-T333 (b).(<i>Fuente: Robotis</i>)	12
2.4. Un fragmento de la tabla de control para Dynamixel XL330-M077 (<i>Fuente: Robotis</i>)	13
2.5. Representación gráfica de mensajería a través de topics en ROS (<i>Fuente: robinrobotics.blogspot.com</i>)	15
2.6. Principios de encapsulamiento. Funcionamiento de los modificadores de acceso (<i>Fuente: grupocodesi.com</i>)	18
2.7. Tipos de herencia (<i>Fuente: ifgeekthen.com</i>)	19
4.1. Conexionado Dynamixel - PC (<i>Fuente: robotis.com</i>)	66
4.2. Controlador U2D2 (<i>Fuente: robotis.com</i>)	67
4.3. U2D2 Power Hub (<i>Fuente: robotis.com</i>)	68
4.4. Conexiones disponibles en U2D2 Power Hub (<i>Fuente: robotis.com</i>)	68
4.5. Conexionado de motores al U2D2 Power Hub (<i>Fuente: robotis.com</i>)	69
4.6. Tipos de cables Dynamixel (<i>Fuente: robotis.com</i>)	69
4.7. Pines de los cables Dynamixel (<i>Fuente: robotis.com</i>)	70
4.8. Interfaz gráfica de PlotJuggler (<i>Fuente: blog.csdn.net</i>)	72
4.9. Motor utilizado durante el experimento	74
4.10. Lanzamiento de los nodos necesarios en la terminal	74
4.11. Respuesta para la orden de 0 grados	75

ÍNDICE DE FIGURAS

4.12. Posición del motor después de recibir una posición objetivo de 0 grados	75
4.13. Respuesta para la orden de 90 grados	76
4.14. Respuesta para la orden de 270 grados	76
4.15. Posición del motor después de recibir una posición objetivo de 90 grados	77
4.16. Posición del motor después de recibir una posición objetivo de 270 grados	77
4.17. Vista frontal del sistema utilizado en el experimento	78
4.18. Vista lateral del sistema utilizado en el experimento	78
4.19. Vista desde arriba del sistema utilizado en el experimento	79
4.20. Mensaje de éxito tras la correcta inicialización	80
4.21. La publicación del identificador de la posición predefinida “1”	80
4.22. Respuesta de la librería tras recibir la orden correspondiente a la posición predefinida “1”	80
4.23. Variación de posición y velocidad del motor al recibir una referencia nueva	81

Introducción

Contenido

1.1.	Motivación	2
1.2.	Estado del arte sobre actuadores robóticos	3
1.2.1.	Motores de Corriente Continua (CC)	3
1.2.2.	Motores Paso a Paso	5
1.2.3.	Servomotores	5
1.2.4.	Actuadores Inteligentes	5
1.3.	Objetivos y contribución	7
1.3.1.	Requisitos de la librería	7
1.4.	Estructura de la memoria	8

En el capítulo de introducción, el lector encontrará una descripción detallada del contexto y los motivos que llevaron al desarrollo de este proyecto. Se explica cómo los avances en tecnología y robótica han creado nuevas exigencias y desafíos, y se destaca la necesidad de herramientas que simplifiquen la complejidad inherente de los sistemas robóticos, especialmente en el manejo de actuadores inteligentes como los Dynamixel. Además, se presenta el objetivo principal del trabajo: desarrollar una biblioteca que facilite la configuración y el control de estos motores, automatizando tareas complejas y reduciendo la necesidad de conocimientos especializados. La introducción también incluye un resumen de la estructura del documento, proporcionando una guía clara sobre los contenidos que se abordarán en los capítulos posteriores, incluyendo la motivación, el estado del arte, los objetivos y la contribución del proyecto, así como la estructura de la memoria.

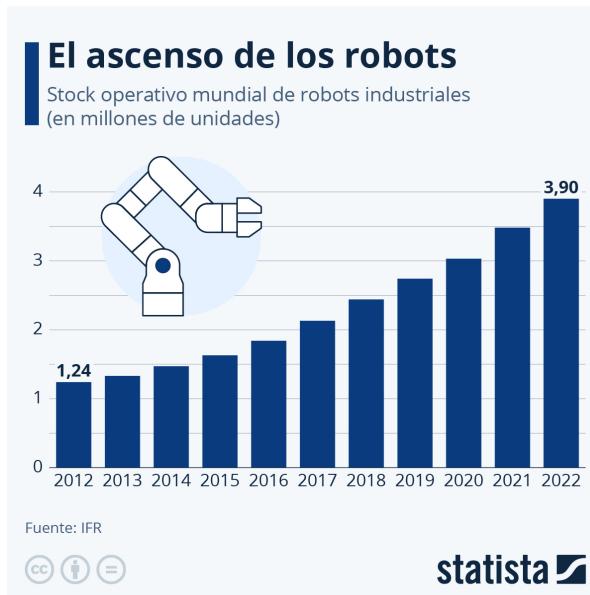


Figura 1.1: Stock operativo de robots industriales a nivel mundial en el periodo 2012-2022 (*Fuente: ifr.org*)

1.1. Motivación

Incluso un análisis superficial de la historia humana en cuanto al desarrollo tecnológico permite ver que cada avance implica nuevas exigencias y desafíos. Los problemas, al ser resueltos, dejan de ser problemas, pero se transforman en el tronco de un árbol de problemas más complejos y avanzados. Esta dinámica es particularmente evidente en el campo de la ingeniería moderna, donde la robótica ha experimentado un crecimiento exponencial en las últimas décadas [1]. Según las últimas investigaciones [2] realizadas por IFR (International Federation of Robotics), el número de robots industriales operativos a nivel mundial se ha triplicado en el periodo desde 2012 hasta 2022 (ver Figura 1.1). Este crecimiento tan pronunciado, entre otras cosas, ha elevado significativamente el nivel de exigencia hacia los sistemas robóticos.

Para seguir manteniendo esta pendiente ascendente en la robótica, los fabricantes y desarrolladores se enfocan cada vez más en la eficiencia y la accesibilidad de los sistemas robóticos [3]. Este enfoque busca simplificar la tecnología, haciendo que las tareas complejas sean segmentadas en unidades más manejables y transparentes para el usuario final. De este modo, se facilita el trabajo no solo de los técnicos altamente especializados, sino también de aquellos usuarios con menos formación técnica, ampliando así el alcance y la aplicabilidad de las soluciones robóticas.

La simplificación de las tareas complejas en unidades más accesibles implica desarrollar interfaces y herramientas que oculten la complejidad subyacente del sistema, permitiendo a los usuarios interactuar con la tecnología de manera más intuitiva. Este proceso no solo mejora la eficiencia operativa, sino que también reduce la curva de aprendizaje y el tiempo necesario para dominar nuevas tecnologías.

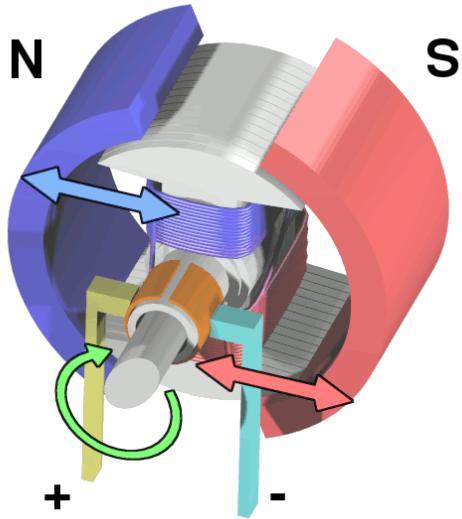


Figura 1.2: Estructura interna de un motor CC (*Fuente: IQSDirectory*)

Uno de los elementos clave a simplificar dentro de un sistema robótico es el actuador principal: el motor. Un claro ejemplo de esta tendencia es la integración de controladores lógicos dentro de los motores, lo que permite simplificar tareas como la medición de parámetros y la monitorización de variables clave [4]. Esta integración facilita el manejo y control de los motores, haciendo que los sistemas robóticos sean más accesibles y eficientes.

1.2. Estado del arte sobre actuadores robóticos

La mayoría de sistemas robóticos utilizan actuadores eléctricos. Aunque también hay robots que utilizan sistemas neumáticos [5] o hidráulicos [6], la tendencia actual es el uso de motores eléctricos. Ésto se debe a diferentes factores, como el abaratamiento de su precio en los últimos años o el aumento de la relación par/peso, entre otros [7].

1.2.1. Motores de Corriente Continua (CC)

Funcionan a través de la interacción de un campo magnético creado por el rotor y el estator (ver Figura 1.2). La velocidad del motor CC se puede controlar normalmente variando el voltaje suministrado, mientras que la polaridad del voltaje aplicado es determinante a la hora de establecer el sentido de giro.

Los motores CC se utilizan con mucha frecuencia en robots móviles, donde se requiere un control preciso de la velocidad para la navegación.

1.2.1.1. Motores con Escobillas

Los motores con escobillas son el tipo más común de motores de corriente continua. Funcionan mediante el uso de escobillas y un conmutador que cambia la dirección de la corriente en el rotor, manteniéndolo en movimiento. Las características principales de los motores con escobillas incluyen:

- **Simplicidad:** La construcción de estos motores es relativamente simple, lo que facilita su fabricación y mantenimiento.
- **Control preciso:** Permiten un control preciso de la velocidad y el torque, lo que los hace ideales para aplicaciones donde se requiere un control fino.
- **Desgaste de Escobillas:** Uno de los inconvenientes es el desgaste de las escobillas con el tiempo, lo que requiere mantenimiento periódico para reemplazarlas.

Estos motores se utilizan en una variedad de aplicaciones, incluyendo electrodomésticos, juguetes y herramientas eléctricas [8] [9].

En el campo de la robótica, estos motores son de uso común en campos como robótica móvil (drones y vehículos autónomos), industrial (manipuladores) y médica (equipos quirúrgicos) [10] [11] [12].

1.2.1.2. Motores sin Escobillas

Los motores sin escobillas, también conocidos como motores brushless, eliminan la necesidad de escobillas y conmutadores. En lugar de eso, utilizan un controlador electrónico para conmutar la dirección de la corriente en el rotor. Las características principales de los motores sin escobillas incluyen:

- **Mayor Eficiencia:** Al no tener escobillas, se reduce la fricción y el desgaste, lo que aumenta la eficiencia del motor.
- **Menor Mantenimiento:** La ausencia de escobillas elimina la necesidad de reemplazarlas, reduciendo los costos y el tiempo de mantenimiento.
- **Mayor Durabilidad:** Estos motores suelen tener una vida útil más larga debido a la falta de desgaste mecánico en las escobillas.

Los motores sin escobillas se utilizan en aplicaciones que requieren alta eficiencia, durabilidad y un rendimiento confiable, como en drones, vehículos eléctricos y equipos industriales [13] [14].



Figura 1.3: Aspecto de un servomotor (*Fuente: GrupoGAES*)

1.2.2. Motores Paso a Paso

Los motores paso a paso dividen una rotación completa en un número igual de pasos, lo que permite un control preciso del ángulo de rotación. Estos motores son ideales para aplicaciones que requieren un posicionamiento exacto, por lo que se utilizan de forma extendida en impresoras 3D y en sistemas de control numérico por computadora (CNC) [15].

1.2.3. Servomotores

Los servomotores, cuyo ejemplo se muestra en la Figura 1.3, son motores CC o CA con un sistema de control incorporado, que permite un control preciso de la posición angular. Incluyen un sensor que mide el ángulo de giro que permite realimentar la posición actual del motor.

Los servomotores se utilizan mayoritariamente en brazos robóticos para lograr un control preciso del movimiento y la posición [16].

1.2.4. Actuadores Inteligentes

En realidad, los actuadores inteligentes, tal como se conocen de forma genérica, son actuadores que integran una placa de circuito impreso (PCB) o algún circuito integrado, aprovechando las ventajas de la electrónica digital. Esta integración permite realizar el control directamente a bordo del dispositivo. Además, estos actuadores suelen disponer de algún framework o herramienta (normalmente un SDK) que permite a los desarrolladores cambiar parámetros y modos de control para obtener diferentes funcionalidades.

Los actuadores inteligentes combinan las ventajas de los servomotores con capacidades adicionales, como la comunicación digital, sensores avanzados y controladores integrados. Estos motores son capaces de realizar diagnósticos en tiempo real, adaptarse a diferentes condiciones de funcionamiento y comunicarse con otros dispositivos a través de interfaces estándar [16].

El uso de servomotores inteligentes dota el sistema robótico de las siguientes ventajas:

- **Precisión y Control:** Permiten un control preciso de la velocidad, la posición y el par.
- **Comunicación:** Soportan protocolos de comunicación avanzados, lo que facilita su integración en redes industriales y sistemas de control.
- **Retroalimentación:** Proporcionan datos en tiempo real sobre el estado del motor, incluyendo temperatura, corriente y posición.
- **Facilidad de Uso:** Simplifican el diseño y la implementación de sistemas robóticos, reduciendo la necesidad de conocimientos técnicos avanzados.

Una de las aplicaciones más destacadas de los actuadores inteligentes es en los brazos robóticos industriales. En estas aplicaciones, los actuadores inteligentes permiten movimientos precisos y repetibles, esenciales para tareas como la soldadura, el ensamblaje y la pintura. La capacidad de realizar diagnósticos en tiempo real y adaptarse a diferentes condiciones de funcionamiento mejora la fiabilidad y eficiencia de los procesos industriales [14, 16].

En el ámbito de la robótica médica, los actuadores inteligentes son cruciales para los robots quirúrgicos, donde la precisión y la capacidad de respuesta son vitales. Los sensores avanzados y los controladores integrados permiten a estos robots realizar movimientos extremadamente delicados y precisos, lo que es fundamental en procedimientos quirúrgicos [17].

Además, en los robots de servicio, los actuadores inteligentes facilitan la interacción segura y efectiva con los humanos. La capacidad de comunicarse a través de interfaces estándar y ajustar dinámicamente los parámetros de control permite a estos robots adaptarse a entornos cambiantes y realizar una variedad de tareas de manera eficiente [14].

En la educación y la investigación, los actuadores inteligentes permiten a los estudiantes y a los investigadores experimentar con sistemas avanzados de control de movimiento. Los kits de robótica que incluyen estos actuadores proporcionan una plataforma versátil para el desarrollo de nuevas tecnologías y aplicaciones robóticas [18].

1.2.4.1. Actuadores Dynamixel

Los servomotores Dynamixel, desarrollados por Robotis, son un ejemplo destacado de servomotores inteligentes. Estos motores se utilizan ampliamente en aplicaciones robóticas debido a sus características avanzadas y su facilidad de uso [19, 20].

Las características más importantes de estos motores son:

- **Control de precisión:** Ofrecen un control de la posición y velocidad.
- **Comunicación digital:** Utilizan protocolos de comunicación como TTL y RS-485 para la conexión en red y el control centralizado.



Figura 1.4: Variedad de motores Dynamixel Series X (*Fuente: Robotis*)

- **Realimentación en tiempo real:** Proporcionan datos sobre el estado del motor, como la temperatura, el voltaje y la carga.

Los servomotores Dynamixel son populares en la investigación y la industria debido a su flexibilidad y capacidades avanzadas. Se utilizan en una amplia variedad de aplicaciones, desde robots educativos hasta robots industriales complejos [21] [22].

Este trabajo se enfocará específicamente en los motores de la serie “Series X” de “Dynamixel” (Figura 1.4), siendo la serie especificada la más usada por el Departamento de Ingeniería de Sistemas y Automática de la Universidad de Málaga debido a su capacidad de controlar la corriente máxima e incluso el par ejercido por el actuador.

El problema principal que surge a la hora de manejar los actuadores inteligentes Dynamixel es que, aunque se proporciona un SDK para acceder a las funcionalidades del motor, estas son de muy bajo nivel. Esto significa que los desarrolladores no pueden abstraerse del hardware, sino que deben programar a nivel de registro. Para ello, se proporcionan tablas de parámetros y registros, aunque estas pueden variar en función del modelo del actuador.

1.3. Objetivos y contribución

Para abordar estos desafíos, se propone el desarrollo de la biblioteca “dynamixel_ros_library”. Esta biblioteca, implementada en C++ y diseñada específicamente para su integración con ROS (Robot Operating System), tiene como objetivo principal simplificar y estructurar la programación relacionada con los motores Dynamixel. Al proporcionar una interfaz clara y coherente, se busca facilitar el manejo de estos motores en entornos robóticos complejos, abordando así las limitaciones actuales y promoviendo un desarrollo más eficiente y accesible en el campo de la robótica.

1.3.1. Requisitos de la librería

La solución final debe reunir lo siguiente:

- **Funcionalidades:** Se optará por la creación de un paquete de ROS que incluya las funcionalidades necesarias para la configuración y control de motores, tanto para configuraciones simples como para programas complejos donde los parámetros deben ajustarse dinámicamente según variables externas. Para ello es necesario que la librería proporcione la posibilidad de modificar cualquier parámetro que tenga que ver con el motor.
- **Abstacción del hardware:** La librería debe dar al usuario la posibilidad de utilizar los motores sin la necesidad de un conocimiento profundo del producto. La librería debe gestionar internamente las relaciones entre el software desarrollado y el hardware y todas sus implicaciones, como: los rangos de valores permitidos, los errores que puedan ocurrir y la localización de datos en la memoria interna del motor.
- **Feedback:** Para que el usuario pueda verificar si se ha llevado a cabo correctamente una operación, la solución debe implementar un sistema de mensajes que vaya actualizando al usuario sobre el estado actual del motor.

En cuanto a la presentación de la información devuelta, ésta debe estar estructurada en un formato establecido, ser concisa y fácilmente interpretable.

1.4. Estructura de la memoria

La estructura de la memoria está formada por varios capítulos interrelacionados.

En el capítulo Contexto y Marco Teórico², se explora el contexto relevante del proyecto, proporcionando conocimientos teóricos necesarios para comprender en profundidad los conceptos clave de este trabajo. Se abordan temas como el funcionamiento de las herramientas ya existentes, así como otros aspectos técnicos fundamentales.

El capítulo 3, Metodología³, explica las decisiones de diseño, la arquitectura de la solución propuesta, así como las herramientas y métodos empleados para la implementación de la librería *dynamixel_ros_library*.

En el capítulo Experimentos y Resultados⁴, se presentan los experimentos realizados para validar la funcionalidad y eficacia de la librería desarrollada. Este capítulo discute objetivamente cómo la librería demuestra el funcionamiento de las funcionalidades definidas.

Finalmente, el capítulo Conclusiones⁵ ofrece una evaluación crítica del proyecto basada en los objetivos planteados y los resultados obtenidos. En este capítulo se discute la relevancia de la solución propuesta, sus limitaciones y posibles áreas de mejora para trabajos futuros.

Contexto y Fundamentos Teóricos del Proyecto

Contenido

2.1.	Herramientas de manejo Dynamixel	10
2.1.1.	Dynamixel SDK	10
2.1.2.	Dynamixel Workbench	10
2.1.3.	Dynamixel Wizzard 2.0	11
2.1.4.	Problemáticas encontradas	12
2.2.	Funcionamiento de ROS	14
2.2.1.	Arquitectura	14
2.2.2.	Comunicaciones	15
2.2.3.	Herramientas de ROS	16
2.2.4.	Paquetes y Dependencias	16
2.2.5.	Tipos de Datos en ROS	16
2.2.6.	Uso de ROS con los Actuadores Dynamixel	17
2.3.	Programación orientada a objetos (POO)	17
2.3.1.	Conceptos Fundamentales de la POO	17
2.3.2.	Ventajas de la POO	18
2.3.3.	Particularización de POO para la programación de los motores Dynamixel	19

Este capítulo introduce y desarrolla los conceptos y las herramientas fundamentales que se han utilizado en la elaboración de este Trabajo de Fin de Grado. La comprensión de estos conceptos es esencial, ya que constituyen los pilares sobre los cuales se ha desarrollado la librería. Entre los temas tratados se incluyen el Robot Operating System (ROS), la programación orientada a objetos (POO) y el funcionamiento de herramientas específicas como Dynamixel SDK y Dynamixel Workbench.

2.1. Herramientas de manejo Dynamixel

Existen varios paquetes y aplicaciones disponibles para programar sistemas que utilicen motores Dynamixel que, si bien ofrecen algunas funcionalidades básicas, carecen de la robustez y la flexibilidad necesarias para manejar sistemas robóticos complejos de manera eficiente. A continuación se describen los paquetes existentes y sus características principales. Al ser similares en cuanto a desventajas, éstas se han recogido en un apartado independiente.

2.1.1. Dynamixel SDK

SDK(Source Development Kit), que constituye una colección de herramientas de software proporcionada por Robotis para la programación y control de los motores inteligentes Dynamixel. Este SDK está diseñado para ofrecer a los desarrolladores una interfaz para comunicarse con los motores y realizar operaciones básicas como la lectura de datos de sensores y el control del movimiento.

El dynamixel SDK es compatible con múltiples sistemas operativos (SO), incluyendo sistemas populares como Windows, Linux o macOS, lo cual permite a los desarrolladores trabajar en el entorno que sea de su preferencia (ver Figura 2.1).

Asimismo, permite la programación en varios lenguajes de programación extendidos como C++, Python, Java y MATLAB, proporcionando flexibilidad de diseño del software.

2.1.2. Dynamixel Workbench

Es una colección de herramientas y bibliotecas proporcionada también por Robotis, diseñadas para simplificar la configuración, el control y la gestión de los motores Dynamixel. A diferencia del Dynamixel SDK, que está orientado a desarrolladores con ciertos conocimientos de programación, el Dynamixel Workbench ofrece una interfaz más amigable y accesible. A continuación se presentan algunas de las características principales del Dynamixel Workbench:

- **Interfaz de usuario:** El Dynamixel Workbench incluye herramientas que proporcionan una interfaz gráfica de usuario (GUI), lo que permite a los usuarios configurar y controlar los

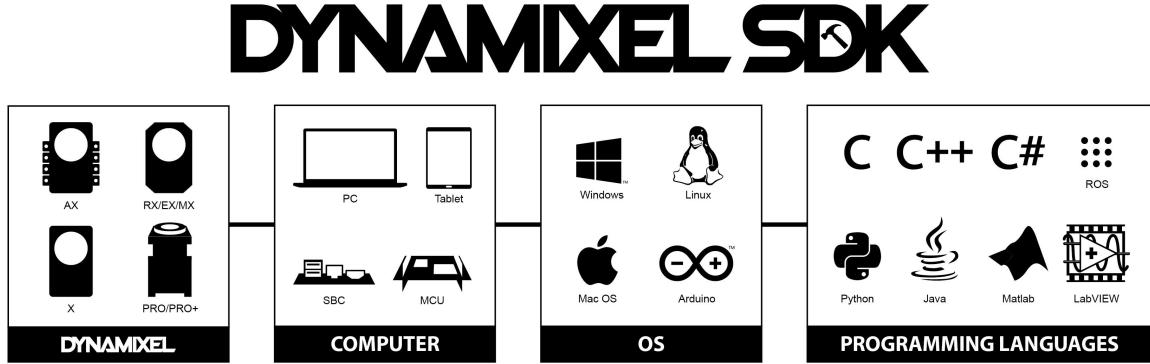


Figura 2.1: Dispositivos, SO y lenguajes de programación compatibles con Dynamixel SDK (*Fuente: Robotis*)

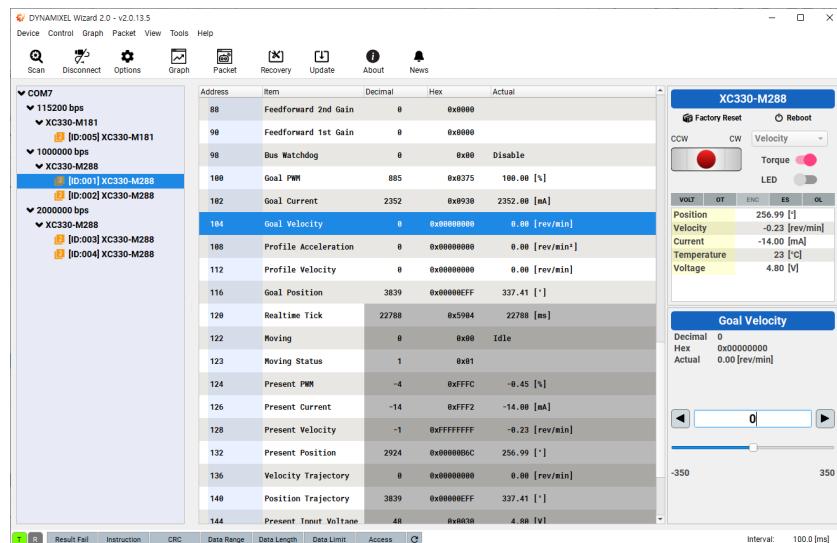


Figura 2.2: Interfaz gráfica de Dynamixel Wizzard 2.0 (*Fuente: Robotis*)

motores sin necesidad de escribir código. Esto es útil para tareas de configuración rápida y pruebas.

