Trabajo de Fin de Grado

Grado en Ingeniería Informática

Vehículo de mapeo automatizado

Automated Mapping Vehicle

Antonio Sanjuán Prieto

La Laguna, 4 de septiembre de 2018

D. **Jonay T. Toledo Carrillo**, con N.I.F. 78698554-Y profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

“Vehículo de mapeo automatizado”

ha sido realizada bajo su dirección por D. **Antonio Sanjuán Prieto**,

con N.I.F. 54080570-A.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 4 de septiembre de 2018

Agradecimientos

En primer lugar debo mi más sincero agradecimiento a mi familia, por estar ahí en todo momento ayudando y escuchando, sin ellos no habría sido posible completar este proyecto.

A todas las personas y amigos que me han acompañado a lo largo de mi vida académica. Por encima de todos, a aquellos que empezaron siendo simples compañeros y terminaron convirtiéndose en amigos.

Finalmente pero por ello no menos importante, a mi tutor y fuente de inspiración Jonay Toledo, gracias por tu enorme paciencia y la ayuda prestada durante todo el desarrollo.

Licencia

Resumen

El objetivo de este proyecto es el de construir una silla de ruedas autónoma de bajo coste, para prestar servicio a personas con movilidad reducida, resolviendo de esta manera un problema a usuarios con graves limitaciones con respecto al acceso y al desplazamiento en determinadas áreas.

El diseño de la electrónica y la infraestructura base del proyecto se han llevado a cabo mediante un estudio a bajo y alto nivel, implementando el diseño mecánico, electrónico y la programación del Firmware necesario..

En este proyecto ROS (Robot Operating System) se empleo como base para realizar el software de alto nivel y la comunicación entre los diferentes módulos que conforman el sistema, permitiéndonos así mapear el área circundante de la silla.

Para el desarrollo del proyecto se empleo WSL (Windows Subsystem for Linux)

Palabras clave: *Silla de ruedas, ROS, WSL, Mapping*

Abstract

This project aims to build an lowcost autonomous wheelchair whose purpose is to provide a service to people with reduced mobility, resolving in this way problems to users with serious limitations with respect to access and displacement in certain areas.

The design of the electronics and the basic infrastructure of the project has been carried out with a study at both low and high level. The mechanical design, electronics and Firmware is developed in this project.

In this project ROS (Robot Operating System) is used as software base and communictaion of the different modules which composes the system.

WSL (Windows Subsystem for Linux) was used to develop the project.

Keywords: *Wheelchair, ROS, WSL, Mapping*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1
Introducción

1.1 Motivación

Una persona que ha perdido la capacidad de deambulación es aquella que tiene permanente o temporalmente limitada la capacidad de moverse sin ayuda externa, bien sea por enfermedad o por accidente. En estos casos el uso de la silla de ruedas convencional o eléctrica les ofrece una mayor autonomía en su día a día, y les permite desplazarse con libertad en su lugar de trabajo, su casa, y/o en las travesías que los separan

Sin embargo existen personas que no pueden desplazarse libremente haciendo uso de una silla de ruedas, ya sea convencional o eléctrica, debido a lesiones o enfermedades que les dificulten o impidan la realización de los movimientos con la mano necesarios para gobernar las ruedas o los motores.

En este Trabajo de Fin de Grado se pretende construir una silla de ruedas autónoma que este adaptada a las necesidades de estas personas, y les provea de la capacidad de desplazarse libremente

1.2 Introducción

Para el desarrollo de este proyecto se parte de una silla de ruedas convencional que se modificará mecánica y electrónicamente para que pueda ser controlada tanto a partir de un sistema informático instalado a bordo como de forma manual.

Para ello, la primera fase del desarrollo se centro en el manejo de los motores, para lo que se empleo un controlador de motores modelo Sabertooth 2x32, el cual nos permite controlar dos motores de forma independiente mediante un microcontrolador central Arduino MEGA, quien determina la velocidad y el sentido de giro de los motores. Con el fin de conocer la velocidad y el sentido de giro de cada rueda se emplean dos sensores acoplados al eje de giro de cada motor, denominados encoders cuadráticos, gracias a los cuales se obtiene la retroalimentación necesaria para poder controlar los motores eficientemente y generar la información odometrica. Por último se incluye una interface de control manual haciendo uso de un Joystick, este está conectado a un microcontrolador modelo Arduino UNO, quien enviá la información del sentido y la velocidad deseada por el usuario al Arduino MEGA, quien finalmente se encarga de gobernar los motores, haciendo uso de un bus de datos.

En lo que respecta a la segunda fase, se centro en la generación de un mapa del entorno circundante al vehículo, para ello se emplearon los datos odométricos recogidos por el microcontrolador central y la información capturada por un láser óptico instalado en la silla de ruedas, el cual provee de un vector con las distancias hasta los obstaculos que rodean al sistema. El procesamiento de esta información se hace desde un microprocesador modelo Raspberry Pi3, en el que esta conectado el sensor láser, y que utiliza el metasistema operativo ROS como medio de intercambio de mensajes con los demas componentes que conforman el sistema informático.

El resultado ha sido un vehículo autónomo capaz de generar un mapa de su entorno y emplear ROS para tareas de navegación

1.3 Estado del arte

Previamente al desarrollo del TFG se realizó un estudio tecnológico con el fin de investigar las herramientas y tecnologías empleadas en proyectos pertenecientes a este campo.

Existen dos grandes tipos de sistemas inteligentes implantados en vehículos en el mercado, por un lado existen los “Automated Guided Vehicles” los cuales siguen un camino prefijado siendo incapaces de evitar obstáculos existentes en el mismo, para determinar la ruta emplean un sencillo sistema de guiado tal y como son las lineas en el terreno o campos electromagnéticos entre otros métodos.

Por otro lado también existen los denominados “Autonomous Land Vehicles”, los cuales elaboran en su memoria un mapa del terreno mediante la información obtenida a través de sus sistemas sensoriales, lo que les permite evitar obstáculos y tomar decisiones en relación a la ruta a tomar.

A día de hoy ha surgido un prospero mercado de vehículos autónomos o semiautónomos, y aunque en muchos casos sus productos aun se consideran prototipos y/o proyectos ya conviven con nosotros vehículos como el “Automóvil Autónomo de Google” (Figura 1.2.2) en el ámbito civil o el “Spot de BostonDynamics” (Figura 1.2.1) en el militar.

Figura 1.3.2: A.A. de Google

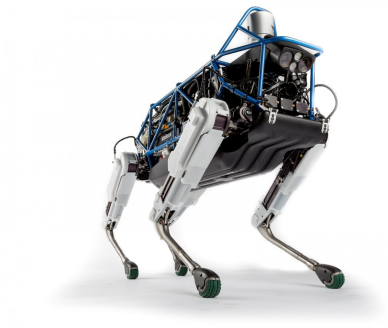


Figura 1.3.1: Spot de BostonDynamics

Con respecto a los antecedentes existentes en cuanto a la temática que nos ocupa, en la Universidad de la Laguna anteriormente se habían desarrollado proyectos tales como el vehículo “Verdino” (Figura 1.2.3) o la silla de ruedas “Perenquén” (Figura 1.2.4) ambos pertenecientes a los ya mencionados “Autonomous Land Vehicles”.

Figura 1.3.3: Verdino

Figura 1.3.4: Perenquén

1.4 Objetivos

Los principales objetivos a desarrollar son:

- Diseño e instalación de la electrónica encargada de los motores.
- Diseño y montaje de piezas 3D en la silla.
- Montaje y configuración del controlador de motores.
- Montaje de sistema sensorial de la silla.
- Diseño e instalación de un sistema de control manual.
- Diseño de la infraestructura lógica.
- Instalación y configuración de ROS en un microprocesador.
- Instalación y configuración del sensor láser en ROS.

1.5 Herramientas empleadas

Para el desarrollo del Trabajo de Fin de Grado (TFG) se han empleado diferentes herramientas, entre las cuales caben destacar:

- ***Hardware***
  - Silla de ruedas eléctrica convencional
  - Raspberry Pi3 model B
  - Sabertooth 2x32
  - Arduino (model MEGA and UNO)
  - Sweep
  - Encoder incrementales de cuadratura

- ***Software***
  - WSL (Windows Subsystem for Linux)
  - Arduino IDE
  - ROS (lunar distribution)
  - DEScribe
  - Raspbian versión 4.14

Capítulo 2
Estado del vehículo

2.1 Características mecánicas

Este capítulo se centrará en resumir y explicar las diferentes opciones que se tomaron durante el diseño y desarrollo de la mecánica de nuestro vehículo.

2.1.1 Estructura de construcción

Originalmente, la silla de ruedas de la marca “Sunrise Medical” modelo F35 R2 (Figura 2.1.1) estaba conformada por:

- Un chasis metálico robusto (Figura 2.1.2)
- Dos motores con reductora.
- Dos baterías de plomo de hasta 50 Amph.
- Dos ruedas motrices y dos ruedas locas.
- Frenos magnéticos.