- **Funciones de configuración y diagnóstico:** La herramienta permite a los usuarios configurar parámetros del motor, realizar diagnósticos y monitorizar el estado del motor en tiempo real.

2.1.3. Dynamixel Wizard 2.0

Dynamixel Wizard 2.0 es una aplicación desarrollada por Dynamixel que permite interactuar con los motores a través de una interfaz gráfica (mostrada en la Figura 2.2) de usuario. Aunque es útil para tareas de configuración básica, su uso puede ser limitado para aplicaciones más avanzadas que requieran programación y control específico del comportamiento de los motores.

2.1.4. Problemáticas encontradas

A continuación se describen los principales desafíos y problemáticas observados:

- Falta de abstracción del hardware:** La necesidad de comprender en detalle las características del hardware, ya que no todos los parámetros de los modelos distintos se encuentran en el mismo registro, se modifican de la misma forma y sus valores digitales tienen la misma interpretación física. Un claro ejemplo de esto podría ser el registro que contiene la velocidad de comunicación. Es un registro de 8 bits, pero no siempre el valor hexadecimal 0xFF representará la velocidad máxima de un modelo de motor concreto, si no que puede variar.

A continuación se muestran las diferencias en tablas de conversión que relacionan el valor digital escrito en el registro que regula la velocidad de comunicación entre el controlador y el motor y su interpretación en bps. Se han escogido como modelos el XL330-M077¹ y el modelo XW430-T333², cuyas tablas se presentan en la Figuras 2.3 (a) y (b), respectivamente.

Value	Baud Rate	Margin of Error	Value	Baud Rate	Margin of Error
6	4M [bps]	0.000 [%]	7	4.5M [bps]	0.000 [%]
5	3M [bps]	0.000 [%]	6	4M [bps]	0.000 [%]
4	2M [bps]	0.000 [%]	5	3M [bps]	0.000 [%]
3	1M [bps]	0.000 [%]	4	2M [bps]	0.000 [%]
2	115,200 [bps]	0.0064 [%]	3	1M [bps]	0.000 [%]
1(Default)	57,600 [bps]	0.0016 [%]	2	115,200 [bps]	0.000 [%]
0	9,600 [bps]	0.000 [%]	1(Default)	57,600 [bps]	0.000 [%]
			0	9,600 [bps]	0.000 [%]

(a)

(b)

Figura 2.3: Relación Valor-bps para los modelos Dynamixel XL330-M077 (a) y Dynamixel XM430-T333 (b). (*Fuente: Robotis*)

- Localización de datos:** Es necesario conocer la ubicación de la memoria interna en la que se almacenan parámetros relacionados con el motor. El fabricante reúne los datos en las denominadas “tablas de control” (ejemplo mostrado en la Figura 2.4), que vinculan una dirección en la memoria con un registro como puede ser el registro de la velocidad actual, límites de temperatura, etc.

¹Hoja de características del modelo XL330-M077

²Hoja de características del modelo XW430-T333

Address	Size(Byte)	Data Name	Access	Initial Value	Range	Unit
0	2	Model Number	R	1,190	-	-
2	4	Model Information	R	-	-	-
6	1	Firmware Version	R	-	-	-
7	1	ID	RW	1	0 ~ 252	-
8	1	Baud Rate	RW	1	0 ~ 6	-
9	1	Return Delay Time	RW	250	0 ~ 254	2 [μsec]
10	1	Drive Mode	RW	0	0 ~ 5	-
11	1	Operating Mode	RW	3	0 ~ 16	-
12	1	Secondary(Shadow) ID	RW	255	0 ~ 252	-
13	1	Protocol Type	RW	2	2 ~ 22	-
20	4	Homing Offset	RW	0	-1,044,479 ~ 1,044,479	1 [pulse]
24	4	Moving Threshold	RW	10	0 ~ 1,023	0.229 [rev/min]
31	1	Temperature Limit	RW	70	0 ~ 100	1 [°C]
32	2	Max Voltage Limit	RW	70	31 ~ 70	0.1 [V]
34	2	Min Voltage Limit	RW	35	31 ~ 70	0.1 [V]
36	2	PWM Limit	RW	885	0 ~ 885	0.113 [%]
38	2	Current Limit	RW	1,750	0 ~ 1,750	1 [mA]
44	4	Velocity Limit	RW	1,620	0 ~ 2,047	0.229 [rev/min]
48	4	Max Position Limit	RW	4,095	0 ~ 4,095	1 [pulse]
52	4	Min Position Limit	RW	0	0 ~ 4,095	1 [pulse]
60	1	Startup Configuration	RW	0	3	-
62	1	PWM Slope	RW	140	1 ~ 255	1.977 [mV/msec]
63	1	Shutdown	RW	53	-	-

Figura 2.4: Un fragmento de la tabla de control para Dynamixel XL330-M077 (*Fuente: Robotis*)

3. **Rangos de valores:** Es necesario conocer los rangos de valores aceptados por los motores Dynamixel para evitar errores de operación.

4. **Feedback limitado:** La falta de feedback inmediato sobre el resultado de las operaciones realizadas dificulta la depuración y el seguimiento de errores.

La información sobre el éxito de una operación puede ser necesaria para asegurar el flujo de comandos correcto dentro de una aplicación (la ejecución de un comando puede ser consecuencia de la correcta ejecución del comando anterior) o simplemente para una visualización del estado actual del motor.

5. **Secuencia de trabajo:** La necesidad de comprender la secuencia adecuada de pasos a seguir antes de enviar instrucciones al motor puede resultar complicado para usuarios iniciales y ser propenso a errores.

6. **Interpretación de datos:** Es laborioso interpretar la información recibida del motor, especialmente en lo que respecta a la decodificación de bits y registros.

2.2. Funcionamiento de ROS

Dado que “dynamixel_ros_library” está pensada para su uso con ROS, surge la necesidad de detallar algunos conceptos básicos.

El *Robot Operating System* (ROS) es un marco flexible para escribir software de robots [23] [24]. Es una colección de herramientas, bibliotecas y convenciones que tiene como objetivo simplificar la tarea de crear comportamientos complejos y robustos en robots desde una amplia variedad de plataformas robóticas.

2.2.1. Arquitectura

ROS sigue una arquitectura distribuida, lo que significa que los procesos individuales que forman el sistema pueden ejecutarse en diferentes máquinas. Estos procesos se conocen como *nodos*. Un nodo es una instancia de un programa en ejecución que realiza tareas de computación. La comunicación entre nodos se realiza principalmente mediante mensajes.

- **Nodos:** Los nodos son procesos que realizan cálculos. Se utilizan para dividir la funcionalidad del software en módulos intercambiables.
- **Topic:** Los topics son canales de comunicación que permiten el intercambio de mensajes entre nodos. Un nodo puede publicar o suscribirse a un topic.
- **Mensajes:** Los mensajes son paquetes de datos que los nodos envían entre sí. Un nodo publica un mensaje en un topic, y otros nodos suscritos a ese topic reciben el mensaje. Cabe mencionar que dentro de un mismo mensaje pueden transmitirse varios datos, incluso llegando a ser de varios tipos si es necesario.
- **Servicios:** Además de la comunicación basada en topics, ROS también proporciona un sistema de llamadas a procedimientos remotos, conocido como servicios. Un servicio es una operación que se puede llamar desde un nodo y que devuelve una respuesta. De tal modo, los servicios son comunicaciones bilaterales, es decir, requieren la participación de ambos extremos de la línea de comunicación.
- **Parámetros:** ROS tiene un servidor de parámetros que proporciona un almacenamiento global compartido para datos de configuración que se puede acceder mediante nodos.

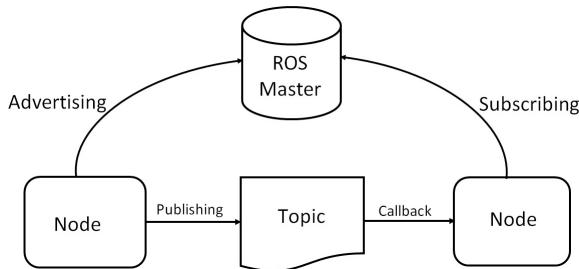


Figura 2.5: Representación gráfica de mensajería a través de topics en ROS (*Fuente: robinrobotics.blogspot.com*)

2.2.2. Comunicaciones

Las comunicaciones en ROS se gestionan mediante un sistema de paso de mensajes. Existen dos tipos principales de comunicación entre nodos: la comunicación basada en topics y la comunicación basada en servicios.

2.2.2.1. Comunicación Basada en Topics

En la comunicación basada en topics representada en la Figura 2.5, los nodos intercambian mensajes mediante la publicación y suscripción a los topics [25]. Éstos no son mas que direcciones donde un nodo puede depositar un mensaje, o lo que es lo mismo, una información.

En el envío y la recepción de un mensaje, podemos destacar 2 partes fundamentales:

- **Publicadores (Publishers):** Nodo que envía mensajes.
- **Suscriptores (Subscribers):** Nodo que recibe mensajes.

Es importante destacar que el intercambio de mensajes no es directo entre los nodos. Siempre hay un nodo maestro que es el encargado de valorar si un nodo debe recibir un mensaje publicado (en el caso de estar suscrito al topic donde se ha publicado dicho mensaje).

2.2.2.2. Comunicación Basada en Servicios

En la comunicación basada en servicios, un nodo ofrece una funcionalidad específica y otros nodos pueden llamar a esta funcionalidad mediante solicitudes de servicio. Un servicio tiene un nombre, una solicitud y una respuesta.

- **Servidores de Servicios:** Un nodo que proporciona una funcionalidad se denomina *servidor de servicios*. El servidor de servicios define la lógica de la operación y envía una respuesta al cliente.
- **Clientes de Servicios:** Un nodo que llama a una funcionalidad se denomina *cliente de servicios*. El cliente de servicios envía una solicitud al servidor y espera una respuesta.

2.2.3. Herramientas de ROS

ROS proporciona una serie de herramientas que facilitan el desarrollo y la depuración de software. Las herramientas básicas necesarias son están listadas a continuación:

- **roscore:** Es el nodo maestro que debe estar ejecutándose para que otros nodos ROS puedan comunicarse entre sí. *roscore* proporciona los servicios de nombre y registro.
- **rosnode:** Es una herramienta para la gestión de nodos. Permite listar, obtener información y finalizar nodos.
- **rostopic:** Es una herramienta para interactuar con topics. Permite listar, publicar, suscribirse y obtener información sobre los topics existentes en el sistema.
- **rosservice:** Herramienta análoga a rostopic para el manejo de servicios.
- **rqt:** Es una interfaz gráfica de usuario que proporciona una colección de herramientas visuales para depurar y visualizar la información de ROS.

2.2.4. Paquetes y Dependencias

En ROS, el software se organiza en unidades conocidas como *paquetes*. Un paquete puede contener nodos, bibliotecas, datos de configuración, y más. Los paquetes pueden depender de otros paquetes, y estas dependencias se gestionan mediante los siguientes archivos:

- **CMakeLists.txt:** Es el archivo de configuración de CMake que especifica cómo se debe construir el paquete.
- **package.xml:** Es el archivo de manifiesto que describe el paquete y sus dependencias.

2.2.5. Tipos de Datos en ROS

ROS proporciona una variedad de tipos de datos estándar para los mensajes. Estos tipos de datos se definen en archivos con extensión .msg y son utilizados por los nodos para intercambiar información.

- **Tipos Primitivos:** Incluyen enteros, flotantes, booleanos, y cadenas de caracteres.
- **Tipos Compuestos:** Incluyen vectores y arreglos.
- **Tipos Específicos de ROS:** Incluyen tipos de datos específicos como Header, que contiene información sobre el tiempo y el marco de coordenadas.

2.2.6. Uso de ROS con los Actuadores Dynamixel

En el marco de servomotores inteligentes de Dynamixel, ROS es una herramienta clave para un control preciso y eficiente del motor. Las estructuras como nodos y la comunicación entre ellos facilita la monitorización de los parámetros del motor y el control del mismo.

2.3. Programación orientada a objetos (POO)

La Programación Orientada a Objetos (POO) es un paradigma de programación que se centra en elementos abstractos llamados “objetos” y sus características para diseñar aplicaciones y programas informáticos. Este enfoque se centra en los conceptos de “objetos” y “clases”, y es ampliamente utilizado debido a sus beneficios en términos de modularidad, reutilización y facilidad de mantenimiento del código.

Este modelo de programación está inspirado en la manera de ver y entender la vida cotidiana por los seres humanos. En concreto, la manera de la que le asignamos ciertas características a un objeto cualquiera (color de un coche, marca de un reloj, etc.) [26, 27].

2.3.1. Conceptos Fundamentales de la POO

2.3.1.1. Clases y Objetos

Clase: Una clase es una plantilla o modelo a partir del cual se crean los objetos. Define un conjunto de propiedades y métodos que son comunes a todos los objetos de ese tipo. Por ejemplo, una clase Coche puede tener propiedades como color, marca, y modelo, y métodos como acelerar y frenar.

Objeto: Un objeto es una instancia de una clase. Representa una entidad concreta que tiene un estado y un comportamiento definidos por las propiedades y métodos de la clase correspondiente. Por ejemplo, un objeto miCoche de la clase Coche puede tener el color negro, la marca SEAT y el modelo Arona.

2.3.1.2. Propiedades y Métodos

Propiedades: Las propiedades, también conocidas como atributos o campos, representan las características de una clase. Definen el estado del objeto. En la analogía del coche, un atributo podría ser el tamaño, el color o el año de fabricación.

Métodos: Los métodos son funciones o procedimientos asociados a una clase que definen los comportamientos de los objetos de esa clase. Volviendo a la analogía utilizada, un método de la clase Coche podría ser acelerar o encenderIntermitente.

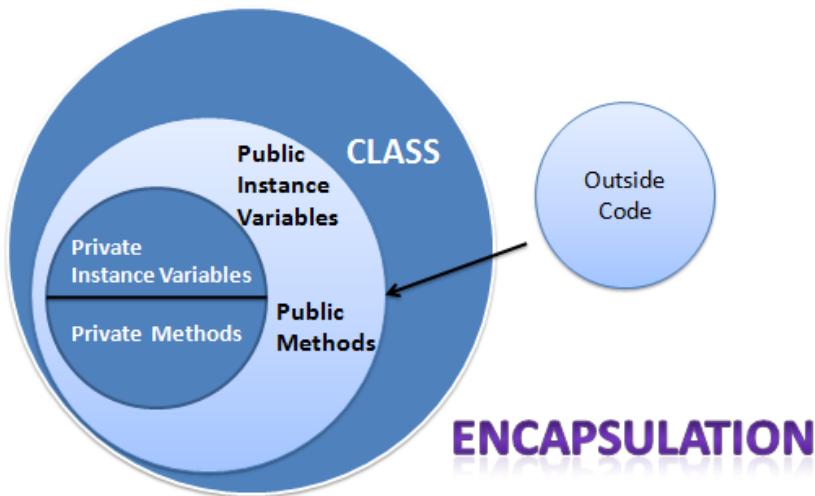


Figura 2.6: Principios de encapsulamiento. Funcionamiento de los modificadores de acceso (*Fuente: grupocodesi.com*)

2.3.1.3. Encapsulamiento

Este concepto se refiere a la agrupación de datos y métodos que operan sobre esos datos en una única unidad, que es la clase [28]. Además, el acceso desde el exterior de la clase a los atributos que le pertenecen puede ser modificado por el usuario (público, privado, protegido).

En aplicaciones no demasiado complejas se suelen utilizar los modificadores de acceso `private` y `public`. Debido al encapsulamiento, resulta imposible acceder o modificar una variable acompañada del modificador `private`. Sin embargo, una variable con el modificador de acceso `public` puede ser modificada y/o consultada en cualquier momento. En la Figura 2.6 se representa un esquema básico de funcionamiento de estos dos modificadores.

2.3.1.4. Herencia

La herencia es un mecanismo que permite crear una nueva clase a partir de una clase existente [29]. La nueva clase, llamada subclase o clase derivada, hereda las propiedades y métodos de la clase base o clase padre, y puede añadir nuevas propiedades y métodos o modificar los existentes.

La herencia puede estar organizada de varias maneras, algunas de ellas se muestran en la Figura 2.7.

2.3.2. Ventajas de la POO

Vistas las características de algunos conceptos relevantes, se procede a analizar las ventajas que ofrece el concepto de POO.

Modularidad: La POO permite segmentar un programa complejo en unidades más pequeñas y manejables llamadas clases. Esto facilita la comprensión y el mantenimiento del código.

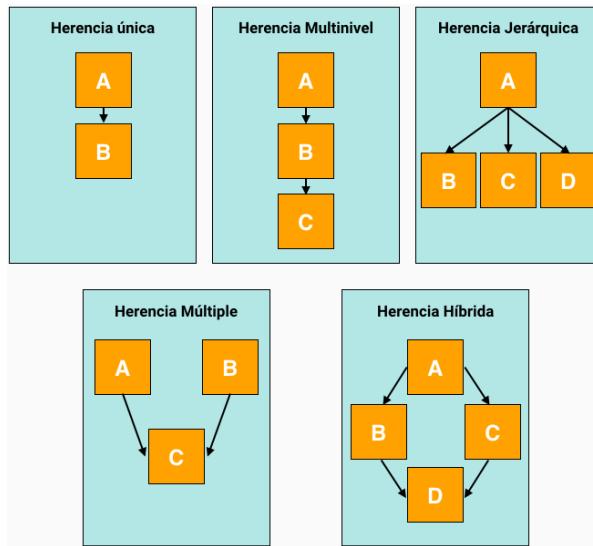


Figura 2.7: Tipos de herencia (*Fuente: ifgeekthen.com*)

Reutilización: Las clases y objetos pueden reutilizarse en diferentes programas, lo que reduce el tiempo de desarrollo y mejora la eficiencia.

Mantenibilidad: El código orientado a objetos es más fácil de mantener y actualizar. Los cambios en una clase no afectan necesariamente a otras clases, lo que facilita la incorporación de nuevas funcionalidades.

Flexibilidad y Extensibilidad: La POO permite la creación de sistemas flexibles y extensibles mediante el uso de herencia y polimorfismo. Los programas pueden adaptarse fácilmente a nuevos requisitos sin necesidad de realizar cambios significativos en el código existente.

Seguridad: El encapsulamiento proporciona una capa adicional de seguridad al ocultar los datos internos de los objetos y permitir el acceso solo a través de métodos públicos.

2.3.3. Particularización de POO para la programación de los motores Dynamixel

La programación orientada a objetos es una herramienta potente y muy utilizada hoy en día tanto en el sector de la robótica como fuera de él.

En el caso de los actuadores inteligentes Dynamixel, el uso de POO permitirá la creación de una herramienta potente y eficaz, que puede ser fácilmente modificada en el futuro y/o adaptada para modelos nuevos del hardware. La identificación de la clase a implementar y los atributos que deben pertenecer a la misma se analizan en el capítulo siguiente.

Metodología

Contenido

3.1.	Elección de C++ y ROS	22
3.2.	Dependencias	22
3.3.	Estructura	23
3.3.1.	Tablas de Control	23
3.3.2.	Campos de clase	23
3.3.3.	Atributos estáticos de dynamixelMotor	24
3.3.4.	Métodos estáticos de dynamixelMotor	26
3.3.5.	Métodos de objeto dynamixelMotor	26

Este capítulo tratará de desarrollar todos los pasos y decisiones que se han tomado durante la creación de la solución final, su estructura y peculiaridades. Se comenzará con varios comentarios sobre la elección del lenguaje de programación, el entorno ROS y las dependencias. Posteriormente, se explicará la estructura básica del paquete, comentando las posibles alternativas y los criterios para la elección de la solución final, y se describirán las particularidades y el uso correcto de todos y cada uno de los elementos que conforman el paquete “dynamixel_ros_library”.

3.1. Elección de C++ y ROS

La elección de C++ como lenguaje de programación para el desarrollo de la librería *dynamixel_ros_library* se basa en su eficiencia y rendimiento. Aunque Python es extensamente utilizado en proyectos de robótica debido a su facilidad de uso y flexibilidad, es un lenguaje interpretado y puede ser más lento en comparación con C++.

Por otro lado, ROS (Robot Operating System) se ha convertido en un estándar de facto en la comunidad de robótica debido a su arquitectura modular y su amplia gama de herramientas y bibliotecas disponibles. Al aprovechar ROS, se puede acceder a una serie de funcionalidades listas para usar, como la comunicación entre nodos, la visualización de datos y la simulación, lo que facilita el desarrollo y la depuración de sistemas robóticos complejos.

3.2. Dependencias

El paquete *dynamixel_ros_library* depende de varias bibliotecas y paquetes externos que son fundamentales para su funcionamiento. A continuación se detallan las dependencias y la razón por la cual son necesarias:

1. **roscpp:** Este es uno de los paquetes fundamentales del Robot Operating System (ROS). Proporciona las herramientas y bibliotecas esenciales para el desarrollo de nodos ROS en C++. La librería *dynamixel_ros_library* utiliza roscpp para la comunicación con el sistema de ROS, la gestión de mensajes y la creación de nodos que interactúan con otros componentes del sistema robótico.
2. **dynamixel_sdk:** Al proporcionar definiciones de elementos fundamentales como las funciones de escritura y lectura, y clases para manejo del puerto USB, este paquete es la base sobre la que se construye el paquete *dynamixel_ros_library*.
3. **std_msgs:** Este paquete define mensajes estándar utilizados en ROS para la comunicación entre nodos. Incluye tipos de mensajes como String, Int32, Float64, entre otros, que son utilizados para intercambiar datos entre los nodos de ROS de manera consistente y eficiente. La librería *dynamixel_ros_library* hace uso de std_msgs para enviar y recibir mensajes estándar entre nodos ROS, facilitando la interoperabilidad con otros componentes del sistema robótico.

Estas dependencias externas son críticas para el funcionamiento adecuado de la librería *dynamixel_ros_library*, ya que proporcionan las herramientas necesarias para la comunicación, la gestión de mensajes y la interoperabilidad dentro del ecosistema de ROS, así como la interacción específica con los actuadores Dynamixel de Robotis.

3.3. Estructura

3.3.1. Tablas de Control

En el desarrollo del presente Trabajo de Fin de Grado, se ha empleado la estructura de datos `std::map` proporcionada por el lenguaje de programación C++. Esta estructura se ha revelado como una herramienta fundamental para la gestión eficiente y organizada de datos clave en el sistema implementado.

`std::map` es una estructura de datos asociativa que permite almacenar elementos en pares clave-valor, donde cada clave está asociada a un único valor correspondiente. En el contexto de este proyecto, `std::map` ha sido utilizada para vincular los nombres de parámetros específicos a direcciones de registros correspondientes en la memoria interna de los motores Dynamixel.

Una de las ventajas fundamentales de `std::map` radica en su capacidad para proporcionar un acceso rápido y eficiente a los valores almacenados. Al utilizar las claves asociadas, se pueden recuperar los valores correspondientes de manera instantánea, lo que resulta especialmente beneficioso en un entorno donde se necesita un acceso rápido a la información de los motores Dynamixel.

Otra característica destacada de `std::map` es su capacidad para manejar de forma eficiente la inserción, eliminación y búsqueda de elementos. Esto se traduce en un código más limpio, organizado y fácil de mantener, ya que `std::map` se encarga de las operaciones subyacentes de manera transparente.