Figura 2.1.1: Original

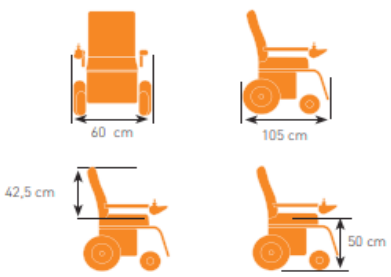


Figura 2.1.2: Dimensiones

Durante el desarrollo del proyecto, se fueron incorporando y/o modificando diferentes elementos del vehículo, algunos de los más destacables fueron:

- Se añadió una placa de polímero plástico en la parte trasera del respaldo de la silla para dar soporte a la electrónica. (Figura 2.1.3)
- Se diseñó y fabricó una pieza en 3D que permitiera el acople de encoders ópticos a los motores de la silla de ruedas.
- Se modificaron los frenos para poder hacer pasar a través de ellos la pieza, ya diseñada, que haría de conexión entre el eje del motor y los sensores encargados de capturar el movimiento, los Encoders Cuadráticos. (Figura 2.1.4)
- Se diseñó y creó la electrónica necesaria para permitir el manejo manual de la silla a partir de un Joystick de doble precisión.
- Se añadieron soportes para los sistemas electrónicos encargados del control manual de la silla de ruedas bajo el mando. (Figura 2.1.5)
- Se incorporó el sensor láser al vehículo para obtener información de las distancias circundantes.

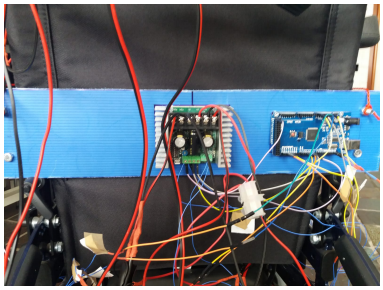


Figura 2.1.3: Soporte


Figura 2.1.4: Mando


Figura 2.1.5: Frenos

2.1.2 Alimentación

Con respecto a la alimentación del vehículo, tal y como se especificó en el apartado anterior, se dispone de 2 baterías conectadas entre ellas que suministran aproximadamente 50 Amph, con los que se pueden alimentar a todos los sistemas.

Como método de carga para las baterías se dispone de un cargador a 8 Amph que nos permite recargar las baterías en un muy breve periodo de tiempo.

Como sistema de protección se ha empleado un fusible de bayoneta de 50 Amph el cual se ha situado entre los componentes y las baterías, con lo que se evita que algún elemento del sistema pudiera sufrir algún daño provocado por un pico de corriente, este sistema de protección se encuentran entre la fuente de la alimentación y el controlador de motores, a partir del cual se alimenta al resto de componentes.

Por último les mostramos el esquema eléctrico de conexiones (Figura 2.1.6)

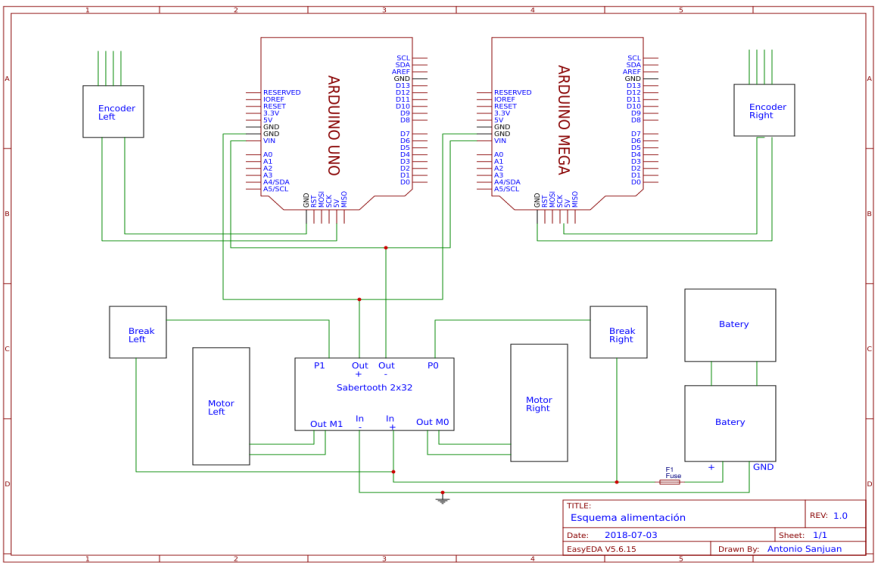


Figura 2.1.6: Esquema

2.1.3 Cinemática

En lo que respecta a la configuración de la cinemática se hace uso de un modelo cinemático diferencial ya que con él se puede determinar con facilidad las relaciones entre las velocidades de cada una de las ruedas y las del extremo del robot.