En resumen, la utilización de `std::map` en este proyecto ha sido fundamentada en su capacidad para proporcionar un acceso eficiente, organizado y estructurado a la información clave de los motores Dynamixel. Su naturaleza asociativa, ordenada y eficiente en términos de operaciones la convierte en una herramienta invaluable para el desarrollo de sistemas robóticos complejos y de alto rendimiento.

3.3.2. Campos de clase

A continuación, se nombrarán y se explicarán los atributos que pertenecen a un objeto cualquiera de la clase `dynamixelMotor`. Cabe destacar que, por razones de seguridad para evitar un manejo incorrecto de los valores por parte del usuario, todos los campos de clase utilizan la etiqueta `private`. Esto significa que no se puede acceder directamente a ellos desde fuera de la clase. Para interactuar con estos atributos, se proporcionan sus respectivos métodos getters (para consultar el valor) y setters (para cambiarlo).

- **ID:** atributo de tipo básico `int`, almacena el número de identificación correspondiente al motor. Su importancia radica en que todas las funciones implementadas dentro de la librería que interactúan con los registros del motor requieren este ID como parámetro, determinando así sobre qué motor realizar la acción. Cabe destacar que, aunque es posible asignar el mismo

ID a varios dynamixels, esta práctica es altamente desaconsejada debido a que puede ocasionar errores difíciles de identificar y un funcionamiento incorrecto.

- **MODEL:** atributo de tipo básico `int`, contiene el número de modelo específico del dynamixel. Cada modelo de dynamixel tiene un número único proporcionado por el fabricante. El atributo se utiliza en múltiples métodos para distinguir entre los diferentes modelos existentes y, como resultado, ejecutar diferentes partes del código. Se ha elegido el tipo `int` para este atributo en lugar de `std::string` por su mayor conveniencia en el uso de sentencias como `switch` o `if`.
- **IDENTIFIER:** atributo de tipo `std::string`, ha sido creado exclusivamente para la comodidad y uso del usuario. Su función es identificar el motor como una pieza dentro de un sistema más amplio compuesto por varios dynamixels.
- **CONTROL_TABLE:** es un elemento de tipo `std::map` que representa la tabla de control específica de cada Dynamixel, la cual varía según el modelo del motor. Este atributo se inicializa cuando el usuario invoca la función `setControlTable()` (cuya explicación detallada se encuentra en secciones posteriores). Esta tabla es esencial para la correcta interacción con el Dynamixel, ya que contiene las direcciones de los registros que controlan sus diferentes funcionalidades y parámetros. El uso de un `std::map` permite un acceso eficiente a estas direcciones durante la ejecución de los métodos de la librería, facilitando así la configuración y lectura de los valores necesarios para el control del motor Dynamixel.

3.3.3. Atributos estáticos de `dynamixelMotor`

Los atributos estáticos descritos a continuación, incluidas las constantes de clase, son elementos relacionados con la clase `dynamixelMotor` que poseen una utilidad específica y constante. Es importante destacar que al ser estáticos, estos atributos pertenecen a la clase en sí misma, no a una instancia particular de la misma.

- **DMXL_MODELS:** Es un `std::map` que establece una relación entre el número de modelo proporcionado por Dynamixel y un modelo específico del mismo. Este atributo es fundamental dentro de la librería, ya que internamente se manejan los números de modelo para facilitar su gestión. Sin embargo, al mostrar esta información al usuario, es preferible y más comprensible devolver el nombre del modelo en lugar de su número correspondiente.
- **ADDR_DMXL22:** Este elemento del tipo `std::map` se utiliza en el código para representar el área EEPROM de la tabla de control de los Dynamixels con 22 parámetros en la misma. Su función principal radica en la configuración de la tabla de control de la instancia de `dynamixelMotor` sobre la cual se ha aplicado el método `setControlTable()`. Esta versión con 22 parámetros es compatible con los modelos XW, XD430, XH430 y XM430 de Dynamixels.

- **ADDR_DMXL25:** Este miembro del tipo `std::map` desempeña un papel similar al de `ADDR_DMXL22`, pero está diseñado específicamente para los modelos de Dynamixels con una tabla de control de 25 parámetros en la EEPROM. Al utilizar la función `setControlTable()`, este elemento se encarga de configurar la tabla de control de la instancia de `dynamixelMotor`. La versión con 25 parámetros es compatible con los modelos XD540, XH540 y XM540 de Dynamixels.
- **CURRENT_CONTROL_MODE:** Esta constante de clase está diseñada para referirse al modo de control del motor mediante la corriente suministrada. Se ha creado con el propósito de facilitar al usuario el manejo de los métodos que requieren como parámetro un modo de control específico. Al utilizar `dynamixelMotor.CURRENT_CONTROL_MODE`, el usuario puede especificar el modo de control deseado al llamar a los métodos relevantes, sin la necesidad de recordar los valores numéricos correspondientes.
- **VELOCITY_CONTROL_MODE:** Esta constante de clase se utiliza para referirse al modo de control del motor mediante una consigna de velocidad. Su propósito es proporcionar al usuario una forma fácil y directa de acceder a los valores numéricos asignados a cada modo de control de velocidad. Al utilizar `dynamixelMotor.VELOCITY_CONTROL_MODE`, el usuario puede especificar el modo de control deseado al llamar a los métodos relevantes, sin la necesidad de recordar los valores numéricos correspondientes.
- **POSITION_CONTROL_MODE:** Esta constante de clase está diseñada para referirse al modo de control del motor mediante una consigna de posición. Su objetivo es proporcionar al usuario una manera sencilla de acceder a los valores numéricos asignados a cada modo de control de posición. Al utilizar `dynamixelMotor.POSITION_CONTROL_MODE`, el usuario puede especificar el modo de control deseado al llamar a los métodos pertinentes, sin la necesidad de recordar los valores numéricos correspondientes.
- **EXTENDED_POSITION_CONTROL_MODE:** Este modo de control del motor está diseñado para manejar la posición mediante una consigna extendida. Para utilizar este modo de control en la clase `dynamixelMotor`, simplemente se hace referencia a la constante `EXTENDED_POSITION_CONTROL_MODE`, proporcionando así una forma conveniente y legible de configurar el motor para este tipo específico de control.
- **CURRENT_BASED_CONTROL_MODE:** Este modo de control del motor permite tanto el control basado en la corriente como el control mediante una consigna de posición. Proporciona una versatilidad adicional al usuario al permitir el ajuste dinámico del controlador para adaptarse a diferentes situaciones de funcionamiento. Al utilizar la constante `CURRENT_BASED_CONTROL_MODE` en la clase `dynamixelMotor`, los usuarios pueden seleccionar fácilmente este modo de control sin tener que preocuparse por los detalles técnicos asociados.

Esto simplifica significativamente la interfaz y hace que sea más accesible para aquellos que buscan un control preciso y versátil del motor Dynamixel.

- **PWM_CONTROL_MODE:** Este modo de control del motor permite controlar el motor Dynamixel mediante una consigna de PWM (Modulación por Ancho de Pulso). En este modo, el usuario puede especificar la duración del pulso PWM enviado al motor para controlar su velocidad y posición. Esta funcionalidad es particularmente útil cuando se requiere un control preciso sobre la velocidad y posición del motor Dynamixel en diversas aplicaciones. Al utilizar la constante PWM_CONTROL_MODE en la clase dynamixelMotor, los usuarios pueden seleccionar fácilmente este modo de control sin tener que preocuparse por los detalles técnicos asociados, lo que simplifica la interfaz y hace que sea más accesible para aquellos que buscan una forma intuitiva de controlar el motor Dynamixel.

3.3.4. Métodos estáticos de dynamixelMotor

La librería dynamixel_ros_library dispone de un único método con la etiqueta static.

- **static bool iniComm(char* PORT_NAME, float PROTOCOL_VERSION, int BAUDRATE):** Este método estático de la clase dynamixelMotor se utiliza para inicializar la comunicación con los motores. Toma tres parámetros: el nombre del puerto (PORT_NAME), la versión del protocolo de comunicación (PROTOCOL_VERSION) y la velocidad de transmisión de datos (BAUDRATE).

El método comienza creando instancias de PortHandler y PacketHandler utilizando los valores proporcionados de PORT_NAME y PROTOCOL_VERSION.

A continuación, intenta abrir el puerto de comunicación especificado. Si la apertura del puerto falla, el método registra un mensaje de error utilizando ROS_ERROR e inmediatamente retorna un valor booleano false. Si el puerto se abre correctamente, intenta establecer la velocidad de transmisión especificada por BAUDRATE.

Si no se puede establecer la velocidad de transmisión, el método registra un mensaje de error y retorna false. Si tanto la apertura del puerto como la configuración de la velocidad de transmisión son exitosas, el método registra un mensaje utilizando ROS_INFO y retorna true.

En resumen, este método es fundamental para establecer una comunicación inicial con los motores Dynamixel antes de enviar comandos o recibir datos. Maneja adecuadamente los posibles errores de comunicación y asegura que los parámetros de conexión sean configurados correctamente.

3.3.5. Métodos de objeto dynamixelMotor

En este apartado se abordarán los métodos que pueden aplicarse a un objeto de la clase dynamixelMotor. Estos métodos, en su mayoría, tienen como propósito modificar una variable

específica almacenada en la memoria del motor dynamixel, como por ejemplo la consigna de velocidad, el límite de temperatura, u otros atributos definidos como IDENTIFIER o MODEL. Para la implementación adecuada de estos métodos, se ha seguido la convención de dividirlos en dos categorías principales: getters (para consultar el valor de una variable) y setters (para modificarlo).

Además de estos, existen métodos diseñados para la configuración del motor, cuyo uso puede resultar en cambios en múltiples variables o en ninguna de ellas. A continuación, se detallan todos los métodos que pertenecen a un objeto de la clase dynamixelMotor:

- **dynamixelMotor(std::string IDENTIFIER, int ID):** Este método actúa como el constructor de los objetos pertenecientes a la clase dynamixelMotor y se invoca al crear una nueva instancia de dicha clase. Su función principal es inicializar un objeto de dynamixelMotor con los valores proporcionados por los parámetros de entrada.

El constructor toma dos parámetros: `std::string IDENTIFIER` y `int ID`. El parámetro IDENTIFIER es un dato del tipo `string` que representa el identificador del motor, y el parámetro ID es un dato del tipo `int` que representa el identificador numérico único del motor.

Al invocar este constructor, los valores proporcionados para IDENTIFIER e ID se asignan de manera inmediata a los atributos correspondientes del objeto dynamixelMotor. Esto permite que cada instancia de la clase tenga un identificador y un ID que la distingue de otros motores en el sistema. Esta inicialización es crucial para el posterior funcionamiento y control de los motores Dynamixel, ya que estos identificadores se utilizan en la comunicación y operación del hardware.

El uso de este constructor asegura que cada objeto dynamixelMotor esté correctamente configurado con sus atributos básicos desde el momento de su creación, estableciendo una base sólida para las operaciones subsecuentes que se realizarán con los motores Dynamixel.

- **~dynamixelMotor():** El método destructor de la clase dynamixelMotor. Este método se invoca automáticamente cuando un objeto de la clase dynamixelMotor alcanza el final de su ciclo de vida, es decir, cuando ya no es necesario y su memoria puede ser liberada. La función principal del destructor es realizar cualquier limpieza necesaria, como la liberación de recursos asignados durante la vida del objeto, garantizando así que no haya fugas de memoria y que los recursos se gestionen de manera eficiente. Aunque en este caso específico no se realicen operaciones complejas en el destructor, su presencia es crucial para el correcto manejo de la memoria y los recursos en C++.
- **void dynamixelMotor::setControlTable():** Este método establece la tabla de control del motor Dynamixel en función del modelo específico del motor. El método comienza asignando dinámicamente memoria para un puntero `model_number` de tipo `uint16_t`. Luego, utiliza la función `read2ByteTxRx` de la clase `PacketHandler` para leer el número de modelo del motor

desde la dirección 0, almacenando el resultado en `model_number` y verificando si hay errores de comunicación mediante `dxl_error`.

Si la lectura del número de modelo falla, se registra un mensaje de error con el identificador del motor y el código de error. En caso de éxito, el número de modelo se asigna al atributo `MODEL` del objeto `dynamixelMotor`. A continuación, se utiliza una estructura `switch` para comparar el número de modelo con una serie de valores predefinidos. Dependiendo del modelo, se asigna la tabla de control correspondiente al atributo `CONTROL_TABLE`.

Si el número de modelo no coincide con ninguno de los valores predefinidos, se asigna un valor de 0 al atributo `MODEL`, indicando un modelo no reconocido. Finalmente, se imprime un mensaje informativo en el que se indica que la tabla de control ha sido configurada para el modelo específico del motor, y se libera la memoria asignada dinámicamente para `model_number`.

Este método es esencial para configurar correctamente los parámetros del motor en función de su modelo, garantizando así una operación adecuada y eficiente.

- **int dynamixelMotor::getID()**: Este método es un accesor de la clase `dynamixelMotor` que se utiliza para obtener el ID del motor Dynamixel. Devuelve el valor del atributo `ID` del objeto, que es de tipo `int`. Al llamar a este método, se proporciona el identificador único del motor, lo cual es útil para realizar operaciones específicas o identificar el motor en un sistema con múltiples motores.
- **void dynamixelMotor::setID(int NEW_ID)**: Este método se utiliza para cambiar el ID del motor Dynamixel. Toma un parámetro de tipo `int`, `NEW_ID`, que representa el nuevo identificador que se desea asignar al motor.

El método primero verifica que `NEW_ID` esté en el rango válido (1 a 253). Si `NEW_ID` es válido, el método utiliza `myPacketHandler` para enviar un comando al motor a través de `myPortHandler`, escribiendo el nuevo identificador en el registro correspondiente del motor. Si la operación de comunicación falla, se registra un mensaje de error usando `ROS_ERROR`, indicando el código de error específico. Si la operación es exitosa, se registrará un mensaje de información usando `ROS_INFO`, y el atributo `ID` del objeto se actualiza con el nuevo valor.

Si `NEW_ID` no está en el rango válido, se registra un mensaje de error indicando que se debe especificar un identificador válido.

- **std::string dynamixelMotor::getModel()**: Este método es un accesor de la clase `dynamixelMotor` que devuelve el modelo del motor Dynamixel en forma de cadena de caracteres (`std::string`). No toma parámetros y simplemente accede al atributo `MODEL` del objeto para obtener el índice correspondiente al modelo en el mapa estático `DMXL_MODELS` de la clase. Luego, devuelve la cadena de caracteres que representa el modelo del motor.

Este método es útil para obtener información sobre el modelo específico de un motor Dynamixel, lo cual puede ser importante para la configuración y el control adecuados del motor en un sistema robótico.

- **int dynamixelMotor::getBaudrate():** Este método es un accesor de la clase `dynamixelMotor` que devuelve la velocidad de transmisión actual del motor Dynamixel en bits por segundo (bps). No toma parámetros y realiza una lectura del registro correspondiente al baudrate del motor a través del método `read1ByteTxRx` del paquete de comunicación básico de Dynamixel. Luego, según el modelo del motor, determina el valor del baudrate leído y lo devuelve como un entero.

El método comienza declarando variables locales para almacenar posibles errores de comunicación y el valor del baudrate. Luego, se realizan comprobaciones del modelo del motor para determinar el rango de baudrates válidos. Dependiendo del modelo del motor, se establecen las posibles velocidades de transmisión y se asigna el valor correspondiente al baudrate.

Finalmente, se imprime un mensaje informativo en la consola de ROS indicando el ID del motor y la velocidad de transmisión actual en bps. Se libera la memoria asignada para almacenar los datos temporales y se devuelve el valor del baudrate.

Este método es útil para obtener información sobre la configuración de comunicación actual del motor Dynamixel, lo cual puede ser importante para el diagnóstico y la depuración de problemas de comunicación en un sistema robótico.

- **int dynamixelMotor::getReturnDelayTime():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el tiempo de retardo de retorno actual del motor Dynamixel en microsegundos. No toma parámetros y realiza una lectura del registro correspondiente al tiempo de retardo de retorno del motor a través del método `read1ByteTxRx` del paquete de comunicación Dynamixel. Luego, ajusta el valor leído para obtener el tiempo en microsegundos y lo devuelve como un entero.

El método comienza declarando variables locales para almacenar posibles errores de comunicación y el valor del tiempo de retardo de retorno. Luego, se realiza una solicitud de lectura al motor para obtener el valor actual del tiempo de retardo de retorno. Si la comunicación es exitosa, el valor leído se ajusta para obtener el tiempo en microsegundos.

Finalmente, se imprime un mensaje informativo en la consola de ROS indicando el ID del motor y el tiempo de retardo de retorno actual en microsegundos. Se devuelve el valor del tiempo de retardo de retorno y se libera la memoria asignada para almacenar los datos temporales.

Este método es útil para obtener información sobre el tiempo que el motor Dynamixel espera antes de enviar una respuesta después de recibir una instrucción de escritura.

- **void dynamixelMotor::setReturnDelayTime(int RETURN_DELAY_TIME):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el tiempo de retardo de

retorno del motor Dynamixel. Toma como parámetro un entero RETURN_DELAY_TIME que representa el tiempo de retardo de retorno deseado en microsegundos.

El método comienza verificando si el valor proporcionado está dentro del rango permitido, es decir, entre 0 y 508 microsegundos. Si el valor es válido, se realiza una solicitud de escritura al motor para establecer el nuevo tiempo de retardo de retorno. La operación de escritura se lleva a cabo mediante el método write1ByteTxRx del paquete de comunicación Dynamixel.

Si la comunicación con el motor es exitosa, se imprime un mensaje informativo en la consola de ROS indicando que el tiempo de retardo de retorno se ha cambiado correctamente. En caso de error de comunicación, se imprime un mensaje de error correspondiente.

Este método es útil para ajustar el tiempo que el motor espera antes de enviar una respuesta después de recibir una instrucción de escritura. Asegúrese de especificar un valor de tiempo de retardo de retorno válido dentro del rango permitido para garantizar un funcionamiento adecuado del motor Dynamixel.

- **std::string dynamixelMotor::getOperatingMode():** Este método es un accesor de la clase dynamixelMotor que devuelve el modo de operación actual del motor Dynamixel como una cadena de texto. El método realiza una lectura del registro correspondiente al modo de operación del motor utilizando el método read1ByteTxRx del paquete de comunicación Dynamixel. Luego, utiliza un bloque switch para determinar el modo de operación basado en el valor leído del registro. Dependiendo del modo de operación, el método devuelve una cadena de texto descriptiva que indica el modo actual. Si se produce algún error durante la comunicación con el motor, se registra un mensaje de error utilizando ROS_ERROR y se devuelve un puntero nulo. Es importante destacar que se debe gestionar correctamente la liberación de la memoria asignada para el arreglo op_mode utilizando el operador delete al final del método.
- **void dynamixelMotor::setOperatingMode(int MODE):** Este método es un modificador de la clase dynamixelMotor que permite cambiar el modo de operación del motor Dynamixel. Toma como parámetro un entero que representa el nuevo modo de operación deseado. Primero, verifica si el modo especificado es válido, es decir, si corresponde a uno de los modos de operación definidos en la clase dynamixelMotor. Si el modo especificado es válido, utiliza el método write1ByteTxRx del paquete de comunicación Dynamixel para escribir el nuevo modo de operación en el registro correspondiente del motor. Si la comunicación con el motor es exitosa, se registra un mensaje de información utilizando ROS_INFO indicando que se ha cambiado el modo de operación. En caso de que ocurra un error durante la comunicación, se registra un mensaje de error utilizando ROS_ERROR junto con el código de error proporcionado por el motor. Si se especifica un modo de operación inválido, se registra un mensaje de error indicando que se debe especificar un modo válido. Es importante destacar que la validez del modo de operación se verifica utilizando la variable modeExists, que se inicializa con

el resultado de una comparación de igualdad entre el modo especificado y los modos de operación definidos en la clase `dynamixelMotor`.

- **int dynamixelMotor::getShadowID():** Este método es un accesador de la clase `dynamixelMotor` que devuelve el ID secundario (shadow ID) del motor Dynamixel. El ID secundario es un identificador alternativo que se puede utilizar para controlar el motor en lugar del ID principal. El método realiza una lectura del registro correspondiente al ID secundario del motor utilizando el método `read1ByteTxRx` del paquete de comunicación Dynamixel. Si la comunicación con el motor es exitosa, el método devuelve el ID secundario como un entero. Si se produce un error durante la comunicación, se registra un mensaje de error utilizando `ROS_ERROR` junto con el código de error proporcionado por el motor. Es importante destacar que el ID secundario se devuelve como un entero, y si este valor es mayor que 252, se considera que el ID secundario está desactivado, y se registra un mensaje de información utilizando `ROS_INFO`. Una vez completada la operación, se devuelve el ID secundario.
- **void dynamixelMotor::setShadowID(int NEW_SH_ID):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el ID secundario (shadow ID) del motor Dynamixel. El método toma un parámetro `NEW_SH_ID` que representa el nuevo ID secundario que se desea establecer para el motor. Antes de realizar cualquier cambio, se verifica que el nuevo ID secundario esté dentro del rango válido de 0 a 255. Si el valor proporcionado está dentro del rango válido, se utiliza el método `write1ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo ID secundario en el registro correspondiente del motor. Si la operación de escritura es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo ID secundario. Si el nuevo ID secundario es 253, 254 o 255, se considera que el ID secundario está desactivado, y se registra un mensaje adicional de información. En caso de que la operación de escritura falle, se registra un mensaje de error utilizando `ROS_ERROR` junto con el código de error proporcionado por el motor.
- **int dynamixelMotor::getProtocolType():** Este método es un accesador de la clase `dynamixelMotor` que devuelve el tipo de protocolo actual utilizado por el motor Dynamixel. No toma ningún parámetro y realiza una lectura del registro correspondiente al tipo de protocolo del motor utilizando el método `read1ByteTxRx` del paquete de comunicación Dynamixel. Luego, devuelve el valor del tipo de protocolo como un entero. Si la lectura del registro es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, indicando el tipo de protocolo actual. En caso de que la operación de lectura falle, se registra un mensaje de error utilizando `ROS_ERROR` junto con el código de error proporcionado por el motor.
- **int dynamixelMotor::getHomingOffset():** Este método es un accesador de la clase `dynamixelMotor` que devuelve el desplazamiento de homing actual del motor Dynamixel en grados. No toma parámetros y realiza una lectura del registro correspondiente utilizando el método

`read4ByteTxRx` del paquete de comunicación Dynamixel. Luego, convierte el valor leído en grados utilizando el factor de conversión proporcionado por el fabricante (0.088° por unidad) y lo devuelve como un entero. Si la lectura del registro es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, indicando el desplazamiento de homing actual en grados. En caso de que la operación de lectura falle, se registra un mensaje de error utilizando `ROS_ERROR` junto con el código de error proporcionado por el motor.

- **void dynamixelMotor::setHomingOffset(int DEGREES):** Este método es un modificador de la clase `dynamixelMotor` que establece el desfase de homing del motor Dynamixel en grados. Toma un parámetro `DEGREES` que representa el nuevo desplazamiento de homing que se desea establecer en grados. Antes de enviar el comando de escritura al motor, verifica si el valor de desplazamiento de homing especificado está dentro del rango permitido por el motor, que es de -255 revoluciones a 255 revoluciones.

Si el valor especificado está dentro del rango permitido, el método calcula el valor correspondiente en unidades del motor (utilizando el factor de conversión proporcionado por el fabricante, que es 0.088° por unidad), y luego envía el comando de escritura utilizando el método `write4ByteTxRx` del paquete de comunicación Dynamixel. Si la operación de escritura es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo desplazamiento de homing establecido en grados. En caso de que la operación de escritura falle, se registra un mensaje de error utilizando `ROS_ERROR` junto con el código de error proporcionado por el motor.

- **double dynamixelMotor::getMovingThreshold():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el umbral de movimiento actual del motor Dynamixel en revoluciones por minuto (RPM). No toma parámetros y realiza una lectura del registro correspondiente del motor utilizando el método `read4ByteTxRx` del paquete de comunicación Dynamixel.

Después de leer el valor del umbral de movimiento, el método calcula el valor correspondiente en RPM utilizando el factor de conversión proporcionado por el fabricante, que es de 0.229 RPM por unidad. Luego, registra un mensaje informativo utilizando `ROS_INFO`, indicando el umbral de movimiento actual. Si la lectura es exitosa, devuelve el umbral de movimiento como un valor de tipo double. En caso de que la lectura falle, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

- **void dynamixelMotor::setMovingThreshold(double RPM):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el umbral de movimiento del motor Dynamixel en revoluciones por minuto (RPM). Toma un parámetro de tipo double que representa el nuevo valor del umbral de movimiento.

Primero, verifica si el valor proporcionado está dentro del rango permitido, que va desde 0 hasta 235 RPM, según las especificaciones del fabricante. Si el valor está dentro de este rango, el método calcula el valor equivalente en unidades del motor, utilizando el factor de conversión proporcionado (0.229 RPM por unidad). Luego, utiliza el método `write4ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor del umbral de movimiento en el registro correspondiente del motor.

Si la operación de escritura es exitosa, registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo umbral de movimiento establecido en RPM. En caso de que la operación de escritura falle, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor. Si el valor proporcionado está fuera del rango permitido, registra un mensaje de error indicando que se debe especificar un umbral de movimiento válido dentro del rango aceptado.

- **int dynamixelMotor::getTempLimit():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el límite de temperatura actual del motor Dynamixel en grados Celsius (°C). No toma parámetros y realiza una lectura del registro correspondiente al límite de temperatura del motor a través del método `read1ByteTxRx` del paquete de comunicación Dynamixel. Luego, devuelve el valor leído como un entero.

Si la lectura es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el límite de temperatura actual del motor en grados Celsius. En caso de que la lectura falle, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

- **void dynamixelMotor::setTempLimit(int TEMPERATURE):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el límite de temperatura del motor Dynamixel. Toma como parámetro un entero `TEMPERATURE`, que representa el nuevo límite de temperatura en grados Celsius (°C).

Antes de realizar la escritura, se verifica si el valor proporcionado está dentro del rango permitido (entre 0 y 100 °C). Si el valor es válido, se utiliza el método `write1ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo límite de temperatura en el registro correspondiente del motor.

Si la escritura es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo límite de temperatura establecido para el motor. En caso de que la escritura falle, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

- **float dynamixelMotor::getMaxVoltageLimit():** Este método es un accesor de la clase `dynamixelMotor` que permite obtener el límite de voltaje del motor Dynamixel. No toma parámetros y devuelve un valor flotante que representa el límite en voltios (V).

Para obtener este valor, se utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer el registro correspondiente del motor que almacena el límite máximo de voltaje. El valor leído se convierte a unidades de voltaje dividiendo por 10 para obtener el valor en voltios.

Si la lectura es exitosa (COMM_SUCCESS), registra un mensaje informativo utilizando ROS_INFO, indicando el límite máximo de voltaje actual del motor. En caso de que la lectura falle, registra un mensaje de error utilizando ROS_ERROR, junto con el código de error proporcionado por el motor.

- **void dynamixelMotor::setMaxVoltageLimit(float MAX_VOLTAGE):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el límite de voltaje del motor Dynamixel. Toma un parámetro de tipo flotante que representa el nuevo límite en voltios (V) y no devuelve ningún valor.

Antes de realizar cualquier acción, se verifica que el valor proporcionado para el nuevo límite de voltaje esté dentro del rango permitido, que va desde 9.5 V hasta 16 V. Si el valor está dentro de este rango, se realiza la conversión necesaria para ajustar el valor al formato requerido por el registro del motor y se utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo límite en el registro correspondiente del motor.

Si la escritura es exitosa (COMM_SUCCESS), se registra un mensaje informativo utilizando ROS_INFO, indicando que el límite máximo de voltaje ha sido cambiado. En caso de que la escritura falle, se registra un mensaje de error utilizando ROS_ERROR, junto con el código de error proporcionado por el motor. Si el valor proporcionado está fuera del rango permitido, se registra un mensaje de error indicando que se debe especificar un límite máximo de voltaje válido.

- **float dynamixelMotor::getMinVoltageLimit():** Este método es un accesor de la clase `dynamixelMotor` que permite obtener el límite mínimo de voltaje del motor Dynamixel. No toma parámetros y devuelve un valor de punto flotante que representa el límite mínimo de voltaje en voltios (V).

El método inicia creando un arreglo dinámico de un solo elemento de tipo `uint16_t`, que se utilizará para almacenar el valor leído del registro correspondiente al límite mínimo de voltaje del motor. Luego, utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para realizar una lectura del registro del motor.

Si la comunicación con el motor es exitosa (COMM_SUCCESS), el valor leído se convierte a voltios y se registra un mensaje informativo utilizando ROS_INFO, indicando el límite mínimo de voltaje actual del motor. Finalmente, el método devuelve el valor leído.

En caso de que la comunicación con el motor falle, se registra un mensaje de error utilizando ROS_ERROR, junto con el código de error proporcionado por el motor. Si se produce un error,

el método devuelve -1. Una vez finalizada la lectura, se libera la memoria asignada al arreglo dinámico.

- **void dynamixelMotor::setMinVoltageLimit(float MIN_VOLTAGE):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el límite inferior de voltaje del motor Dynamixel. Toma un parámetro de tipo flotante que representa el nuevo valor del límite en voltios (V).

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra variable de tipo entero (`conversion`) que se utilizará para convertir el valor de voltaje proporcionado a la unidad necesaria para escribir en el registro del motor. Luego, verifica si el valor de voltaje proporcionado está dentro del rango permitido (entre 9.5 V y 16 V).

Si el valor proporcionado es válido, realiza una conversión necesaria de unidades y utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor del límite inferior de voltaje en el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando que el límite mínimo de voltaje ha sido cambiado correctamente. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Si el valor proporcionado está fuera del rango permitido, registra un mensaje de error indicando que se debe especificar un límite válido.

En resumen, este método permite cambiar el límite inferior de voltaje del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **int dynamixelMotor::getPWMLimit():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el límite de ancho de pulso modulado (PWM) actual del motor Dynamixel. No toma parámetros y devuelve un entero que representa el porcentaje del límite PWM.

El método comienza declarando una variable `dxl_error` para almacenar los posibles errores de comunicación, un puntero `data` para almacenar los datos leídos del motor y una variable `conversion` para calcular el porcentaje del límite PWM a partir del valor leído. Luego, utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer el valor del límite PWM del motor desde el registro correspondiente.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), calcula el porcentaje del límite PWM utilizando la conversión necesaria y registra un mensaje informativo utilizando `ROS_INFO`, indicando el límite PWM actual. Finalmente, devuelve el valor del límite PWM como un entero.

En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor, y devuelve un -1 para indicar un error.

En resumen, este método permite obtener el límite de ancho de pulso modulado (PWM) actual del motor Dynamixel y gestionar adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setPWMLimit(int PWM):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el límite de ancho de pulso modulado (PWM) del motor Dynamixel. Toma un parámetro `PWM` que representa el nuevo valor del límite PWM en porcentaje del total posible.

El método comienza declarando una variable `dxl_error` para almacenar errores de comunicación y una variable `conversion` para calcular el valor necesario para escribir en el registro del motor. A continuación, verifica si el valor proporcionado para el límite PWM está dentro del rango válido (0-100 %). Si el valor es válido, calcula el valor correspondiente para escribir en el registro del motor utilizando la conversión inversa necesaria.

Luego, utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor del límite PWM en el registro correspondiente del motor. Si la comunicación es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo límite PWM establecido. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

En resumen, este método permite cambiar el límite de ancho de pulso modulado (PWM) del motor Dynamixel y gestiona adecuadamente los posibles errores de comunicación.

- **float dynamixelMotor::getCurrentLimit():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el límite de corriente actual del motor Dynamixel. No toma ningún parámetro.

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación y una variable `conversion` para calcular el valor en mA a partir del valor leído en el registro del motor. Luego, utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer el límite de corriente actual del motor desde el registro correspondiente.

Si la comunicación es exitosa (`COMM_SUCCESS`), calcula el valor de la corriente actual en mA utilizando la conversión necesaria y registra un mensaje informativo utilizando `ROS_INFO`, indicando el límite de corriente actual del motor Dynamixel. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

En resumen, este método permite obtener el límite de corriente actual del motor Dynamixel y gestiona adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setCurrentLimit(float CURRENT_mA):** Este método es un modificador de la clase `dynamixelMotor` que establece el límite de corriente actual del motor Dynamixel. Toma un parámetro `CURRENT_mA`, que representa el nuevo límite de corriente en miliamperios.

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación y una variable `conversion` para calcular el valor que se escribirá en el registro del motor a partir del valor en mA proporcionado.

Luego, verifica si el valor de la corriente proporcionado está dentro del rango permitido (0 a 3200 mA). Si es así, utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo límite de corriente en el registro correspondiente del motor.

Si la comunicación es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo límite de corriente establecido para el motor Dynamixel. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

En resumen, este método permite establecer el límite de corriente actual del motor Dynamixel y gestiona adecuadamente los posibles errores de comunicación.

- **float dynamixelMotor::getVelLimit():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el límite de velocidad actual del motor Dynamixel. No toma parámetros y devuelve el límite de velocidad en revoluciones por minuto (rpm).

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación, una variable `data` para almacenar los datos leídos del registro del motor y una variable `conversion` para calcular el valor en rpm a partir del valor leído del registro.

Luego, utiliza el método `read4ByteTxRx` del paquete de comunicación Dynamixel para leer el límite de velocidad actual del motor desde el registro correspondiente. Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), calcula el límite de velocidad en rpm utilizando el valor leído y la conversión adecuada.

Finalmente, registra un mensaje informativo utilizando `ROS_INFO`, indicando el límite de velocidad actual establecido para el motor Dynamixel. Si la comunicación con el motor falla, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

En resumen, este método permite obtener el límite de velocidad actual del motor Dynamixel en rpm y maneja adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setVelLimit(float VEL_LIMIT_RPM):** Este método es un modificador de la clase `dynamixelMotor` que establece el límite de velocidad para el motor Dynamixel. Toma un parámetro `VEL_LIMIT_RPM` que representa el límite de velocidad deseado en revoluciones por minuto (rpm).

Comienza declarando una variable `dxl_error` para almacenar posibles errores de comunicación y una variable `conversion` para convertir el valor de rpm a la unidad utilizada por el motor.

Luego, verifica si el valor de `VEL_LIMIT_RPM` está dentro del rango permitido (0 a 234 rpm). Si el valor es válido, utiliza el método `write4ByteTxRx` del manejador de paquetes Dynamixel para escribir el nuevo límite de velocidad en el registro correspondiente de la tabla de control del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo límite de velocidad establecido. Si la comunicación falla, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error.

Si el valor de `VEL_LIMIT_RPM` está fuera del rango permitido, registra un mensaje de error indicando que el valor especificado no es válido.

En resumen, este método permite establecer el límite de velocidad para el motor Dynamixel, asegurando que el valor proporcionado esté dentro de un rango válido y manejando adecuadamente los posibles errores de comunicación.

- **float dynamixelMotor::getMaxPosLimit():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el límite máximo de posición actual del motor Dynamixel.

Comienza declarando una variable `dxl_error` para almacenar posibles errores de comunicación, una variable `data` para almacenar los datos leídos del registro del motor y una variable `conversion` para calcular el valor en grados a partir del valor leído del registro.

Luego, utiliza el método `read4ByteTxRx` del paquete de comunicación Dynamixel para leer el límite máximo de posición actual del motor desde el registro correspondiente. Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), calcula el límite máximo de posición en grados utilizando el valor leído y la conversión adecuada.

Finalmente, registra un mensaje informativo utilizando `ROS_INFO`, indicando el límite máximo de posición actual establecido para el motor Dynamixel. Si la comunicación con el motor falla, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

En resumen, este método permite obtener el límite máximo de posición actual del motor Dynamixel en grados y maneja adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setMaxPosLimit(float MAX_POS_LIMIT_DEGREES):** Este método establece el límite máximo de posición del motor Dynamixel en grados.

Comienza declarando una variable `dxl_error` para almacenar posibles errores de comunicación y una variable `conversion` para convertir el valor en grados a unidades del registro del motor.

El método verifica que el valor proporcionado `MAX_POS_LIMIT_DEGREES` esté dentro del rango permitido (0 a 360 grados). Si el valor está dentro de este rango, utiliza el método `write4ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo límite máximo de posición en el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el nuevo límite máximo de posición establecido para el motor Dynamixel. Si la comunicación falla, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Si el valor proporcionado está fuera del rango permitido, registra un mensaje de error utilizando `ROS_ERROR`, indicando que se debe especificar un límite máximo de posición válido.

En resumen, este método permite establecer el límite máximo de posición del motor Dynamixel en grados y maneja adecuadamente los posibles errores de comunicación y de validación de valores.

- **float dynamixelMotor::getMinPosLimit():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el límite inferior de posición del motor Dynamixel. No toma parámetros y devuelve el límite en grados.

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación, una variable `data` para almacenar los datos leídos del registro del motor y una variable `conversion` para calcular el valor en grados a partir del valor leído del registro.

Luego, utiliza el método `read4ByteTxRx` del paquete de comunicación Dynamixel para leer el límite mínimo de posición actual del motor desde el registro correspondiente. Si la comunicación con el motor no es exitosa (`COMM_SUCCESS`), registra un mensaje de error utilizando `ROS_ERROR`, indicando el código de error proporcionado por el motor y devuelve un `-1`.

Si la comunicación es exitosa, calcula el límite mínimo de posición utilizando el valor leído y la conversión adecuada. Registra un mensaje informativo utilizando `ROS_INFO`, indicando el límite inferior de posición establecido para el motor Dynamixel.

Finalmente, el método devuelve el límite mínimo de posición en grados y maneja adecuadamente la liberación de memoria de la variable `data`.

En resumen, este método permite obtener el límite mínimo de posición actual del motor Dynamixel en grados y maneja adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setMinPosLimit(float MIN_POS_LIMIT_DEGREES):** Este método establece el límite mínimo de posición del motor Dynamixel. Toma como parámetro un valor flotante que representa el nuevo límite en grados y no devuelve ningún valor.

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación y una variable `conversion` para convertir el valor de grados a la unidad utilizada internamente por el registro del motor.

Verifica si el valor del límite proporcionado (MIN_POS_LIMIT_DEGREES) está dentro del rango válido de 0 a 360 grados. Si el valor está fuera de este rango, registra un mensaje de error utilizando ROS_ERROR y no realiza ninguna acción adicional.

Si el valor está dentro del rango válido, utiliza el método write4ByteTxRx del paquete de comunicación Dynamixel para escribir el nuevo límite mínimo de posición en el registro correspondiente del motor. Si la comunicación con el motor no es exitosa (COMM_SUCCESS), registra un mensaje de error utilizando ROS_ERROR, indicando el código de error proporcionado por el motor.

Si la comunicación es exitosa, registra un mensaje informativo utilizando ROS_INFO, indicando que el límite mínimo de posición ha sido cambiado correctamente al nuevo valor especificado en grados.

En resumen, este método permite establecer el límite mínimo de posición del motor Dynamixel en grados, asegurándose de que el valor proporcionado sea válido y manejando adecuadamente los posibles errores de comunicación.

- **bool dynamixelMotor::getTorqueState():** Este método es un accesor de la clase dynamixelMotor que devuelve el estado actual del torque del motor Dynamixel. No toma parámetros y devuelve un valor booleano que indica si el torque está habilitado (true) o deshabilitado (false).

Comienza declarando una variable dxl_error para almacenar errores de comunicación y una variable data para almacenar el dato leído del registro del motor.

Luego, utiliza el método read1ByteTxRx del paquete de comunicación Dynamixel para leer el estado del torque del motor desde el registro correspondiente. Si la comunicación con el motor no es exitosa (COMM_SUCCESS), registra un mensaje de error utilizando ROS_ERROR, indicando el código de error proporcionado por el motor y devuelve -1.

Si la comunicación es exitosa, determina el estado del torque a partir del valor leído. Si el valor leído es 0, el estado del torque es false (OFF). Si el valor leído es diferente de 0, el estado del torque es true (ON).

Registra un mensaje informativo utilizando ROS_INFO, indicando el estado actual del torque del motor Dynamixel. Finalmente, devuelve el estado del torque.

En resumen, este método permite obtener el estado actual del torque del motor Dynamixel y maneja adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setTorqueState(bool TORQUE_ENABLE):** Este método establece el estado del torque del motor Dynamixel. Toma como parámetro un valor booleano TORQUE_ENABLE que indica si el torque debe ser habilitado (true) o deshabilitado (false) y no devuelve ningún valor.

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación y una variable `s_torque_state` que almacena una cadena de texto que representa el estado del torque (“ON” o “OFF”) según el valor de `TORQUE_ENABLE`.

Utiliza el método `write1ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo estado del torque en el registro correspondiente del motor. Si la comunicación con el motor no es exitosa (`COMM_SUCCESS`), registra un mensaje de error utilizando `ROS_ERROR`, indicando el código de error proporcionado por el motor.

Si la comunicación es exitosa, registra un mensaje informativo utilizando `ROS_INFO`, indicando que el estado del torque ha sido cambiado correctamente al nuevo valor especificado.

En resumen, este método permite establecer el estado del torque del motor Dynamixel, manejando adecuadamente los posibles errores de comunicación y confirmando el cambio de estado mediante un mensaje informativo.

- **bool dynamixelMotor::getLedState():** Este método es un accesor de la clase `dynamixelMotor` que devuelve el estado actual del LED del motor Dynamixel. No toma parámetros y devuelve un valor booleano que indica si el LED está encendido (`true`) o apagado (`false`).

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación y una variable `data` para almacenar el dato leído del registro del motor.

Luego, utiliza el método `read1ByteTxRx` del paquete de comunicación Dynamixel para leer el estado del LED del motor desde el registro correspondiente. Si la comunicación con el motor no es exitosa (`COMM_SUCCESS`), registra un mensaje de error utilizando `ROS_ERROR`, indicando el código de error proporcionado por el motor y devuelve -1.

Si la comunicación es exitosa, determina el estado del LED a partir del valor leído. Si el valor leído es 0, el estado del LED es `false` (OFF). Si el valor leído es diferente de 0, el estado del LED es `true` (ON).

Registra un mensaje informativo utilizando `ROS_INFO`, indicando el estado actual del LED del motor Dynamixel. Finalmente, devuelve el estado del LED.

En resumen, este método permite obtener el estado actual del LED del motor Dynamixel y maneja adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setLedState(bool LED_STATE):** Este método establece el estado del LED del motor Dynamixel. Toma como parámetro un valor booleano `LED_STATE` que indica si el LED debe ser encendido (`true`) o apagado (`false`) y no devuelve ningún valor.

Comienza declarando una variable `dxl_error` para almacenar errores de comunicación y una variable `s_led_state` que almacena una cadena de texto que representa el estado del LED (“ON” o “OFF”) según el valor de `LED_STATE`.

Utiliza el método `write1ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo estado del LED en el registro correspondiente del motor. Si la comunicación con el motor no es exitosa (`COMM_SUCCESS`), registra un mensaje de error utilizando `ROS_ERROR`, indicando el código de error proporcionado por el motor.

Si la comunicación es exitosa, registra un mensaje informativo utilizando `ROS_INFO`, indicando que el estado del LED ha sido cambiado correctamente al nuevo valor especificado.

En resumen, este método permite establecer el estado del LED del motor Dynamixel, manejando adecuadamente los posibles errores de comunicación y confirmando el cambio de estado mediante un mensaje informativo.

- **int dynamixelMotor::getStatusReturnLevel():** Este método es un accesor de la clase `dynamixelMotor` que permite obtener el nivel de retorno de estado del motor Dynamixel. El nivel de retorno de estado es un parámetro que define a qué tipos de peticiones se les responderá con un paquete de estado (Status Packet).

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`) y un puntero a un array de `uint8_t` para almacenar los datos leídos (`data`).

A continuación, utiliza el método `read1ByteTxRx` del paquete de comunicación Dynamixel para leer un byte de datos del registro correspondiente al nivel de retorno de estado en la tabla de control del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["STATUS_RETURN_PACKET"]`), el puntero al array de datos (`data`) y el puntero a la variable de errores (`&dxl_error`).

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. El método devuelve -1 en este caso.

Si la comunicación es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, mostrando el nivel de retorno de estado actual del motor. El método devuelve el valor leído, convertido a entero.

Finalmente, se libera la memoria asignada al array de datos utilizando `delete []`.

En resumen, este método permite obtener el nivel de retorno de estado del motor Dynamixel, manejando adecuadamente los posibles errores de comunicación y asegurando la correcta liberación de memoria.

- **void dynamixelMotor::setStatusReturnLevel(int STATUS_RETURN_LEVEL):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el nivel de retorno de estado del motor Dynamixel. Los valores posibles para este parámetro son:

- **0:** Retorna el paquete de estado solo para la instrucción PING.
- **1:** Retorna el paquete de estado para las instrucciones PING y READ.
- **2:** Retorna el paquete de estado para todas las instrucciones.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`).

A continuación, utiliza el método `write1ByteTxRx` del paquete de comunicación Dynamixel para escribir un byte de datos en el registro correspondiente al nivel de retorno de estado en la tabla de control del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["STATUS_RETURN_LEVEL"]`), el nuevo valor del nivel de retorno de estado (`STATUS_RETURN_LEVEL`) y el puntero a la variable de errores (`&dxl_error`).

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error.

Si la comunicación es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, mostrando el nuevo nivel de retorno de estado del motor.

En resumen, este método permite cambiar el nivel de retorno de estado del motor Dynamixel, permitiendo al usuario definir cómo y cuándo el motor debe responder a las instrucciones recibidas, y manejando adecuadamente los posibles errores de comunicación.

- **`std::vector<bool> dynamixelMotor::getHardwareErrorStatus()`:** Este método permite obtener el estado de los errores de hardware del motor Dynamixel. Devuelve un vector de booleanos que representa los diferentes estados de error de hardware, donde cada bit del byte leído corresponde a un error específico.

El método comienza inicializando un vector de booleanos `errorStatus` con 8 elementos. También declara una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`) y otra variable de tipo `uint8_t` para almacenar los datos leídos del motor (`data`).

Utiliza el método `read1ByteTxRx` del paquete de comunicación Dynamixel para leer un byte de datos del registro correspondiente al estado de error de hardware en la tabla de control del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["HARDWARE_ERROR_STATUS"]`), el puntero a la variable de datos (`data`) y el puntero a la variable de errores (`&dxl_error`).

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error.

Si la comunicación es exitosa, se procesa el byte leído (`data[0]`) para descomponerlo en bits individuales y almacenarlos en el vector `errorStatus`. Cada bit del byte representa un estado de error específico:

- **Bit 0:** Error de Voltaje de Entrada.
- **Bit 1, 6 y 7:** No utilizados.
- **Bit 2:** Error de Sobrecaleamiento.
- **Bit 3:** Error del Codificador del Motor.
- **Bit 4:** Error de Choque Eléctrico.
- **Bit 5:** Error de Sobrecarga.

Los valores de estos bits se guardan en el vector `errorStatus` y se muestran en la consola utilizando `ROS_INFO`.

En resumen, este método permite obtener y desglosar el estado de error de hardware del motor Dynamixel, proporcionando una representación detallada de los posibles errores de hardware y manejando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::getVelocityPIValues(int &P, int &I):** Este método es un modificador de la clase `dynamixelMotor` que permite obtener los valores de los componentes proporcionales (P) e integrales (I) del controlador de velocidad del motor Dynamixel. Los valores obtenidos se almacenan en las referencias de enteros proporcionadas como parámetros (P y I).

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`) y una variable de tipo `uint16_t` para almacenar los datos leídos del motor (`data`).

Primero, se utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer dos bytes de datos del registro correspondiente al componente integral (I) del controlador de velocidad en la tabla de control del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["VELOCITY_I_GAIN"]`), el puntero a la variable de datos (`data`) y el puntero a la variable de errores (`&dxl_error`).

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. Si la comunicación es exitosa, se guarda el valor leído en la referencia proporcionada para el componente I (I) y se registra un mensaje informativo utilizando `ROS_INFO`.

A continuación, se repite el mismo procedimiento para leer el valor del componente proporcional (P) del controlador de velocidad.