Físicamente la silla de ruedas se compone de dos ruedas motrices traseras alineadas en un mismo eje, y otras dos ruedas locas delanteras en paralelo a estas primeras, esto ocasiona que la obtención de los datos odométricos sean mucho más fiables de lo que serian si se dispusiera de otro par de ruedas locas, tal y como sucede en versiones como el "Perenquén" desarrollado por la ULL, el cual padece de problemas de tracción y/o patinaje (Figura 2.1.7) por tener dos pares de ruedas locas.

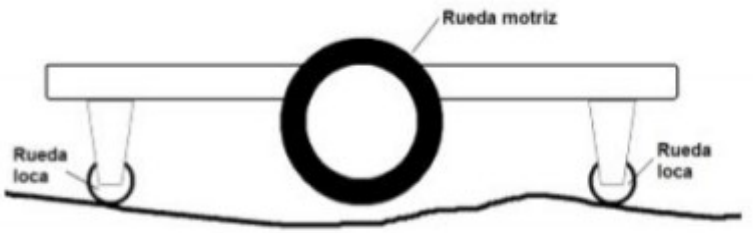


Figura 2.1.7: Problemática del doble apoyo

Para la ejecución de movimientos se inyectan diferentes valores en el controlador de los motores, el cual nos permite controlar la dirección y la velocidad de los motores mediante entradas digitales, una de esas entradas es el identificador del motor, mientras que la otra es una representación de la velocidad deseada mediante una modulación por ancho de pulsos o PWM.

2.1.4 Hardware

En cuanto al hardware empleado:

El hardware utilizado para el *operador de control* consta de 2 microcontroladores diferentes, por un lado y para el *subsistema maestro* se empleó un Arduino MEGA (Figura 2.2.1) por la gran cantidad de puertos I/O, interrupciones y memoria que nos ofrecía.

Para el *subsistema esclavo* se determinó emplear un Arduino UNO (Figura 2.2.2), aprovechando su bajo coste y las prestaciones de las que dispone.

Arduino es una plataforma de hardware y software libre, por lo que presenta la oportunidad de poder rediseñar un modelo que se adapte perfectamente a nuestras necesidades a un precio realmente competitivo en próximas modificaciones.

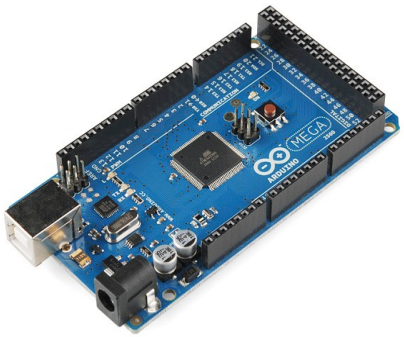


Figura 2.1.8: Arduino MEGA

Figura 2.1.9: Arduino UNO

Por otro lado el hardware que se decidió emplear para el *operador de planificación* fue un microprocesador que pese a que no fuera costoso ofreciera unas buenas prestaciones, es por eso que nos decantamos por una Raspberry Pi3 model B, (Figura 2.2.3) dado que permitía también utilizar una gran cantidad de SSOO's y una enorme diversidad de lenguajes de programación

Figura 2.1.10: Raspberry Pi3 model B

	Mega	Uno
Procesador	*Atmega 2560*	*Atmega 328*
V. de operacion	*5V*	*5V*
Vin recomendado	*7-12V*	*7-12V*
Vin límite	*6-20V*	*6-20V*
I/O Digital Pins	*54 (14 PWM)*	*14 (6 PWM)*
I/O Analog Pins	*16*	*6*
Interrupciones	*6*	*2*
Corriente I/O Pin	*40 mA*	*40 mA*
Memoria Flash	*256 KB*	*32 KB*
SRAM	*8 KB*	*2 KB*
EEPROM	*4 KB*	*1 KB*
Frecuencia de reloj	*16 MHz*	*16 MHz*

Tabla 2.1 1: Características de microcontroladores

	Raspberry Pi3 model B
Procesador	*ARM11*
RAM	*512 MB*
Memoria	*SD 8GB*
USB	*4*
SSOO	*Raspbian*
Ethernet	*10/100 Mbps Ethernet RJ-45*
Video	*Micro HDMI*
Frecuencia de reloj	*770MHz*

Tabla 2.1 2: Características de microprocesador

2.2 Estructura de control

El diseño lógico implementado en el sistema consta de dos elementos, por un lado se dispone de un *operador de control*, está conformado por los dos mirocontroladores, y es el encargado de llevar a cabo operaciones directamente relacionadas con los sensores y actuadores del vehículo, mientras que, por otro lado se define el *operador de planificación*, el cual se compone del microprocesador y cuya labor es la de recopilar toda la información emitida por *operador de control*, y tras una valoración de la misma expedirle un comando con la siguiente acción a realizar.