Si la comunicación con el motor no es exitosa, se registra un mensaje de error similar al anterior. Si la comunicación es exitosa, se guarda el valor leído en la referencia proporcionada para el componente P (P) y se registra un mensaje informativo.

En resumen, este método permite obtener los valores actuales de los componentes proporcional e integral del controlador de velocidad del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y almacenando los valores en las referencias proporcionadas.

- **void dynamixelMotor::setVelocityPIValues(int P, int I):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer los valores de los componentes proporcional (P) e integral (I) del controlador de velocidad del motor Dynamixel. Toma dos parámetros de tipo entero que representan los nuevos valores para los componentes P e I.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`).

Primero, el método verifica si el valor proporcionado para el componente P está dentro del rango válido (0 a 16383). Si el valor está dentro del rango, se utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir dos bytes de datos en el registro correspondiente al componente proporcional (P) del controlador de velocidad en la tabla de control del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["VELOCITY_P_GAIN"]`), el valor a escribir (P) y el puntero a la variable de errores (`&dxl_error`).

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. Si la comunicación es exitosa, se registra un mensaje informativo utilizando `ROS_INFO` indicando que el valor del componente P se ha establecido correctamente. Si el valor proporcionado está fuera del rango permitido, se registra un mensaje de error indicando que se debe especificar un valor válido para el componente P del controlador de velocidad.

A continuación, se repite el mismo procedimiento para el valor del componente integral (I). El método verifica si el valor proporcionado para el componente I está dentro del rango válido (0 a 16383). Si el valor está dentro del rango, se utiliza el método `write2ByteTxRx` con la dirección del registro correspondiente en la tabla de control (`this->CONTROL_TABLE["VELOCITY_I_GAIN"]`) y el valor a escribir (I).

Si la comunicación con el motor no es exitosa, se registra un mensaje de error similar al anterior. Si la comunicación es exitosa, se registra un mensaje informativo indicando que el valor del componente I se ha establecido correctamente. Si el valor proporcionado está fuera del rango permitido, se registra un mensaje de error indicando que se debe especificar un valor válido para el componente I del controlador de velocidad.

En resumen, este método permite establecer los valores de los componentes proporcional e integral del controlador de velocidad del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::getPositionPIDValues(int &P, int &I, int &D):** Este método es un modificador de la clase dynamixelMotor que permite obtener los valores actuales de los componentes proporcional (P), integral (I) y derivativo (D) del controlador de posición del motor Dynamixel. Toma tres referencias a enteros como parámetros, los cuales serán actualizados con los valores correspondientes.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`) y un puntero a `uint16_t` para almacenar los datos leídos (`data`).

Primero, se utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer dos bytes de datos del registro correspondiente al componente proporcional (P) del controlador de posición en la tabla de control del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["POSITION_P_GAIN"]`), el puntero al almacenamiento de datos (`data`) y el puntero a la variable de errores (`&dxl_error`).

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. Si la comunicación es exitosa, se registra un mensaje informativo utilizando `ROS_INFO` indicando que el valor del componente P actual ha sido leído correctamente y actualiza la referencia del parámetro P con el valor leído.

A continuación, se repite el mismo procedimiento para los componentes integral (I) y derivativo (D) del controlador de posición. Para cada componente, el método verifica si la lectura del registro correspondiente en la tabla de control es exitosa. En caso de fallo, se registra un mensaje de error indicando el fallo y el código de error.

Finalmente, se libera la memoria asignada para almacenar los datos (`delete [] data`).

En resumen, este método permite obtener los valores actuales de los componentes proporcional, integral y derivativo del controlador de posición del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setPositionPIDValues(int P, int I, int D):** Este método permite establecer los valores de los componentes proporcional (P), integral (I) y derivativo (D) del controlador de posición del motor Dynamixel. Toma tres parámetros enteros que representan los nuevos valores de los componentes P, I y D, respectivamente.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`).

Para cada componente (P, I y D), el método verifica si el valor proporcionado está dentro del rango permitido (0 - 16383). Si el valor está dentro del rango, utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir el valor en el registro correspondiente de la tabla de control del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["POSITION_P_GAIN"]`, `this->CONTROL_TABLE["POSITION_I_GAIN"]` o `this->CONTROL_TABLE["POSITION_D_GAIN"]`), el valor a escribir (P, I o D) y el puntero a la variable de errores (`&dxl_error`).

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. Si la comunicación es exitosa, se registra un mensaje informativo utilizando `ROS_INFO` indicando que el valor del componente P, I o D ha sido cambiado correctamente.

Si el valor proporcionado está fuera del rango permitido, se registra un mensaje de error utilizando `ROS_ERROR` indicando que se debe especificar un valor válido para el componente del controlador de posición.

En resumen, este método permite cambiar los valores de los componentes proporcional, integral y derivativo del controlador de posición del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::getFeedforwardGains(int &FFG1, int &FFG2):** Este método permite obtener los valores de los componentes de ganancia de alimentación directa (Feedforward) del motor Dynamixel. Toma dos parámetros de referencia enteros que representan los valores de los componentes de ganancia de alimentación directa FFG1 y FFG2, respectivamente.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`). Luego, utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer los valores de los componentes de ganancia de alimentación directa del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["FEEDFORWARD_1st_GAIN"]` o `this->CONTROL_TABLE["FEEDFORWARD_2nd_GAIN"]`), y los punteros a las variables de los componentes FFG1 y FFG2 (`&FFG1` y `&FFG2`) respectivamente.

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. Si la comunicación es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, indicando que los valores de los componentes FFG1 y FFG2 han sido obtenidos correctamente.

Finalmente, se liberan los recursos de memoria asignados para la variable `data`.

En resumen, este método permite obtener los valores de los componentes de ganancia de alimentación directa del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setFeedforwardGains(int FFG1, int FFG2):** Este método permite establecer los valores de los componentes de ganancia de alimentación directa (Feedforward) del motor Dynamixel. Toma dos parámetros enteros que representan los valores de los componentes de ganancia de alimentación directa FFG1 y FFG2, respectivamente.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`). Luego, verifica si los valores proporcionados para FFG1 y FFG2 están dentro del rango permitido (0 - 16.383). Si ambos valores son válidos, utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir los nuevos valores de los componentes de ganancia de alimentación directa del motor. Este método requiere el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro en la tabla de control (`this->CONTROL_TABLE["FEEDFORWARD_1st_GAIN"]` o `this->CONTROL_TABLE["FEEDFORWARD_2nd_GAIN"]`), y los valores de los componentes FFG1 y FFG2.

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. Si la comunicación es exitosa, se registra un mensaje informativo utilizando `ROS_INFO`, indicando que los valores de los componentes FFG1 y FFG2 han sido cambiados correctamente.

En resumen, este método permite establecer los valores de los componentes de ganancia de alimentación directa del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **int dynamixelMotor::getBusWatchdog():** Este método es un accesor de la clase `dynamixelMotor` que permite obtener el estado del Bus Watchdog del motor Dynamixel. No toma ningún parámetro.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`), y un puntero a un array de tipo `uint8_t` para almacenar los datos leídos del motor. Luego, utiliza el método `read1ByteTxRx` del paquete de comunicación Dynamixel para leer el valor del Bus Watchdog desde el registro correspondiente del motor.

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error. En este caso, el método devuelve -1 para indicar que ha ocurrido un error.

Si la comunicación es exitosa, el método interpreta el valor leído del Bus Watchdog. Si el valor es 0, indica que la función del Bus Watchdog está desactivada y se registra un mensaje informativo utilizando `ROS_INFO`. Si el valor está en el rango de 1 a 127, se interpreta como el

tiempo de activación del Bus Watchdog en milisegundos y se calcula el tiempo correspondiente. Luego, se registra un mensaje informativo indicando el valor del Bus Watchdog y su tiempo correspondiente en milisegundos. Finalmente, el método devuelve el valor del Bus Watchdog. En resumen, este método permite obtener el estado del Bus Watchdog del motor Dynamixel, interpretando su valor y proporcionando información sobre su estado y configuración.

- **void dynamixelMotor::setBusWatchdog(int BUS_WATCHDOG_VALUE):** Este método es un modificador de la clase `dynamixelMotor` que permite configurar el estado del Bus Watchdog del motor Dynamixel. Toma un parámetro de tipo entero (`BUS_WATCHDOG_VALUE`) que representa el nuevo valor del Bus Watchdog.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`). Luego, utiliza el método `write1ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor del Bus Watchdog en el registro correspondiente del motor.

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error.

Si la comunicación es exitosa, el método interpreta el nuevo valor del Bus Watchdog. Si el valor es 0, indica que se desactiva la función del Bus Watchdog y se registra un mensaje informativo utilizando `ROS_INFO`. Si el valor es distinto de 0, se calcula el tiempo correspondiente en milisegundos y se registra un mensaje informativo indicando el nuevo valor del Bus Watchdog y su tiempo correspondiente en milisegundos.

En resumen, este método permite configurar el estado del Bus Watchdog del motor Dynamixel, activándolo o desactivándolo según el valor proporcionado.

- **int dynamixelMotor::getGoalPWM():** Este método es un accesario de la clase `dynamixelMotor` que permite obtener el valor consigna del ancho de pulso modulado (PWM) del motor Dynamixel. Devuelve un valor entero que representa el valor en porcentaje.

El método comienza declarando una variable de tipo `uint8_t` para almacenar posibles errores de comunicación (`dxl_error`) y una variable de tipo `float` (`conversion_to_percent`) que se utiliza para convertir el valor del PWM a porcentaje. Luego, utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer el valor objetivo del PWM del motor.

Si la comunicación con el motor no es exitosa (`dxl_comm_result != COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el fallo y el código de error, y se devuelve -1.

Si la comunicación es exitosa, el método calcula el valor del PWM en porcentaje utilizando una conversión y registra un mensaje informativo utilizando `ROS_INFO`, indicando el valor objetivo del PWM en unidades y en porcentaje.

En resumen, este método permite obtener el valor objetivo del ancho de pulso modulado (PWM) del motor Dynamixel en porcentaje.

- **void dynamixelMotor::setGoalPWM(int GOAL_PWM):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el valor PWM (Pulse Width Modulation) consigna del motor Dynamixel.

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra variable de tipo flotante (`conversion`) que se utiliza para convertir el valor de PWM proporcionado al valor necesario para escribir en el registro del motor.

Primero, el método verifica si el valor de PWM proporcionado está dentro del límite permitido llamando al método `getPWMLimit()` de la clase. Si el valor es válido, realiza la conversión necesaria de unidades y utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor de PWM objetivo en el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando que el valor de PWM objetivo ha sido cambiado correctamente. El mensaje incluye tanto el valor convertido como el valor original en porcentaje.

En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Si el valor proporcionado está fuera del rango permitido, registra un mensaje de error indicando que se debe especificar un valor de PWM objetivo válido que sea menor que el límite de PWM.

En resumen, este método permite cambiar el valor de PWM objetivo del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **int dynamixelMotor::getGoalCurrent():** Este método permite obtener el valor actual de la consigna de corriente del motor Dynamixel. No toma ningún parámetro y devuelve un entero que representa la corriente en miliamperios (mA).

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra variable de tipo puntero a `uint16_t` para almacenar los datos leídos. También declara una variable de tipo flotante (`conversion_to_mA`) que se utiliza para convertir el valor leído a miliamperios.

A continuación, utiliza el método `read2ByteTxRx` del paquete de comunicación Dynamixel para leer el valor actual de la corriente objetivo desde el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), el método convierte el valor leído a miliamperios utilizando la constante de conversión y registra un mensaje informativo

utilizando `ROS_INFO`, indicando el valor de la corriente objetivo en miliamperios. Finalmente, devuelve el valor convertido.

En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor, y devuelve -1.

En resumen, este método permite leer el valor actual de la corriente objetivo del motor Dynamixel, asegurando una conversión adecuada de las unidades y gestionando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setGoalCurrent(int GOAL_CURRENT):** Este método permite establecer el valor de la consigna de corriente del motor Dynamixel. Toma un parámetro de tipo entero que representa el nuevo valor objetivo de corriente en miliamperios (mA).

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra variable de tipo flotante (`conversion`) que se utiliza para convertir el valor de corriente proporcionado a la unidad necesaria para escribir en el registro del motor.

A continuación, verifica si el valor de corriente proporcionado está dentro del límite permitido utilizando el método `getCurrentLimit()` de la clase.

Si el valor proporcionado es válido, realiza una conversión necesaria de unidades y utiliza el método `write2ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor de corriente objetivo en el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando que el valor de corriente objetivo ha sido cambiado correctamente. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Si el valor proporcionado está fuera del límite permitido, registra un mensaje de error indicando que se debe especificar un valor de corriente objetivo válido.

En resumen, este método permite cambiar el valor de corriente objetivo del motor Dynamixel, asegurando que se cumplan las restricciones de límite y gestionando adecuadamente los posibles errores de comunicación.

- **double dynamixelMotor::getGoalVelocity():** Este método es un accesor de la clase `dynamixelMotor` que permite obtener el valor de la consigna de velocidad del motor Dynamixel. Devuelve un valor de tipo `double` que representa la velocidad objetivo en revoluciones por minuto (rpm).

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra variable de tipo puntero a `uint32_t` para almacenar el dato leído del registro del motor. Además, se define una constante de tipo flotante (`conversion_to_revmin`) que se utiliza para convertir el valor leído a rpm.

A continuación, utiliza el método `read4ByteTxRx` del paquete de comunicación Dynamixel para leer el valor de la velocidad objetivo desde el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), convierte el valor leído a rpm utilizando la constante de conversión y registra un mensaje informativo utilizando `ROS_INFO`, indicando el valor de la velocidad objetivo actual. Luego, devuelve el valor de la velocidad objetivo en rpm. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor, y devuelve -1.

Finalmente, se libera la memoria asignada para el puntero `data` utilizando `delete[]`.

En resumen, este método permite obtener el valor de velocidad objetivo del motor Dynamixel en revoluciones por minuto, gestionando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setGoalVelocity(double GOAL_VELOCITY):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el valor objetivo de velocidad del motor Dynamixel. Toma un parámetro de tipo `double` que representa la nueva velocidad objetivo en revoluciones por minuto (rpm).

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra variable de tipo flotante (`conversion`) que se utilizará para convertir el valor de velocidad proporcionado a la unidad necesaria para escribir en el registro del motor.

Luego, verifica si el valor de velocidad proporcionado está dentro del rango permitido (inferior o igual al límite de velocidad).

Si el valor proporcionado es válido, realiza una conversión necesaria de unidades y utiliza el método `write4ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor de la velocidad objetivo en el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando que la velocidad objetivo ha sido cambiada correctamente. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Si el valor proporcionado está fuera del rango permitido, registra un mensaje de error indicando que se debe especificar una velocidad objetivo válida.

En resumen, este método permite cambiar la velocidad objetivo del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **double dynamixelMotor::getProfileAcceleration():** Este método es un accesor de la clase `dynamixelMotor` que permite obtener el valor de la aceleración del perfil del motor Dynamixel. Devuelve un valor de tipo doble que representa la aceleración del perfil.

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra variable de tipo puntero a `uint32_t` para almacenar el valor leído de la memoria del motor.

A continuación, utiliza el método `read4ByteTxRx` del paquete de comunicación Dynamixel para leer el valor de aceleración del perfil desde el registro correspondiente.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el valor actual de la aceleración del perfil. Luego, devuelve este valor convertido a tipo doble. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor, y devuelve -1.

Finalmente, libera la memoria asignada para la variable de datos.

En resumen, este método permite obtener el valor actual de la aceleración del perfil del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y asegurando la liberación de la memoria utilizada.

- **void dynamixelMotor::setProfileAcceleration(double PROFILE_ACCELERATION):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el valor de la aceleración del perfil del motor Dynamixel. Toma un parámetro de tipo doble que representa el nuevo valor de la aceleración del perfil.

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación. Luego, verifica si el valor proporcionado para la aceleración del perfil está dentro del rango permitido (0 a 32767).

Si el valor proporcionado es válido, utiliza el método `write4ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor de la aceleración del perfil en el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando que el valor de la aceleración del perfil ha sido cambiado correctamente. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Si el valor proporcionado está fuera del rango permitido, registra un mensaje de error indicando que se debe especificar un valor de aceleración del perfil válido.

En resumen, este método permite cambiar el valor de la aceleración del perfil del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **double dynamixelMotor::getProfileVelocity():** Este método permite obtener el valor de la velocidad del perfil del motor Dynamixel. Retorna un valor de tipo doble que representa la velocidad del perfil.

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación y otra de tipo `uint32_t*` para almacenar los datos leídos.

Luego, utiliza el método `read4ByteTxRx` del paquete de comunicación Dynamixel para leer el valor de la velocidad del perfil desde el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando el valor actual de la velocidad del perfil. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Finalmente, retorna el valor de la velocidad del perfil como un `double`.

En resumen, este método permite obtener el valor de la velocidad del perfil del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setProfileVelocity(double PROFILE_VELOCITY):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer el valor de la velocidad del perfil del motor Dynamixel. Toma un parámetro de tipo doble que representa la nueva velocidad del perfil.

El método comienza declarando una variable de tipo `uint8_t` para almacenar errores de comunicación.

Luego, verifica si el valor de la velocidad proporcionado está dentro del rango permitido (entre 0 y 32767).

Si el valor proporcionado es válido, utiliza el método `write4ByteTxRx` del paquete de comunicación Dynamixel para escribir el nuevo valor de la velocidad del perfil en el registro correspondiente del motor.

Si la comunicación con el motor es exitosa (`COMM_SUCCESS`), registra un mensaje informativo utilizando `ROS_INFO`, indicando que la velocidad del perfil ha sido cambiada correctamente. En caso de fallo de comunicación, registra un mensaje de error utilizando `ROS_ERROR`, junto con el código de error proporcionado por el motor.

Si el valor proporcionado está fuera del rango permitido, registra un mensaje de error indicando que se debe especificar una velocidad del perfil válida.

En resumen, este método permite cambiar la velocidad del perfil del motor Dynamixel, asegurando que se cumplan las restricciones de rango y gestionando adecuadamente los posibles errores de comunicación.

- **double dynamixelMotor::getGoalPosition():** Este método es un accesor de la clase `dynamixelMotor` que permite obtener la posición objetivo actual del motor Dynamixel en grados. No toma parámetros y devuelve un valor de tipo `double` que representa la consigna de posición en grados.

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero de tipo `uint32_t` (`data`) para almacenar los datos leídos del registro del motor. También se declara una constante de tipo `float` (`conversion`) con el valor `0.088`, que se utilizará para convertir el valor de la posición proporcionado por el motor en grados.

A continuación, el método utiliza la función `read4ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 4 bytes de datos desde el registro de la posición objetivo (`GOAL_POSITION`) del motor, especificado por el identificador de la tabla de control (`CONTROL_TABLE["GOAL_POSITION"]`). Esta función toma como parámetros el manejador del puerto (`myPortHandler`), el identificador del motor (`this->ID`), la dirección del registro y un puntero para almacenar los datos leídos, así como un puntero para almacenar posibles errores de comunicación (`dxl_error`).

Si la comunicación con el motor falla (es decir, si el resultado de `dxl_comm_result` no es `COMM_SUCCESS`), se registra un mensaje de error utilizando `ROS_ERROR`, indicando el identificador del motor y el código de error, y el método retorna `-1`.

Si la lectura es exitosa, el valor leído (almacenado en `data`) se convierte a grados multiplicándolo por la constante de conversión (`conversion`). El valor resultante se almacena en la variable `degrees`. Se registra un mensaje informativo utilizando `ROS_INFO`, indicando el identificador del motor y la posición objetivo en grados, y finalmente, el método retorna el valor convertido (`degrees`).

Al final del método, se libera la memoria asignada para el puntero `data` utilizando `delete []`.

En resumen, este método permite obtener la posición objetivo actual del motor Dynamixel en grados, asegurando que se gestione correctamente la comunicación con el motor y se manejen adecuadamente los posibles errores de comunicación.

- **void dynamixelMotor::setGoalPosition(double GOAL_POSITION):** Este método es un modificador de la clase `dynamixelMotor` que permite establecer la posición objetivo del motor Dynamixel en grados o revoluciones, dependiendo del modo de operación. Toma un parámetro de tipo `double` que representa la nueva posición objetivo.

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación. Luego, se verifica el modo de operación actual del motor utilizando el método `getOperatingMode()`.

Si el modo de operación es "Position Control", se verifica si la posición objetivo está dentro del rango permitido (0 a 360 grados). Si el valor proporcionado es válido, se utiliza la función `write4ByteTxRx` del manejador de paquetes (`myPacketHandler`) para escribir 4 bytes de datos en el registro de la posición objetivo (`GOAL_POSITION`) del motor. La posición se convierte de grados a unidades internas del motor dividiendo el valor proporcionado por

0.088. Si la comunicación es exitosa (COMM_SUCCESS), se registra un mensaje informativo utilizando ROS_INFO indicando la nueva posición objetivo en grados. Si la comunicación falla, se registra un mensaje de error utilizando ROS_ERROR.

Si el modo de operación es "Extended Position Control", se verifica si la posición objetivo está dentro del rango permitido (-256 a 256 revoluciones). Si el valor proporcionado es válido, se utiliza la función write4ByteTxRx del manejador de paquetes para escribir 4 bytes de datos en el registro de la posición objetivo del motor. La posición se convierte de revoluciones a unidades internas del motor multiplicando el valor proporcionado por 4096. Si la comunicación es exitosa, se registra un mensaje informativo indicando la nueva posición objetivo en revoluciones. Si la comunicación falla, se registra un mensaje de error.

Si la posición objetivo proporcionada está fuera del rango permitido en cualquiera de los modos de operación, se registra un mensaje de error indicando que se debe especificar una posición objetivo válida.

En resumen, este método permite cambiar la posición objetivo del motor Dynamixel, asegurando que se cumplan las restricciones de rango según el modo de operación y gestionando adecuadamente los posibles errores de comunicación.

- **double dynamixelMotor::getRealtimeTick():** Este método es un accesor de la clase dynamixelMotor que permite obtener el valor actual del realtime tick del motor Dynamixel. Devuelve un valor de tipo double que representa el realtime tick actual.

El método comienza declarando una variable de tipo uint8_t (dxl_error) para almacenar posibles errores de comunicación y un puntero a un uint16_t (data) para almacenar el valor leído del registro del motor.

Luego, se utiliza la función read2ByteTxRx del manejador de paquetes (myPacketHandler) para leer 2 bytes de datos del registro del realtime tick del motor (REALTIME_TICK) y almacenarlos en data. Si la comunicación falla, se registra un mensaje de error utilizando ROS_ERROR y se devuelve -1. Si la comunicación es exitosa (COMM_SUCCESS), se convierte el valor leído a tipo double, se registra un mensaje informativo utilizando ROS_INFO indicando el valor actual del realtime tick, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a data utilizando delete [] data.

En resumen, este método permite leer y devolver el valor actual del realtime tick del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y liberando la memoria utilizada para almacenar el valor leído.

- **bool dynamixelMotor::isMoving():** Este método es un accesor de la clase dynamixelMotor que permite verificar si el motor Dynamixel se encuentra en movimiento. Devuelve un valor de tipo bool que indica si el motor está en movimiento (true) o no (false).

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un `uint8_t` (`data`) para almacenar el valor leído del registro del motor que indica si está en movimiento (MOVING).

Luego, se utiliza la función `read1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 1 byte de datos del registro de movimiento del motor (MOVING) y almacenarlo en `data`. Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR` y se devuelve -1. Si la comunicación es exitosa (COMM_SUCCESS), se convierte el valor leído a tipo `bool`, se registra un mensaje informativo utilizando `ROS_INFO` indicando el estado de movimiento del motor, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a `data` utilizando `delete[] data`.

En resumen, este método permite verificar si el motor Dynamixel está en movimiento o no, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor booleano que indica el estado de movimiento del motor.

- **int dynamixelMotor::getPresentPWM():** Este método es un accesos de la clase `dynamixelMotor` que permite obtener el valor actual del ancho de pulso modulado (PWM) del motor Dynamixel. Devuelve un valor entero que representa el valor actual del PWM.