Con el fin de simplificar una división de tareas para con ambos operadores se determinó que el *operador de control* sería la encargado de hacerse cargo del control a bajo nivel de los componentes que conforman el sistema, y que correspondería a el *operador de planificación* encargarse del control a alto nivel.

El *operador de control* se compone a su vez de dos subsistemas, un maestro y un esclavo comunicados entre si mediante un bus de datos I_2C, el *subsistemas esclavo*, compuesto por el microcontrolador Arduino UNO, es el encargado de permitir el control manual del vehículo, mientras que el *subsistema maestro*, compuesto esta vez por el Arduino MEGA, se ocupa de examinar los sensores, gobernar los frenos y dirigir los motores. El *subsistema maestro* comprueba el estado de los sensores acoplados a los ejes de los motores, esto se efectúa con el fin de mantener una copia precisa de cual ha sido el avance y/o retroceso que se ha llevado a cabo en cada una de las ruedas, esta información se complementa con las dimensiones de la silla de ruedas para obtener la información, en metros, tanto del avance como

de la velocidad y la dirección, es con esto con lo que *subsistema maestro* genera un paquete con los datos odométricos del sistema.

Para conocer y manipular la velocidad del vehículo se empleo un algoritmo de control PID en Arduino. El controlador PID es un mecanismo de control por realimentación a partir del cual se gobierna y conoce la velocidad del motor, esta conformado por tres parámetros

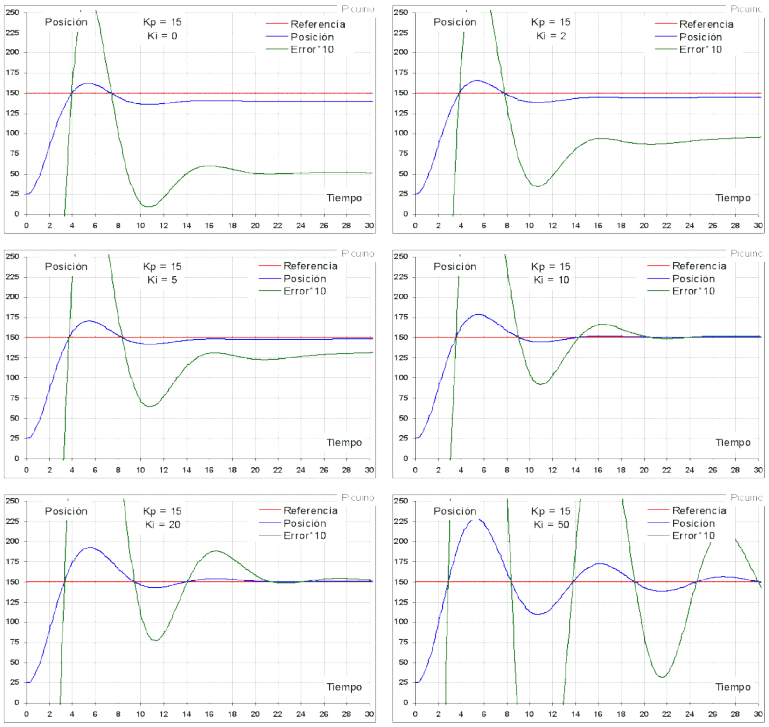


Figura 2.2.1: Ajuste PID

- El valor *proporcional* depende de la diferencia entre el valor o la velocidad deseada y la obtenida.

- El valor *integral* depende de la suma de la diferencia de los errores pasados.

- El valor *derivativo* es una predicción de los errores futuros en relación a los errores previos.

A través del ajuste manual de las constantes Kp, Ki y Kd relacionadas directamente con cada uno de los parámetros se pudo llegar a probar y ajustar el comportamiento del sistema, estabilizándolo y adecuándolo a su propósito.