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un arreglo de `uint16_t` (`data`) para almacenar el valor leído del registro de PWM del motor (PRESENT_PWM). Además, se declara una constante de conversión (`conversion_to_percent`) para convertir el valor leído a porcentaje.

Luego, se utiliza la función `read2ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 2 bytes de datos del registro de PWM del motor y almacenarlos en `data`. Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR` y se devuelve -1. Si la comunicación es exitosa (COMM_SUCCESS), se realiza la conversión del valor leído a porcentaje, se registra un mensaje informativo utilizando `ROS_INFO` indicando el valor actual del PWM en forma de porcentaje, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a `data` utilizando `delete[] data`.

En resumen, este método permite obtener el valor actual del ancho de pulso modulado (PWM) del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor entero que representa el PWM en forma de porcentaje.

- **int dynamixelMotor::getPresentCurrent():** Este método es un accesos de la clase `dynamixelMotor` que permite obtener el valor actual de corriente del motor Dynamixel. Devuelve un valor entero que representa el valor actual de corriente en miliamperios (mA).

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un arreglo de `uint16_t` (`data`) para

almacenar el valor leído del registro de corriente del motor (PRESENT_CURRENT). Además, se declara una constante de conversión (conversion_to_mA) para convertir el valor leído a miliamperios.

Luego, se utiliza la función `read2ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 2 bytes de datos del registro de corriente del motor y almacenarlos en `data`. Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR` y se devuelve -1. Si la comunicación es exitosa (`COMM_SUCCESS`), se realiza la conversión del valor leído a miliamperios, se registra un mensaje informativo utilizando `ROS_INFO` indicando el valor actual de la corriente en miliamperios, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a `data` utilizando `delete [] data`.

En resumen, este método permite obtener el valor actual de corriente del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor entero que representa la corriente en miliamperios.

- **double dynamixelMotor::getPresentVelocity():** Este método es un accesos de la clase `dynamixelMotor` que permite obtener la velocidad actual del motor Dynamixel. Devuelve un valor decimal que representa la velocidad actual en revoluciones por minuto (rpm).

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un arreglo de `uint32_t` (`data`) para almacenar el valor leído del registro de velocidad del motor (PRESENT_VELOCITY). Además, se declara una constante de conversión (`conversion_to_revmin`) para convertir el valor leído a revoluciones por minuto.

Luego, se utiliza la función `read4ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 4 bytes de datos del registro de velocidad del motor y almacenarlos en `data`. Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR` y se devuelve -1. Si la comunicación es exitosa (`COMM_SUCCESS`), se realiza la conversión del valor leído a revoluciones por minuto, se registra un mensaje informativo utilizando `ROS_INFO` indicando la velocidad actual en rpm, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a `data` utilizando `delete [] data`.

En resumen, este método permite obtener la velocidad actual del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor decimal que representa la velocidad en revoluciones por minuto.

- **double dynamixelMotor::getPresentPosition():** Este método es un accesos de la clase `dynamixelMotor` que permite obtener la posición actual del motor Dynamixel. Devuelve un valor decimal que representa la posición actual en grados.

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un arreglo de `uint32_t` (`data`) para almac-

nar el valor leído del registro de posición del motor (PRESENT_POSITION). Además, se declara una constante de conversión (conversion) para convertir el valor leído a grados.

Luego, se utiliza la función `read4ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 4 bytes de datos del registro de posición del motor y almacenarlos en `data`. Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR` y se devuelve -1. Si la comunicación es exitosa (`COMM_SUCCESS`), se realiza la conversión del valor leído a grados, se registra un mensaje informativo utilizando `ROS_INFO` indicando la posición actual en grados, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a `data` utilizando `delete[] data`.

En resumen, este método permite obtener la posición actual del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor decimal que representa la posición en grados.

- **float dynamixelMotor::getPresentInputV():** Este método es un accesos de la clase `dynamixelMotor` que permite obtener el voltaje de entrada presente del motor Dynamixel. Devuelve un valor decimal que representa el voltaje de entrada actual en voltios (V).

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un arreglo de `uint16_t` (`data`) para almacenar el valor leído del registro de voltaje de entrada del motor (PRESENT_INPUT_VOLTAGE).

Luego, se utiliza la función `read2ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 2 bytes de datos del registro de voltaje de entrada del motor y almacenarlos en `data`. Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR` y se devuelve -1. Si la comunicación es exitosa (`COMM_SUCCESS`), se realiza la conversión del valor leído a voltios y se registra un mensaje informativo utilizando `ROS_INFO` indicando el voltaje de entrada actual en voltios, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a `data` utilizando `delete[] data`.

En resumen, este método permite obtener el voltaje de entrada actual del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor decimal que representa el voltaje de entrada en voltios.

- **int dynamixelMotor::getPresentTemperature():** Este método es un accesos de la clase `dynamixelMotor` que permite obtener la temperatura presente del motor Dynamixel. Devuelve un valor entero que representa la temperatura actual en grados Celsius (°C).

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un byte sin signo (`data`) para almacenar el valor leído del registro de temperatura presente del motor (PRESENT_TEMPERATURE).

Luego, se utiliza la función `read1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 1 byte de datos del registro de temperatura presente del motor y almacenarlos en

data. Si la comunicación falla, se registra un mensaje de error utilizando ROS_ERROR y se devuelve -1. Si la comunicación es exitosa (COMM_SUCCESS), se registra un mensaje informativo utilizando ROS_INFO indicando la temperatura actual en grados Celsius, y se devuelve el valor leído.

Finalmente, se libera la memoria asignada a data utilizando `delete [] data`.

En resumen, este método permite obtener la temperatura actual del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor entero que representa la temperatura en grados Celsius.

- **bool dynamixelMotor::getBackupReady():** Este método es un accesor de la clase dynamixelMotor que permite verificar si existe una copia de seguridad de la tabla de control del motor Dynamixel después de enviar el paquete de copia de seguridad de la tabla de control. Devuelve un valor booleano que indica si existe (verdadero) o no (falso) una copia de seguridad guardada.

El método comienza declarando una variable de tipo `uint8_t` (`dxl_error`) para almacenar posibles errores de comunicación y un puntero a un byte sin signo (`data`) para almacenar el valor leído del registro de copia de seguridad del motor (BACKUP_READY).

Luego, se utiliza la función `read1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer 1 byte de datos del registro de copia de seguridad del motor y almacenarlos en `data`. Si la comunicación falla, se registra un mensaje de error utilizando ROS_ERROR y se devuelve -1. Si la comunicación es exitosa (COMM_SUCCESS), se convierte el valor leído a un valor booleano de acuerdo con la descripción proporcionada (0 indica que la copia de seguridad no existe, 1 indica que existe una copia de seguridad) y se registra un mensaje informativo utilizando ROS_INFO indicando si existe una copia de seguridad guardada.

Finalmente, se libera la memoria asignada a `data` utilizando `delete [] data`.

En resumen, este método permite verificar si existe una copia de seguridad de la tabla de control del motor Dynamixel después de enviar el paquete de copia de seguridad de la tabla de control, gestionando adecuadamente los posibles errores de comunicación y devolviendo un valor booleano que indica la existencia de la copia de seguridad guardada.

- **void dynamixelMotor::configDriveMode(bool REVERSE_MODE, bool SLAVE_MODE, bool TIME_BASED_PROFILE, bool TORQUE_AUTO_ON):** Este método es un modificador de la clase dynamixelMotor que permite configurar el modo de funcionamiento del motor Dynamixel. Toma cuatro parámetros booleanos que indican si se activan o desactivan diferentes modos de operación: modo inverso, modo esclavo, perfil basado en tiempo y activación automática de torque.

El método comienza declarando cuatro variables de tipo cadena (`s_reverse_mode`, `s_slave_mode`, `s_time_based_profile` y `s_torque_auto_on`) que representan el estado de cada modo (activado o desactivado) en formato de cadena de caracteres.

Luego, se calcula un valor de datos (`data`) combinando los diferentes modos de operación según los parámetros proporcionados. Cada modo activado establece el bit correspondiente en el valor de datos utilizando operaciones de bits.

A continuación, se utiliza la función `write1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para escribir el valor de datos calculado en el registro de modo de conducción del motor (`DRIVE_MODE`). Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR`. Si la comunicación es exitosa (`COMM_SUCCESS`), se registra un mensaje informativo utilizando `ROS_INFO` indicando que la configuración del modo de conducción ha sido cambiada, seguido de la visualización de la configuración actual del modo de conducción mediante el método `showDriveModeConfig`.

En resumen, este método permite configurar el modo de funcionamiento del motor Dynamixel activando o desactivando diferentes modos de operación, gestionando adecuadamente los posibles errores de comunicación y mostrando la configuración actualizada del modo de conducción después del cambio.

- **void dynamixelMotor::showDriveModeConfig():** Este método es un visualizador de la clase `dynamixelMotor` que muestra la configuración actual del modo de funcionamiento del motor Dynamixel.

El método comienza declarando una variable de puntero a `uint8_t` (`config`) que contendrá la configuración del modo de conducción y una variable de tipo `uint8_t` (`dxl_error`) para almacenar errores de comunicación.

A continuación, se utiliza la función `read1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer la configuración actual del modo de conducción desde el registro correspondiente del motor (`DRIVE_MODE`). Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR`.

Si la comunicación es exitosa (`COMM_SUCCESS`), se extraen los diferentes modos de operación del valor de configuración utilizando operaciones de bits. Luego, se convierten estos valores booleanos en cadenas de caracteres (`s_reverse_mode`, `s_slave_mode`, `s_time_based_profile` y `s_torque_auto_on`), indicando si cada modo está activado o desactivado.

Finalmente, se registra un mensaje informativo utilizando `ROS_INFO` mostrando la configuración actual del modo de conducción, incluyendo el estado de cada modo (activado o desactivado).

En resumen, este método proporciona una forma de visualizar la configuración actual del modo de funcionamiento del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y mostrando la configuración en un formato legible para el usuario.

- **void dynamixelMotor::configStartup(bool TORQUE_ON, bool RAM_RESTORE):** Este método es un configurador de la clase `dynamixelMotor` que permite establecer la configuración de arranque del motor Dynamixel.

El método toma dos parámetros booleanos: `TORQUE_ON` indica si se activará el torque al arrancar, y `RAM_RESTORE` indica si se restaurarán los valores de la RAM al arrancar.

El método comienza declarando una variable de tipo `uint8_t` (`data`) que almacenará la configuración de arranque y otra variable de tipo `uint8_t` (`dxl_error`) para almacenar errores de comunicación.

Luego, se utiliza la operación de bits OR para establecer los bits correspondientes en la variable `data` basándose en los valores de los parámetros `TORQUE_ON` y `RAM_RESTORE`.

Posteriormente, se utiliza la función `write1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para escribir la configuración de arranque en el registro correspondiente del motor (`STARTUP_CONFIGURATION`). Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR`.

Si la comunicación es exitosa (`COMM_SUCCESS`), se registra un mensaje informativo utilizando `ROS_INFO` indicando que la configuración de arranque ha sido cambiada correctamente. Además, se llama al método `showStartupConfig` para mostrar la configuración de arranque actual.

En resumen, este método proporciona una forma de configurar la acción del motor Dynamixel al arrancar, gestionando adecuadamente los posibles errores de comunicación y mostrando mensajes informativos sobre el estado del proceso de configuración.

- **void dynamixelMotor::showStartupConfig():** Este método es un visualizador de la clase `dynamixelMotor` que muestra la configuración de arranque actual del motor Dynamixel.

El método comienza declarando una variable de tipo `uint8_t` (`config`) que almacenará la configuración de arranque y otra variable de tipo `uint8_t` (`dxl_error`) para almacenar errores de comunicación.

Luego, se utiliza la función `read1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer la configuración de arranque del registro correspondiente del motor (`STARTUP_CONFIGURATION`). Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR`.

Si la comunicación es exitosa (COMM_SUCCESS), se utilizan operaciones de bits para extraer los valores de los bits correspondientes en la variable `config`, que representan la configuración de arranque. Luego, se asignan estos valores a las variables booleanas `TORQUE_ON` y `RAM_RESTORE`. Se utiliza `ROS_INFO` para mostrar un mensaje informativo con la configuración de arranque actual, utilizando las variables booleanas convertidas a cadenas de texto para representar los estados "ON" o "OFF".

En resumen, este método proporciona una forma de visualizar la configuración de arranque actual del motor Dynamixel, gestionando adecuadamente los posibles errores de comunicación y mostrando mensajes informativos sobre la configuración actual.

- **void dynamixelMotor::configShutdown(bool INPUT_VOLTAGE_ERROR, bool OVERHEATING_ERROR, bool ENCODER_ERROR, bool ELECTRICAL_SHOCK_ERROR, bool OVERLOAD_ERROR):** Este método es un configurador de la clase `dynamixelMotor` que permite configurar el comportamiento de apagado del motor Dynamixel en caso de diferentes tipos de errores.

El método toma cinco parámetros booleanos (`INPUT_VOLTAGE_ERROR`, `OVERHEATING_ERROR`, `ENCODER_ERROR`, `ELECTRICAL_SHOCK_ERROR` y `OVERLOAD_ERROR`) que representan los tipos de errores que pueden provocar el apagado del motor.

Se declara una variable de tipo `uint8_t` (`data`) que almacenará la configuración de apagado y otra variable de tipo `uint8_t` (`dxl_error`) para almacenar errores de comunicación.

Se utilizan operaciones de bits para configurar los bits correspondientes en la variable `data` según los valores de los parámetros booleanos proporcionados.

Luego, se utiliza la función `write1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para escribir la configuración de apagado en el registro correspondiente del motor (`SHUTDOWN`). Si la comunicación falla, se registra un mensaje de error utilizando `ROS_ERROR`.

Si la comunicación es exitosa (COMM_SUCCESS), se registra un mensaje informativo utilizando `ROS_INFO`, indicando que la configuración de apagado ha sido cambiada correctamente. Además, se llama al método `showShutdownConfig` para mostrar la configuración de apagado actual.

En resumen, este método proporciona una forma de configurar el comportamiento de apagado del motor Dynamixel en respuesta a diferentes tipos de errores, gestionando adecuadamente los posibles errores de comunicación y mostrando mensajes informativos sobre el cambio de configuración.

- **void dynamixelMotor::showShutdownConfig():** Este método es un visualizador de la clase `dynamixelMotor` que muestra la configuración de apagado del motor Dynamixel.

El método utiliza la función `read1ByteTxRx` del manejador de paquetes (`myPacketHandler`) para leer la configuración de apagado actual del motor desde el registro correspondiente

(SHUTDOWN). Se declara una variable de tipo `uint8_t` (`dxl_error`) para almacenar errores de comunicación.

Si la comunicación con el motor falla, se registra un mensaje de error utilizando `ROS_ERROR`, indicando el código de error proporcionado por el motor.

Si la comunicación es exitosa (`COMM_SUCCESS`), se utilizan operaciones de bits para interpretar los bits de la configuración leída y determinar el estado de los diferentes tipos de errores que pueden provocar el apagado del motor.

Luego, se construyen cadenas de texto (`s_input_voltage_error`, `s_overheating_error`, `s_encoder_error`, `s_electrical_shock_error` y `s_overload_error`) indicando si cada tipo de error está activado o desactivado.

Finalmente, se registra un mensaje informativo utilizando `ROS_INFO`, mostrando la configuración de apagado actual del motor, incluyendo el estado de cada tipo de error.

En resumen, este método proporciona una forma de visualizar la configuración de apagado del motor Dynamixel, interpretando los bits de la configuración leída y mostrando mensajes informativos sobre el estado de cada tipo de error.

Experimentos y Resultados

Contenido

4.1.	Entorno experimental	66
4.1.1.	Setup de Hardware	66
4.1.2.	Configuración del PC	69
4.2.	Protocolo de experimentación	70
4.2.1.	Objetivo de los Experimentos	70
4.2.2.	Preparación del Entorno	71
4.2.3.	Procedimiento Experimental	71
4.2.4.	Recopilación y Análisis de Datos	72
4.2.5.	Criterios de Éxito	72
4.3.	Experimento 1: actuadores aislados	73
4.3.1.	Contexto	73
4.3.2.	Ejecución	73
4.3.3.	Resultado	75
4.4.	Experimento 2: sistema completo	78
4.4.1.	Contexto	78
4.4.2.	Ejecución	79
4.4.3.	Resultado	81
4.5.	Discusión de los resultados	82

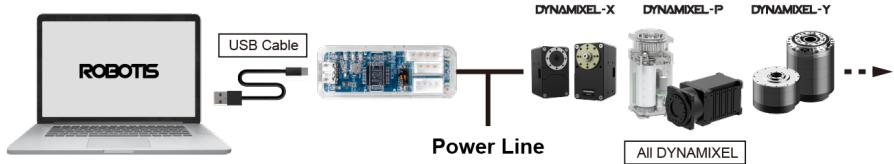


Figura 4.1: Conexiónado Dynamixel - PC (*Fuente: robotis.com*)

Este capítulo presenta los experimentos realizados para verificar el correcto funcionamiento de la librería desarrollada, `dynamixel_ros_library`. Se detallan las condiciones de los experimentos, así como los resultados obtenidos de los mismos.

Para la evaluación, se han diseñado y ejecutado dos experimentos. Durante el primer experimento, los motores Dynamixel son probados de manera individual, conectados directamente a un PC. Este enfoque permite observar y analizar el comportamiento de la librería en un entorno controlado y simplificado, facilitando la identificación y resolución de posibles errores en la implementación básica de las funcionalidades. En el segundo experimento, se evalúa la librería en un sistema robótico real compuesto por múltiples motores Dynamixel. Este experimento ofrece una visión más completa del rendimiento y la funcionalidad de la librería en un escenario práctico y de mayor complejidad, permitiendo así validar su eficacia en condiciones similares a las de su uso final.

A lo largo de este capítulo, se describirán de manera detallada los procedimientos experimentales, incluyendo la configuración del hardware y software, los parámetros de prueba y los criterios de evaluación utilizados.

Al final del capítulo, se discuten los resultados de los experimentos realizados, comentando cómo la solución final cumple con los requisitos establecidos previamente y definiendo las limitaciones actuales. Esta discusión permitirá evaluar la efectividad de la librería y considerar posibles mejoras futuras.

4.1. Entorno experimental

Es imprescindible comprender el conexionado físico que existe entre los actuadores y el dispositivo que se utilizará para supervisar el funcionamiento y mandar órdenes (en este caso un PC). Para una mayor comprensión por parte del lector de cada una de las piezas que conforman el conexionado, el setup utilizado durante los experimentos se describirá a continuación.

4.1.1. Setup de Hardware

El esquema general de conexión proporcionado por Robotis para el caso de conexión con PC es mostrado en la Figura 4.1.

A continuación se proporciona una visión mas completa de cada uno de los elementos que aparecen en el esquema de conexiónado.

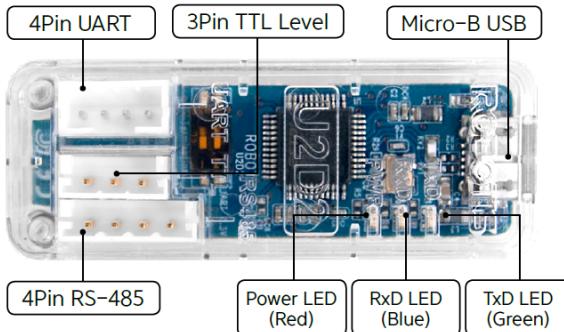


Figura 4.2: Controlador U2D2 (*Fuente: robotis.com*)

- **PC:** Es el dispositivo que juega un papel crucial como el controlador central, ya que es el responsable de ejecutar los programas y scripts que controlan el comportamiento de los motores, ya sea en términos de movimiento, velocidad, par, o cualquier otra variable operativa. Las especificaciones a nivel de software se especifican en la siguiente sección.
- **Controlador:** Se ha utilizado un controlador U2D2 de Robotis, mostrado en la Figura 4.2, para efectuar la comunicación entre el PC y los motores Dynamixel.

Una de las características destacadas del U2D2 es su versatilidad. No solo es compatible con una amplia gama de motores Dynamixel, sino que también soporta diferentes tipos de buses de comunicación, como el bus TTL y el bus RS485. Esta compatibilidad permite a los usuarios integrar fácilmente el U2D2 en diferentes sistemas y configuraciones de robots sin necesidad de hardware adicional.

El U2D2 también incluye indicadores LED que proporcionan información visual sobre el estado de la comunicación y la alimentación del dispositivo. Esto facilita la identificación y solución de problemas durante la configuración y operación del sistema. Además, el controlador cuenta con protecciones incorporadas contra sobrecorriente y cortocircuitos, lo que ayuda a prevenir daños a los motores y al propio U2D2, aumentando así la fiabilidad del sistema.

- **Alimentación:** El controlador U2D2 descrito anteriormente no se encarga de alimentar a los actuadores, lo cual hace necesario disponer de un suministro de voltaje externo. Para este fin, se ha utilizado el dispositivo U2D2 Power Hub, mostrado en la Figura 4.3, desarrollado por Robotis y que sirve específicamente para suministrar la alimentación al sistema.

El U2D2 Power Hub está diseñado para facilitar su integración con el controlador U2D2 y los motores Dynamixel. Por ello, dispone de una conexión de alimentación principal (un conector de entrada donde se conecta la fuente de alimentación externa) y múltiples puertos de salida para motores Dynamixel. Todas las conexiones del U2D2 Power Hub están mostradas en la Figura 4.4.



Figura 4.3: U2D2 Power Hub (*Fuente: robotis.com*)

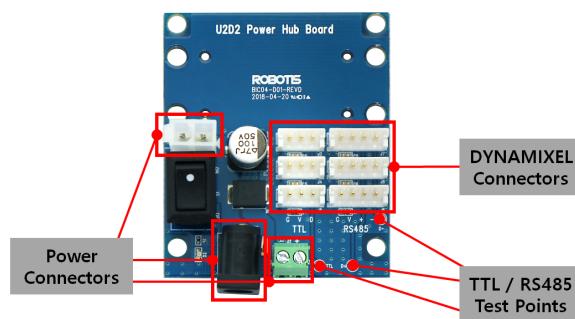


Figura 4.4: Conexiones disponibles en U2D2 Power Hub (*Fuente: robotis.com*)

Una vez conectado el controlador U2D2 (se conecta a los mismos puertos que los motores) y montado en la PCI de U2D2 Power Hub, se conectan los actuadores Dynamixel tal y como se muestra en la Figura 4.5.

- **Motores Dynamixel:** El dispositivo que cierra la cadena de conexionado es el actuador Dynamixel. Cabe destacar que, debido a la versatilidad del hardware empleado, el esquema sería exactamente el mismo en el caso de trabajar con “Series Y” o “Series P”, por ejemplo. Sin embargo, debido a que el paquete de ROS desarrollado está específicamente diseñado para los motores de “Series X”, estos serán los que se van a utilizar a lo largo de la fase de experimentación.
- **Conexiones y Adaptadores:** Para el correcto interconexión entre las partes descritas anteriormente, se emplearon cables USB y adaptadores RS485 para conectar los motores al PC.

Para el conexionado entre las partes suministradas por Robotis (motores, U2D2 y su Power Hub) se han utilizado los cables de 4 (en el caso de RS-485) o 3 (en el caso de TTL) pines desarrollados específicamente para Dynamixel. Los cables mencionados se muestran detalladamente en la Figura 4.6, mientras los pines de cada tipología de los cables se muestran en la Figura 4.7.

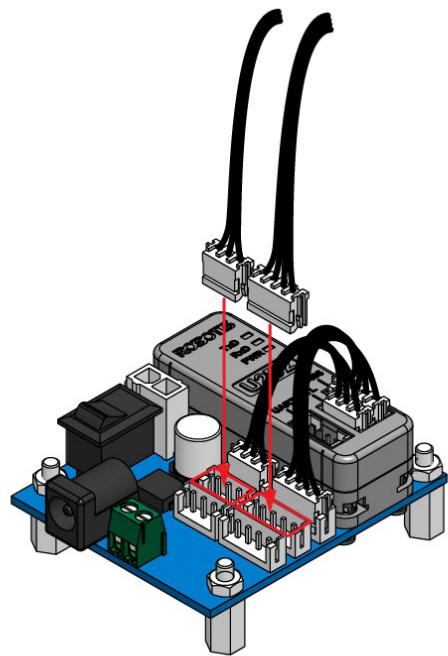


Figura 4.5: Conexión de motores al U2D2 Power Hub (*Fuente: robotis.com*)



Figura 4.6: Tipos de cables Dynamixel (*Fuente: robotis.com*)

4.1.2. Configuración del PC

Durante la realización de ambos experimentos, se ha utilizado un PC con Ubuntu 20.04¹ (también conocida como Focal Fossa) ya que es una distribución de Linux ampliamente utilizada en entornos de desarrollo y producción debido a su estabilidad, seguridad y soporte a largo plazo. Ubuntu 20.04 ofrece un entorno robusto y confiable, ideal para aplicaciones que requieren un alto grado de precisión y control, como es el caso de los sistemas robóticos.