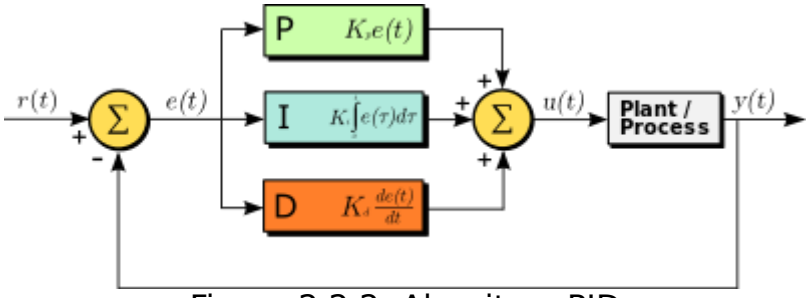


Figura 2.2.2: Algoritmo PID

La odometría es el estudio de la posición durante la navegación de vehículos con ruedas. Para esto usan información sobre la rotación de las ruedas para estimar cambios en la posición.

Como se conoce el tiempo y la velocidad de ejecución de los movimientos se puede obtener fácilmente la odometría, para ello deberemos de conocer las proporciones de nuestro vehículo, tales como la distancia entre las ruedas, el diámetro de estas, entre otros muchos parámetros.

Con estos datos se calculan tanto la posición como la velocidad angular y lineal, de manera que se obtiene cual ha sido el avance total realizado, lo que permite que se pueda generar el paquete odométrico de ROS

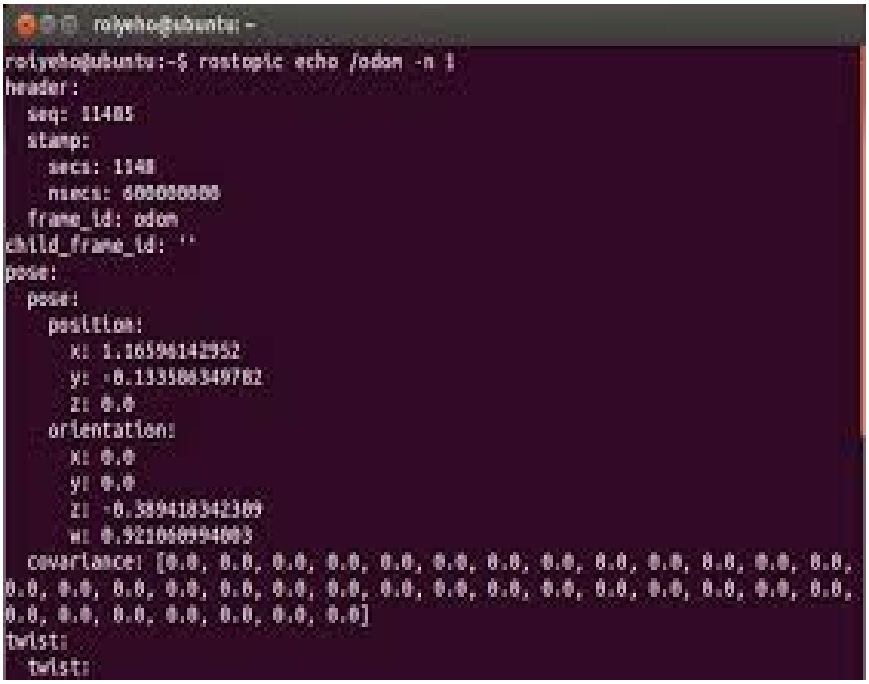


Figura 2.2.3: Paquete odométrico

Por otro lado el *subsistema esclavo* se ocupa de obtener y filtrar constantemente la información que obtiene del joystick, encargado del control manual, para que cuando el *operador de control* maestro realice una petición por el bus, este pueda responder con la información obtenida.

La interface de control manual que se emplea en el proyecto es un Joystick de doble precisión, este dispositivo provee de dos salidas para cada uno de sus dos ejes de movimiento, de esta manera se dobla el rango de valores posibles aunque se disminuye notablemente la dispersión de los mismos.

Cada una de las salidas del Joystick esta conectado a un potenciómetro, estos permiten leer un rango de entre 0 y 5V captando cambios mayores de 0.004V, por lo que se obtiene una resolución de 10bits, lo que a su vez significa que el valor de salida estará comprendido entre 0 y 1023. Con el finde obtener un resumen de los datos coherente con las salidas de un mismo eje se realiza la media y se modula el dato para poder enviarlo a través del bus.

El *operador de planificación* recopila tanto los datos obtenidos a partir de los paquetes de odometría enviados por el *operador de control* como aquella que obtiene directamente desde el Sweep (laser scanner), analiza esta información y responde al *operador de control* con un mensaje en el cual se define la siguiente acción a realizar.

2.2.1 Control del sistema sensorial

El sistema sensorial con el que cuenta el vehículo se puede dividir en dos tipos según el nivel de abstracción en el que trabaja cada uno.

Por un lado y trabajando al nivel más bajo se utilizan los encoders cuadráticos o de cuadratura, estos son unos sensores que se acoplan al eje del motor entregando información con la que más tarde se calculará la velocidad y la dirección del giro gracias a sus dos canales.

Para determinar la dirección del giro aprovechamos que las señales de salida correspondientes a los canales de los que disponen los encoders cuadráticos estas desfasadas 90 grados, por lo que conociendo el estado de la salida actual y previa de estos sensores se puede determinar la dirección, mientras que la velocidad se calcula a partir de la cantidad de cambios en el estado de la salida en un periodo de tiempo concreto. (Figura 2.2.4).

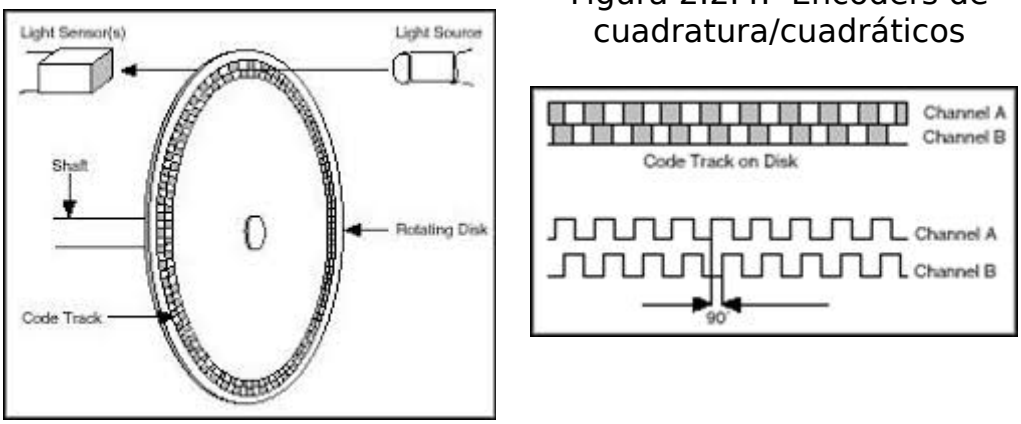


Figura 2.2.4: Encoders de cuadratura/cuadráticos

Para el desarrollo de la silla de ruedas se emplearon dos HEDS-5600, (Figura 2.2.5) el cual nos ofrece 1024 pulsos por vuelta, asegurando cierta precisión en el calculo de la velocidad.

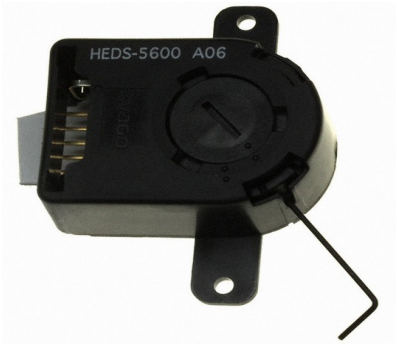


Figura 2.2.5: HEDS-5600

Gracias a la eficiencia de estos sensores es posible generar un paquete de odometría básico, el cual será complementado más tarde con la información capturada por el Sweep, con lo que se podrá elaborar un mapa del terreno circundante a la silla durante su recorrido.

Por otro lado se dispone de un Sweep Laser Scan de la compañía “Scanse” (Figura 2.2.9), el cual trabaja a un nivel de abstracción mucho mayor que los ya comentados encoders cuadráticos. Sweep es un sensor LIDAR de escaneo diseñado para proporcionar una detección de 360 grados a su alrededor.


Figura 2.2.6: Sweep LIDAR sensor

Los sensores LIDAR emplean tele-detección óptica usando la luz del láser para obtener la diferencia del tiempo de retorno de la luz con respecto a su emisión, lo cual se traducirá en la distancia que separa el Sweep de la superficie analizada.

En el sistema a desarrollar se emplearán los datos recogidos por el Sweep como contraste ante la información odométrica que se obtenga del microcontrolador, de esta manera se tendrá en consideración que el vehículo puede derrapar a causa del posicionamiento de las ruedas locas, o debido a una pendiente y/o inclinación en el terreno, lo que provocaría que los estos datos enviados por el microcontrolador en lo que a odometría se refiere se vieran comprometidos.