¹Información sobre Ubuntu 20.04

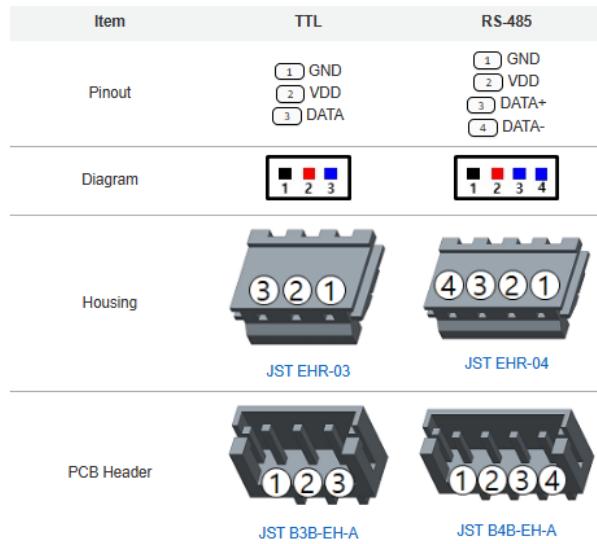


Figura 4.7: Pines de los cables Dynamixel (*Fuente: robotis.com*)

La versión del framework de desarrollo robótico empleado (ROS) es ROS Noetic Ninjemys, que es la versión de ROS compatible con Ubuntu 20.04.

4.2. Protocolo de experimentación

Esta sección describe el protocolo seguido para realizar los experimentos destinados a verificar el correcto funcionamiento de la librería `dynamixel_ros_library`. A continuación, se detallan los pasos y consideraciones tomadas durante el proceso experimental.

4.2.1. Objetivo de los Experimentos

El objetivo principal de los experimentos realizados es verificar el correcto funcionamiento de la librería `dynamixel_ros_library` y los métodos que le pertenecen. Para alcanzar este objetivo, se han llevado a cabo dos tipos de experimentos: uno con actuadores Dynamixel aislados, conectados directamente al PC, y otro con un sistema robótico real compuesto por varios motores Dynamixel.

En concreto, se pretende validar los métodos que tienen un uso frecuente en aplicaciones robóticas, tales como los métodos para configurar la comunicación y aquellos que vinculan una tabla de control con un motor Dynamixel. Es fundamental asegurar que estos métodos operen de manera correcta y eficiente, ya que son esenciales para el funcionamiento general del sistema.

Los aspectos específicos a validar incluyen: - El correcto funcionamiento físico de los actuadores Dynamixel, comprobando que responden adecuadamente a los comandos enviados a través de la librería. - La correcta visualización del feedback proporcionado al usuario, garantizando que los mensajes de confirmación de acciones realizadas y los mensajes de error en caso de fallos sean claros y precisos.

4.2.2. Preparación del Entorno

Antes de llevar a cabo los experimentos, se realizaron varias tareas de preparación para asegurar un entorno adecuado y la correcta ejecución de los procedimientos. Los experimentos se llevaron a cabo en el laboratorio asignado para el proyecto, equipado con todas las herramientas necesarias para la manipulación y prueba de los actuadores Dynamixel.

Se comenzó por identificar cada motor individualmente, asignándole un ID único y restaurándolo a los ajustes predeterminados para eliminar configuraciones previas que pudieran interferir con los experimentos. Esta fase incluyó la verificación de la existencia de ID secundarios y otros parámetros relevantes para el control de los motores.

En el ámbito del software, se aseguró que todas las dependencias estuvieran actualizadas y que se hubiera generado correctamente los archivos de compilación utilizando `catkin_make`. Se verificó que todos los nodos necesarios estuvieran lanzados y operativos antes de proceder con los experimentos principales.

Desde el punto de vista del hardware, se realizó una minuciosa inspección para verificar el correcto conexionado de todos los componentes y la integridad física de los mismos. Especial atención se prestó al funcionamiento de la fuente de alimentación, dado que su correcto funcionamiento es crucial para evitar posibles daños en los actuadores Dynamixel y en el U2D2.

4.2.3. Procedimiento Experimental

Tras verificar que tanto la parte software como hardware están actualizadas y correctamente instaladas, se procede con la ejecución de los experimentos. El procedimiento sigue una secuencia estandarizada independientemente del número de motores y la complejidad del sistema robótico. A continuación se detallan los pasos seguidos:

1. **Establecimiento del objetivo:** Se define el comportamiento deseado que el sistema robótico debe lograr durante el experimento.
2. **Traducción del comportamiento en acciones:** Se realiza una descomposición del comportamiento deseado en acciones específicas y preparatorias. Esta etapa es puramente teórica y tiene como objetivo segmentar el proceso en acciones manejables.
3. **Vinculación con métodos de la librería:** Se identifican y vinculan las acciones requeridas con los métodos disponibles en la librería `dynamixel_ros_library`. Cada comportamiento requiere una inicialización adecuada que incluye la definición de instancias para representar los motores, la configuración de `packetHandler` y `portHandler`, la inicialización de la comunicación y la asignación de una tabla de control específica para el modelo utilizado. Además, se implementan acciones específicas según el comportamiento definido en el paso anterior.

4. **Desarrollo del programa específico:** Se procede al desarrollo del programa específico utilizando las funciones proporcionadas por la librería. Se destaca que los ejemplos incluidos en la librería son altamente reutilizables y facilitan el proceso de programación.
5. **Ejecución y verificación:** Una vez desarrollado y compilado el programa, se inicia el nodo principal *rosmaster* en la terminal de Ubuntu. Se ejecuta el programa específico y se observa la respuesta del motor (esencial verificar que la comunicación y la alimentación funcionan correctamente antes de mandarle ordenes al motor).

Este procedimiento asegura que cada experimento se lleva a cabo de manera sistemática y que se cumplen los objetivos establecidos para verificar el correcto funcionamiento de la librería y los métodos implementados.

4.2.4. Recopilación y Análisis de Datos

Para capturar y analizar los resultados de los experimentos, se utiliza la herramienta de almacenamiento y visualización de datos *PlotJuggler*, cuya interfaz gráfica es mostrada en la Figura 4.8. Esta herramienta, aunque no está incluida por defecto en el paquete de ROS, puede ser instalada para facilitar el análisis de los datos obtenidos durante los experimentos.

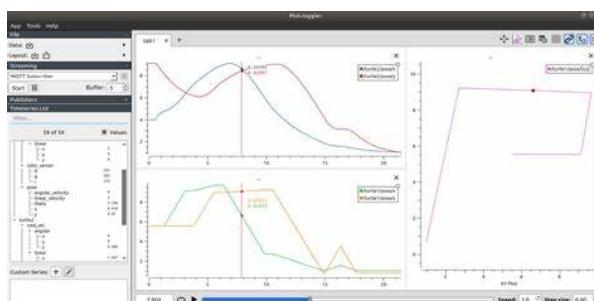


Figura 4.8: Interfaz gráfica de PlotJuggler (Fuente: blog.csdn.net)

4.2.5. Criterios de Éxito

Durante la realización de los experimentos que se detallarán a continuación, se ha considerada exitosa una comprobación si, al ejecutar un método, este se ha comportado de manera esperada. Esto incluye:

- **Acción asignada realizada con éxito:** El método ha cumplido con la acción asignada. En el caso de los métodos setters es el cambio del valor correspondiente en la memoria interna del motor, mientras en el caso de los métodos getters es el correcto acceso y la visualización del dato correspondiente.

- **Feedback proporcionado:** El método debe proporcionar un feedback claro y conciso sobre lo que se ha realizado en caso de éxito. En caso de fallo, se debe visualizar la causa del mismo y/o el código del error.

4.3. Experimento 1: actuadores aislados

4.3.1. Contexto

Este experimento representa la prueba inicial realizada tras el desarrollo completo de la librería. Su propósito fue verificar el correcto funcionamiento de varios aspectos clave de la librería *dynamixel_ros_library*. Los aspectos evaluados incluyen:

1. El manejo adecuado de las instancias `dynamixelMotor` y el correcto funcionamiento del método constructor.
2. La correcta inicialización de la comunicación mediante el método correspondiente.
3. La adecuada configuración del modo de operación en los motores Dynamixel, evaluando específicamente el modo de control de posición y verificando el funcionamiento de las constantes de clase asociadas a estos modos.
4. La correcta escritura de un valor de posición objetivo (Goal position) utilizando los métodos proporcionados por la librería.
5. La adecuada realimentación (feedback) de los métodos ejecutados.

Para llevar a cabo estos objetivos, se empleó el ejemplo `testPositionControl.cpp` disponible en el paquete, utilizando un motor Dynamixel del modelo XM430-W350-R (mostrado en la Figura 4.9). Este experimento se realizó en condiciones controladas para asegurar la precisión de los resultados y la comparabilidad de los mismos.

4.3.2. Ejecución

La ejecución del experimento comenzó con la definición del comportamiento deseado, el cual consistía en controlar un motor para que se moviera a una posición angular entre 0 y 360 grados, en respuesta a mensajes enviados a un topic específico. Para este propósito, se utilizó el ejemplo `testPositionControl.cpp`, ya que su funcionamiento coincidía con las especificaciones del experimento.

El manejo de los nodos necesarios (como `rosmaster` y el nodo encargado de recibir y procesar los mensajes para mover el motor) y el envío de mensajes se llevó a cabo mediante tres terminales de Ubuntu. Después de iniciar `roscore` y ejecutar el nodo asociado a `testPositionControl.cpp`, se observó el comportamiento del sistema como se muestra en la Figura 4.10. En dicha figura, se puede

CAPÍTULO 4. EXPERIMENTOS Y RESULTADOS

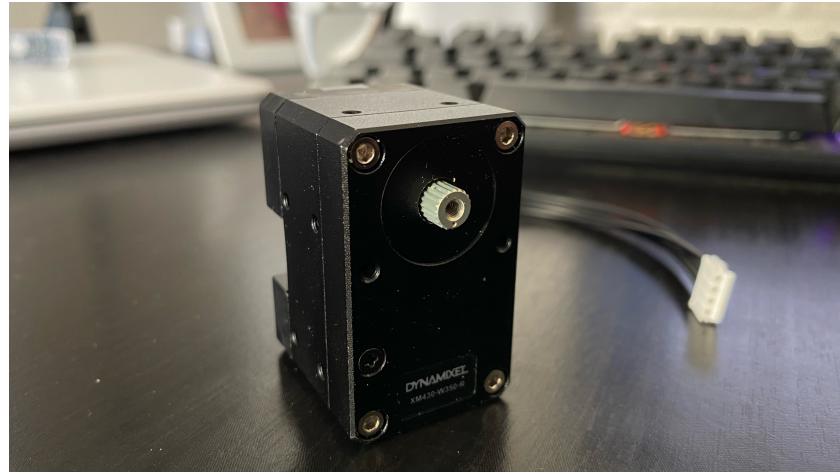


Figura 4.9: Motor utilizado durante el experimento

```

Activities X-terminal-emulator •
maxxx@MaxPC:~$ roscore
maxxx@MaxPC:~$ roscore http://MaxPC:19311/foxy
[ INFO] [1718894860.53207519]: Logging to /home/maxxx/.ros/log/cc7edfb8-2f13-11ef-82ae-331b1a575702/roslaunch-MaxPC-3210.log
[ INFO] [1718894860.53207519]: Checking log file disk usage. This may take a while.
[ INFO] [1718894860.53207519]: Press Ctrl-C to interrupt
[ INFO] [1718894860.53207519]: Done checking log file disk usage. Usage is <1GB.
[ INFO] [1718894860.53207519]: started roslaunch server http://MaxPC:136915/
[ INFO] [1718894860.53207519]: ros_com version 1.16.0
[ INFO] [1718894860.53207519]: SUMMARY
[ INFO] [1718894860.53207519]: processes
[ INFO] [1718894860.53207519]: PARAMETERS
[ INFO] [1718894860.53207519]:   * /rosslstart: noetic
[ INFO] [1718894860.53207519]:   * /rossversion: 1.16.0
[ INFO] [1718894860.53207519]: NODES
[ INFO] [1718894860.53207519]:   * auto-starting new master
[ INFO] [1718894860.53207519]:     process[master]: started with pid [3220]
[ INFO] [1718894860.53207519]:       ROS_MASTER_URI=http://MaxPC:19311/
[ INFO] [1718894860.53207519]:     setting /run_id to c7cd3f8-2f13-11ef-82ae-331b1a575702
[ INFO] [1718894860.53207519]:     process[rosout-1]: started with pid [3230]
[ INFO] [1718894860.53207519]:       started core service [/rosout]
maxxx@MaxPC:~$ rosrun dynamixel_ros_library testPositionControl /dev/ttyUSB0 2.0 57000 2
[ INFO] [1718894860.53207519]: Initialization success
[ INFO] [1718894860.540797938]: DHAL 2: Control table set for: XM430-W350
maxxx@MaxPC:~$ 
maxxx@MaxPC:~$ 101x27
maxxx@MaxPC:~$ 
```

Figura 4.10: Lanzamiento de los nodos necesarios en la terminal

apreciar que al iniciar el nodo, la librería `dynamixel_ros_library` inicializó la comunicación con el motor, reportó un mensaje de confirmación y estableció la tabla de control necesaria vinculada al modelo especificado, quedando en espera de mensajes entrantes.

Cuando se envía un mensaje con una posición objetivo al topic correspondiente, el nodo `testPositionControl` verifica instantáneamente si el motor está configurado en modo de control de posición y si el torque está activado. Solo entonces procede a escribir la posición objetivo. Es importante destacar que el método utilizado filtra automáticamente los valores fuera del rango de 0 a 360 grados, eliminando así la necesidad de que el usuario maneje esta consideración.

Comenzaremos el experimento mandándole al motor una referencia angular de 0 grados (Figura 4.11).

Una vez recibida la posición objetivo, el motor responde notificando al usuario sobre el cambio y

```

Activities X-terminal-emulator •
maxxx@MaxPC:~$ roscore
... logging to /home/maxxx/.ros/log/c7ed3f8-2f13-11ef-82ae-331b1a575702/roslaunch-MaxPC-3210.log
Distro: No distro specified, defaulting to 'noetic'.
Press Ctrl-C to interrupt.
Done checking log file disk usage. Usage is <1GB.
started roslaunch server http://MaxPC:36915/
ros_comm version 1.16.0

SUMMARY
  success: 1
  errors: 0
PARAMETERS
  * /rosdistro: noetic
  * /rosversion: 1.16.0
NODES
  auto-starting new master
    process[master]: started with pid [3220]
    ROS_MASTER_URI=http://MaxPC:11311/
  setting run_id to c7ed3f8-2f13-11ef-82ae-331b1a575702
  success[rosout]: started with pid [3230]
  started core service [/rosout]
maxxx@MaxPC:~$ rostrun dynamixel ros_library testPositionControl /dev/ttyUSB0 2.0 57600 2
[ INFO] [178884860.532672519]: Initialization success
[ INFO] [178884860.532672519]: Set target position for: XH430-W350
[ INFO] [178884976.026449826]: DXL1 Z: Torque ts: OFF
[ INFO] [178884976.026449826]: DXL2 Z: Changed the operating mode.
[ INFO] [178884976.026449826]: DXL3 Z: Torque ts: OFF
[ INFO] [178884976.026449826]: DXL4 Z: Torque state ts set to: ON
[ INFO] [178884976.026449826]: DXL Z: Goal position is set to: 0 degrees
maxxx@MaxPC:~$ rostopic pub -1 /user_position/input lid_pos/int16 "data: 0"
publishing and latching message for 3.0 seconds
maxxx@MaxPC:~$ 

```

Figura 4.11: Respuesta para la orden de 0 grados

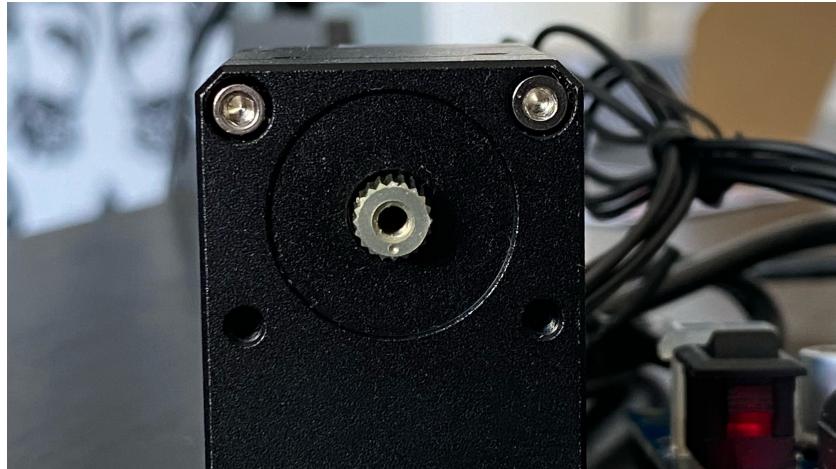


Figura 4.12: Posición del motor después de recibir una posición objetivo de 0 grados

comienza a moverse hacia la nueva posición. La posición final del motor se muestra en la Figura 4.12.

Tras eso, se comprobó el equivalente funcionamiento de la librería para el caso de 90 y 270 grados. Las Figuras 4.13 y 4.14 muestran la respuesta de la librería a la llegada de una posición - objetivo nueva.

A su vez, las posiciones del motor tras recibir y ejecutar las órdenes mostradas anteriormente se muestran en las Figuras 4.15 y 4.16, respectivamente.

4.3.3. Resultado

Los resultados obtenidos en el experimento han demostrado la funcionalidad completa de la librería desarrollada. Se ha verificado que la librería es capaz de establecer la comunicación con el controlador U2D2 internamente, utilizando los datos proporcionados por el usuario como el

CAPÍTULO 4. EXPERIMENTOS Y RESULTADOS

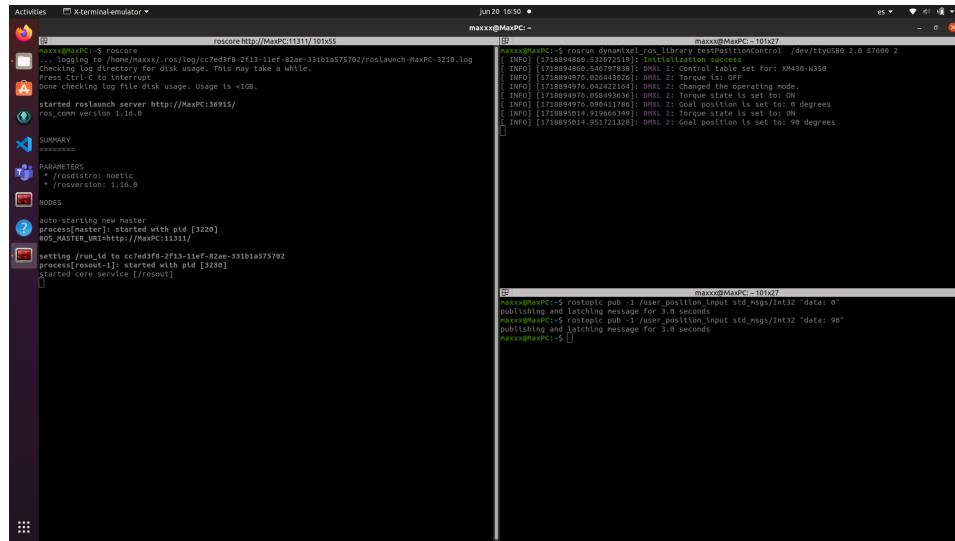


Figura 4.13: Respuesta para la orden de 90 grados

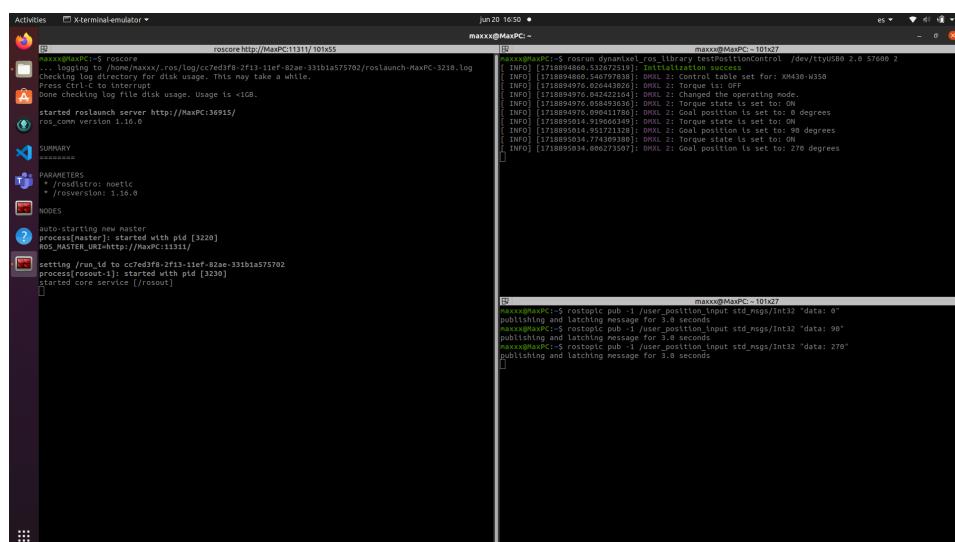


Figura 4.14: Respuesta para la orden de 270 grados



Figura 4.15: Posición del motor después de recibir una posición objetivo de 90 grados

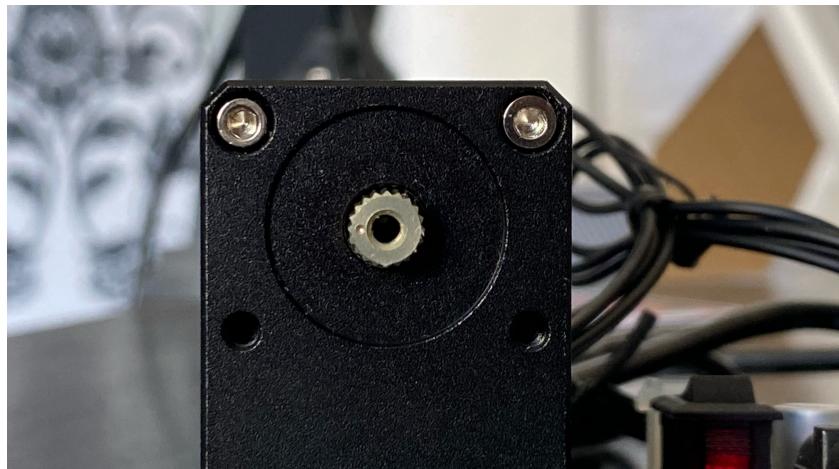


Figura 4.16: Posición del motor después de recibir una posición objetivo de 270 grados

puerto USB, la versión del protocolo y el baudrate. Además, se ha implementado un mecanismo de notificación al usuario una vez que se ha completado la configuración de la comunicación con éxito.

Posteriormente, la librería procede automáticamente a configurar la tabla de control correspondiente al modelo de motor utilizado. Durante este proceso, se ha confirmado que el usuario recibe adecuadamente un mensaje de confirmación al finalizar la configuración de la tabla de control.

En términos de funcionamiento, se realizaron pruebas moviendo el motor a tres posiciones diferentes: 0 grados, 90 grados y 270 grados. Durante estas pruebas, la librería proporcionó notificaciones continuas sobre diversos aspectos del funcionamiento del motor, como cambios en el valor del torque, ajustes en el modo de control según fuera necesario y actualizaciones en la posición objetivo establecida.



Figura 4.17: Vista frontal del sistema utilizado en el experimento



Figura 4.18: Vista lateral del sistema utilizado en el experimento

4.4. Experimento 2: sistema completo

4.4.1. Contexto

En esta ocasión, se trata de un experimento que involucra un sistema considerablemente más complejo. La finalidad de este experimento es demostrar que la cantidad de motores, o lo que es lo mismo, la complejidad física del sistema, no afecta al funcionamiento de la librería desarrollada.

Específicamente, se ha trabajado con un sistema fijo desarrollado por el Departamento de Ingeniería de Sistemas y Automática, el cual consta de tres efectores finales. Este sistema está compuesto por seis motores Dynamixel, de los cuales tres se utilizan para el movimiento giratorio de los efectores finales, siendo el eje de rotación la parte fija del robot, y los otros tres se emplean para permitir la rotación del efecto final sobre su propio eje. Una vista detallada del sistema robótico se muestra en las Figuras 4.17 (vista frontal), 4.18 (vista lateral) y 4.19 (vista desde arriba).

Los aspectos a supervisar en este experimento son exactamente los mismos que en el experimento anterior: existencia de feedback, manejo de rangos de valores, correcta escritura y lectura de datos, entre otros. Estos factores son críticos para validar la robustez y eficacia de la librería



Figura 4.19: Vista desde arriba del sistema utilizado en el experimento

`dynamixel_ros_library` en entornos más complejos y con múltiples actuadores funcionando simultáneamente.

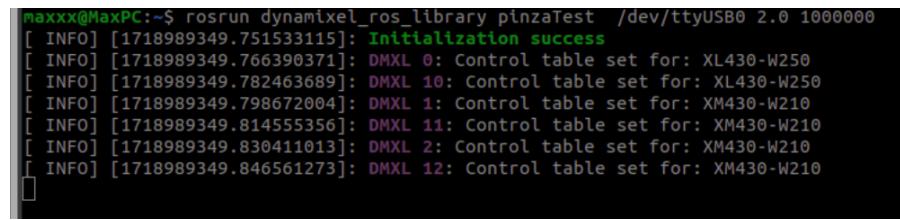
Para trabajar con este sistema en específico, se diseñó y ejecutó un programa denominado `realTest.cpp`. El código fuente de este programa está disponible en el repositorio del proyecto en GitHub². Este programa hace uso de las funcionalidades de la librería `dynamixel_ros_library` y de las herramientas proporcionadas por ROS para gestionar y controlar un sistema de seis motores Dynamixel. A continuación, se describe en detalle el procedimiento seguido durante la ejecución del experimento.

4.4.2. Ejecución

Primero, al lanzar el nodo `realTest`, se inicializa la comunicación entre el PC y los seis actuadores Dynamixel. Este paso es crucial para asegurar que todos los motores estén correctamente conectados y listos para recibir instrucciones. La librería `dynamixel_ros_library` maneja automáticamente la configuración de los parámetros de comunicación, tales como el puerto USB, la versión del protocolo y la tasa de baudios, proporcionando un mensaje de confirmación al usuario una vez que la comunicación ha sido establecida con éxito, tal y como se muestra en la Figura 4.20. Inmediatamente después, el programa aplica el método `setControlTable` sobre los seis motores, que han sido previamente instanciados en el código. Este método configura la tabla de control de cada motor según su modelo específico, asegurando que todos los registros internos estén correctamente inicializados y listos para su uso. Este proceso también genera un mensaje de confirmación para el usuario, indicando que cada motor ha sido configurado correctamente.

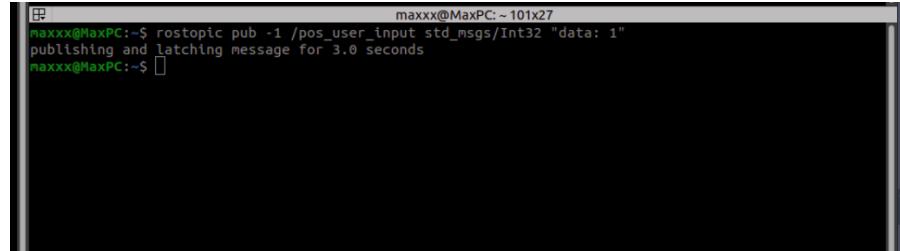
Una vez dentro del bucle de ROS, el programa se queda a la espera de recibir una información a través del tema `/pos_user_input`. La información esperada es un número entero del 0 al 2, que corresponde a una de las tres posiciones predefinidas. Cada una de estas posiciones predefinidas

²Disponible en: https://github.com/TaISLab/dynamixel_ros_library/tree/main



```
maxxx@MaxPC:~$ rosrun dynamixel_ros_library pinzaTest /dev/ttyUSB0 2.0 1000000
[ INFO] [1718989349.751533115]: Initialization success
[ INFO] [1718989349.766390371]: DMXL 0: Control table set for: XL430-W250
[ INFO] [1718989349.782463689]: DMXL 10: Control table set for: XL430-W250
[ INFO] [1718989349.798672004]: DMXL 1: Control table set for: XM430-W210
[ INFO] [1718989349.814555356]: DMXL 11: Control table set for: XM430-W210
[ INFO] [1718989349.830411013]: DMXL 2: Control table set for: XM430-W210
[ INFO] [1718989349.846561273]: DMXL 12: Control table set for: XM430-W210
```

Figura 4.20: Mensaje de éxito tras la correcta inicialización



```
maxxx@MaxPC:~ 101x27
maxxx@MaxPC:~$ rostopic pub -1 /pos_user_input std_msgs/Int32 "data: 1"
publishing and latching message for 3.0 seconds
maxxx@MaxPC:~$ 
```

Figura 4.21: La publicación del identificador de la posición predefinida “1”



```
maxxx@MaxPC:~ 101x27
[ INFO] [1718989383.060094036]: DMXL 0: Torque state is set to: ON
[ INFO] [1718989383.076060235]: DMXL 10: Torque state is set to: ON
[ INFO] [1718989383.091958480]: DMXL 1: Torque state is set to: ON
[ INFO] [1718989383.108279533]: DMXL 11: Torque state is set to: ON
[ INFO] [1718989383.124036814]: DMXL 2: Torque state is set to: ON
[ INFO] [1718989383.140010560]: DMXL 12: Torque state is set to: ON
[ INFO] [1718989383.172024318]: DMXL 0: Goal position is set to: 180 degrees
[ INFO] [1718989383.203933414]: DMXL 10: Goal position is set to: 240 degrees
[ INFO] [1718989383.235943791]: DMXL 1: Goal position is set to: 120 degrees
[ INFO] [1718989383.267945702]: DMXL 11: Goal position is set to: 280 degrees
[ INFO] [1718989383.299933251]: DMXL 2: Goal position is set to: 240 degrees
[ INFO] [1718989383.331943889]: DMXL 12: Goal position is set to: 120 degrees
```

Figura 4.22: Respuesta de la librería tras recibir la orden correspondiente a la posición predefinida “1”

implica un conjunto de seis posiciones específicas, una para cada motor del sistema. Este diseño permite una comprobación coordinada de todos los motores en diversas configuraciones de posición.

Al recibir la referencia de posición predefinida a través del tópico mencionado, tal y como se muestra en la Figura 4.21, el programa inmediatamente establece el valor del torque en true mediante el método `setTorqueState()`, habilitando así el control del motor. Posteriormente, se establece una posición objetivo para cada motor utilizando el método `setGoalPosition()` (esta respuesta se muestra en la Figura 4.22). Este método asegura que cada motor se mueva a la posición especificada, filtrando cualquier valor fuera del rango permitido, lo cual simplifica significativamente la tarea del usuario.

Para una mayor comprensión por parte del lector, el experimento fue grabado y publicado en el repositorio³ perteneciente a este Trabajo Fin de Grado.

Por otro lado, para aprovechar la compatibilidad completa del paquete `dynamixel_ros_library`

³Disponible en: https://github.com/TaISLab/dynamixel_ros_library/tree/main

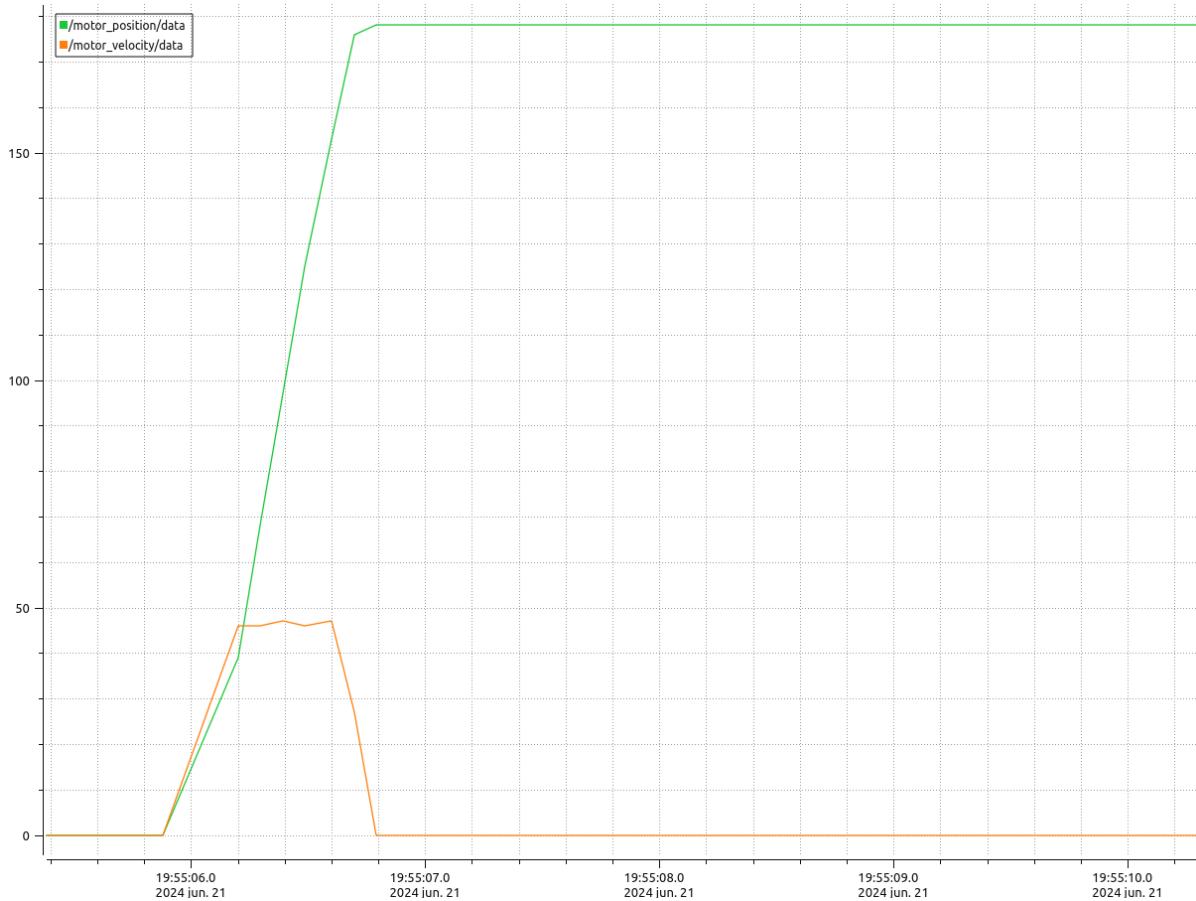


Figura 4.23: Variación de posición y velocidad del motor al recibir una referencia nueva

con ROS, se ha decidido exportar datos como posición y la velocidad actual en un gráfico que represente el cambio mostrado anteriormente. El actuador Dynamixel sobre el cual se va a poner el foco para exportar datos será el correspondiente con el ID:0. El resultado se muestra en la Figura 4.23.

NOTA: Las unidades de los datos publicados en los tópicos /motor_position y /motor_velocity se heredan directamente de los métodos aplicados para la obtención de esos datos del motor. En este caso la velocidad se publica en rpm y la posición en grados.

4.4.3. Resultado

Tras la ejecución del experimento, se han obtenido los siguientes resultados:

1. **Comunicación establecida:** La librería ha demostrado ser capaz de establecer la comunicación con los actuadores Dynamixel mediante el puerto USB especificado por el usuario. Durante este proceso, la librería ha proporcionado un mensaje de confirmación al usuario, indicando la correcta finalización de la configuración de la comunicación. Este resultado valida la robustez del método de inicialización de comunicación, esencial para la correcta operación del sistema.

2. **Identificación de modelos:** La librería ha identificado correctamente todos los modelos de los motores involucrados en el experimento. Al inicio del programa, se mostraron mensajes específicos que indicaban el modelo de cada motor. Esto asegura que la librería puede manejar diversos modelos de Dynamixel sin problemas, lo cual es crucial en entornos donde se utilizan múltiples tipos de actuadores.
3. **Manejo del estado del torque y modos de Operación:** La librería ha manejado eficazmente el estado del torque de cada motor, así como los modos de operación y los valores de consigna de posición. Durante el experimento, se verificó que la librería podía cambiar el estado del torque y el modo de operación según lo requerido, y establecer con precisión los valores de posición objetivo. Este comportamiento se corroboró mediante mensajes de retroalimentación proporcionados por la librería, que informaban al usuario de cada cambio de estado y ajuste de posición.

4.5. Discusión de los resultados

Los experimentos realizados han sido fundamentales para verificar el correcto funcionamiento de la librería `dynamixel_ros_library`. A través de pruebas exhaustivas con diferentes configuraciones y escenarios, se ha confirmado que la librería cumple con los requisitos establecidos y ofrece una solución robusta y eficiente para el manejo de actuadores Dynamixel en el entorno ROS.

En el primer experimento, que involucró el manejo de un único motor Dynamixel, se validaron aspectos críticos como la correcta instanciación de los objetos de la clase `dynamixelMotor`, el inicio de la comunicación y la configuración de la tabla de control. Los resultados mostraron que la librería es capaz de establecer una comunicación fiable con el controlador U2D2 y configurar adecuadamente los modos de operación y parámetros del motor. Además, se comprobó que los métodos responsables de escribir valores de posición objetivo y proporcionar feedback al usuario funcionan como se esperaba. Este experimento demostró la capacidad de la librería para manejar un motor individual de manera eficiente y precisa.

El segundo experimento se centró en un sistema más complejo, compuesto por seis motores Dynamixel. El objetivo era evaluar cómo la librería se comporta en un escenario con mayor complejidad física. Los resultados fueron igualmente satisfactorios, demostrando que la librería puede manejar múltiples motores simultáneamente sin perder rendimiento ni precisión. La comunicación y el control de los motores se realizaron de manera correcta, y se observó que la librería es capaz de proporcionar feedback adecuado y manejar rangos de valores de manera eficiente. Este experimento subrayó la escalabilidad de la librería y su capacidad para integrarse en sistemas robóticos complejos.

En ambos experimentos, la herramienta `plotJuggler` se utilizó para almacenar y visualizar los datos obtenidos, lo que permitió una evaluación detallada del rendimiento de la librería y del comportamiento de los motores. La visualización de los datos confirmó que la librería maneja correctamente

los cambios en los valores de torque, modos de control y posiciones objetivo, proporcionando una interfaz de usuario clara y concisa.

En conclusión, los experimentos realizados han demostrado que la librería `dynamixel_ros_library` es una herramienta fiable y eficiente para el manejo de actuadores Dynamixel en el entorno ROS. Los resultados obtenidos validan que la librería cumple con los objetivos planteados, ofreciendo una solución que simplifica y optimiza el proceso de configuración y control de estos motores. Las pruebas realizadas también han permitido identificar áreas de mejora y posibles líneas futuras de desarrollo, como la implementación en ROS2 y la expansión a otras series de Dynamixel.

Conclusiones

El objetivo principal del presente Trabajo Fin de Grado ha sido la creación de una herramienta versátil y eficiente para la configuración y manejo de los actuadores inteligentes Dynamixel. Esta herramienta debía corregir y resolver los problemas de las herramientas disponibles, que presentan una alta exigencia en cuanto a la formación necesaria para el desarrollo de software de estos motores sin posibilidad de abstracción del hardware. Para lograr que la herramienta sea fácil de integrar y de usar, se tomaron varias decisiones en cuanto al lenguaje de programación y la base de trabajo, siendo la decisión final un paquete de ROS programado en C++.

Al finalizar este trabajo, se ha desarrollado una herramienta capaz de trabajar con varios modelos de Dynamixel, permitiendo consultar y manejar todos y cada uno de los registros internos de la memoria. Además, la librería automatiza procesos como el inicio de comunicación con el controlador U2D2 y el ajuste de la tabla de control del actuador en función del modelo. También se ocupa de la conversión de unidades, el seguimiento de errores y el cumplimiento de los rangos para las variables. La implementación de las tablas de control merece un comentario particular, ya que se ha utilizado una herramienta potente de C++ que representa precisamente una tabla de control, `std::map`.

En base a los resultados obtenidos, se puede concluir que la mayor aportación de este trabajo reside en la disminución del tiempo de formación necesario para manejar un sistema compuesto por actuadores Dynamixel. A diferencia de las herramientas de desarrollo de software disponibles hasta la fecha, esta librería proporciona a un desarrollador llevar a cabo aplicaciones con estos motores con un alto nivel de abstracción del hardware. En concreto, se ha demostrado que el usuario no necesita conocer toda la casuística del motor con el que trabaja, ni tener que realizar manualmente ciertas tareas, ni tampoco interactuar con grandes tablas de control a nivel de registro indicando cada Byte que se manda/recibe de cada motor de forma individual. La mayor ventaja de esta librería respecto a las soluciones existentes es el enfoque desde la programación orientada a objetos. La

creación de una clase permite la representación de un motor concreto dentro del código C++, lo cual hace que trabajar con múltiples motores sea más intuitivo y sencillo. Además, es posible la creación de clases hijas en el futuro que se puedan aplicar para trabajar con nuevos modelos de Dynamixel con características adicionales, simplemente expandiendo y complementando la clase padre.

Como limitaciones de la versión actual de la librería, se destaca la imposibilidad de cambiar el tipo de protocolo durante la ejecución de un programa. Actualmente, el usuario debería cambiar el protocolo utilizado accediendo a la memoria del motor y luego relanzar el nodo con la versión del protocolo requerida. Por otro lado, la librería asume que todos los motores instanciados se comunican a la misma velocidad en bits por segundo, lo cual podría no ser cierto en algunas aplicaciones muy específicas.

El desarrollo futuro de la librería podría centrarse en su implementación en el marco de ROS2, que es la versión más reciente y actual de Robot Operating System. Asimismo, la librería podría expandirse a otras series de Dynamixel, como la “Series Y” o la “Series P”. Estas mejoras permitirían que la herramienta sea aún más versátil y aplicable a un mayor número de situaciones y aplicaciones dentro del campo de la robótica.

En conclusión, este trabajo ha demostrado la viabilidad y utilidad de una librería de manejo de actuadores Dynamixel que simplifica y optimiza la configuración y control de estos motores. A pesar de las limitaciones actuales, las posibles líneas futuras de trabajo prometen un avance significativo en la eficiencia y accesibilidad de los sistemas robóticos que utilizan estos actuadores.

Bibliografía

- [1] E. Garcia, M. A. Jimenez, P. G. De Santos, and M. Armada, “The evolution of robotics research,” *IEEE Robotics & Automation Magazine*, vol. 14, no. 1, pp. 90–103, 2007.
- [2] “Global robotics race: Korea, singapore and germany in the lead,” <https://ifr.org/ifr-press-releases/news/global-robotics-race-korea-singapore-and-germany-in-the-lead>, [Accedido: 2024-06-21].
- [3] N. Kashapov, I. Khafizov, I. Nurullin, and Z. Sadykov, “Influence of introduction of robotics on increase in efficiency of electrochemical production,” in *IOP Conference Series: Materials Science and Engineering*, vol. 412, no. 1. IOP Publishing, 2018, p. 012034.
- [4] A. J. Cordova Bermeo *et al.*, “Análisis y aplicación de los controladores programables en la industria,” B.S. thesis, Escuela Superior Politécnica del Litoral, 1992.
- [5] M. J. E. Tornero and A. J. R. Fernández, “Actuadores neumáticos,” *Ingeniería Industrial, Universidad de Huelva*, 2016.
- [6] K. Suzumori, “New robotics pioneered by fluid power,” *Journal of Robotics and Mechatronics*, vol. 32, no. 5, pp. 854–862, 2020.
- [7] C. J. Verucchi, R. Ruschetti, and G. Kazlauskas, “High efficiency electric motors: economic and energy advantages,” *IEEE Latin America Transactions*, vol. 11, no. 6, pp. 1325–1331, 2013.
- [8] L. Thompson, “Brushless vs. brushed dc motors: Choosing the right motor for your application,” *Electronics World*, vol. 106, no. 1764, pp. 44–48, 2000.
- [9] A. Hughes and B. Drury, *Electric Motors and Drives: Fundamentals, Types and Applications*. Newnes, 2013.
- [10] P. Korondi and H. Hashimoto, “Intelligent space for robotic applications,” *IEEE/ASME Transactions on Mechatronics*, vol. 10, no. 3, pp. 342–348, 2005.
- [11] H. Kang, M. Ma, K. Obstein, and B. Morgan, “Robotics in medical applications,” *Proceedings of the IEEE*, vol. 96, no. 2, pp. 194–207, 2007.

BIBLIOGRAFÍA

- [12] G. Tuna, "Control of mobile robots using brushless dc motor," *Procedia Computer Science*, vol. 1, no. 1, pp. 2763–2771, 2009.
- [13] D. Shah and P. Abbeel, "Propeller selection for high efficiency electric uavs," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4515–4520, 2012.
- [14] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*. Springer, 2016.
- [15] G. Phillips and B. Surgenor, "Stepper motor control for cnc and 3d printing applications," *IEEE Transactions on Industrial Electronics*, vol. 63, no. 3, pp. 1780–1787, 2016.
- [16] M. Kim, J. Park, and S.-R. Lee, "High-precision control of robot arms using low-cost servo motors," *IEEE/ASME Transactions on Mechatronics*, vol. 23, no. 2, pp. 782–793, 2018.
- [17] A. Kaplan, A. Okamura, and M. Sakamoto, "Medical robotics for challenging surgical applications," *Annual Review of Biomedical Engineering*, vol. 8, pp. 453–475, 2006.
- [18] G. A. Bekey, *Autonomous robots: From biological inspiration to implementation and control*. MIT Press, 2005.
- [19] J. Kim and S. Lee, "Robotis dynamixel for robotic applications," *Journal of Intelligent & Robotic Systems*, vol. 55, no. 1-2, pp. 117–123, 2009.
- [20] J. Kim and S. Kim, "Robotis dynamixel actuators for precise robotic manipulation," in *Proceedings of the 2011 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2011, pp. 1988–1993.
- [21] F. Muhamad and J.-S. Kim, "Dynabot: Modular quadruped platform with dynamixel," *Control & Intelligent Robotics Lab, SeoulTech*, vol. (Acceso el 20/06/2024), 2023. [Online]. Available: <https://sites.google.com/site/cdslweb/publication/dynabot-robot>
- [22] J. Ahn, J. Ahn, and S. Oh, "The rise of the machines: Robotis, a frontier in educational and industrial robots in korea," in *International Conference on Electronic Business (ICEB)*, 2015.
- [23] A. Koubaa *et al.*, *Robot Operating System (ROS)*. Springer, 2017, vol. 1.
- [24] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: a practical introduction to the Robot Operating System*. .O'Reilly Media, Inc.", 2015.
- [25] S. Cousins, "Welcome to ros topics [ros topics]," *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, pp. 13–14, 2010.
- [26] P. Wegner, "Concepts and paradigms of object-oriented programming," *ACM Sigplan Oopps Messenger*, vol. 1, no. 1, pp. 7–87, 1990.
- [27] T. Rentsch, "Object oriented programming," *ACM Sigplan Notices*, vol. 17, no. 9, pp. 51–57, 1982.

- [28] M. A. Linton, “Encapsulating a c++ library.” in *C++ Conference*, 1992, pp. 57–66.
- [29] M. Lattanzi and S. Henry, “Software reuse using c++ classes: The question of inheritance,” *Journal of Systems and Software*, vol. 41, no. 2, pp. 127–132, 1998.