2.2.2 Acople del sistema sensorial

Con el fin de acoplar el encoder cuadrático al eje del motor, se optó por diseñar el modelo de una pieza en 3D para más tarde imprimirla en PLA, un polímero versátil empleado en la gran mayoría de impresoras 3D comerciales, y así finalmente comprobar su eficiencia desplegándolo en el sistema.

Las primeras fases de su desarrollo estuvieron marcadas por la toma de medidas y de datos para más tarde poder realizar un primer modelo (Figura 2.2.6) que cumpliera con los requerimientos iniciales básicos.

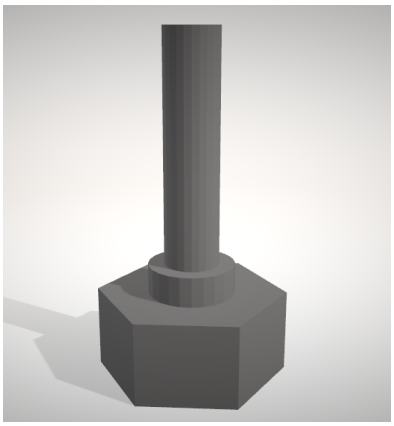


Figura 2.2.7: 3D primer modelo

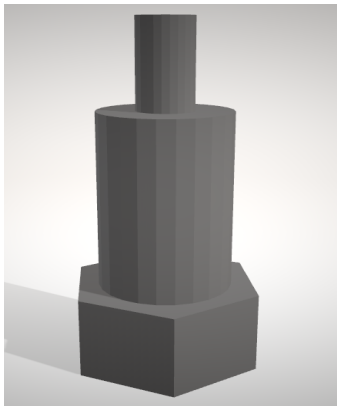


Figura 2.2.8: 3D modelo final

- Fácil de fabricar mediante una impresora 3D o una fresadora CNC.
- Capacidad de realizar modificaciones sobre la pieza una vez impresa.
- Baja densidad del material de producción.
- Resistencia al peso.
- Durabilidad.
- Conductividad térmica

Con las constantes pruebas realizadas se ratificó la susceptibilidad al calor de este polímero, un problema que se trato de solucionar en un primer momento realizando modificaciones (Figura 2.2.7) sobre el modelo de la pieza, aunque con estos cambios se consiguieron mejoras con respecto a su tiempo de vida antes de que a causa de las emisiones de calor provocadas por los frenos magnéticos y de la fricción, resultaran en el reblandecimiento y la deformación de la pieza, lo cual originaba fallos en cadena ocasionados a partir de los datos erróneos obtenidos por los sensores.

Figura 2.2.9: 3D modificaciones del modelo

Finalmente se decidió cambiar el material de impresión de la pieza, para ello se escogió el aluminio como material idóneo para esta tarea debido su gran ductabilidad y muy bajo peso. Al tratarse de un material tan difícil de deformar, se obtuvo una pieza (Figura 2.2.8) que cumplía con los siguientes requisitos comentados.

2.2.3 Control de motores

En lo que respecta al control de los motores, pese a que su comportamiento es definido y calculado desde un microcontrolador modelo “Arduino MEGA 2560” este no realiza un control directo sobre los mismos, para poder controlarlos hacemos uso de un controlador de motores, el Sabertooth 2x32 (Figura 2.2.10) es un controlador de motor de doble canal capaz de suministrar de 32 a 64 Amph a dos motores.

Figura 2.2.10: Sabertooth 2x32A

El Sabertooth 2x32 dispone de un DIP switch gracias al cual será posible configurar las salidas, entradas, el tipo de comunicación, las salidas auxiliares, el tipo de alimentación y procedencia entre otras muchas características, lo cual facilita su implementación.

Para el correcto desarrollo del sistema se conformó un modelo de comunicación packet serial entre el controlador de motores y el microcontrolador , esto se realizó con el fin de simplificar el paso de mensajes entre estos dos elementos, para ello se emplearon las dos entradas seriales de las que dispone el controlador, una entrada para regular el comportamiento del motor (velocidad y sentido de giro) y otra para determinar el motor en cuestión.

Algunas configuraciones del Sabertooth 2x32, como es el caso de la configuración de los puertos de salida auxiliares, empleados para dar soporte a los frenos magnéticos, debe de ser determinados y configurados a partir de un aplicación software denominada DEScribe (Figura 2.2.11), la cual nos ofrece de forma gratuita la compañía desarrolladora del producto.

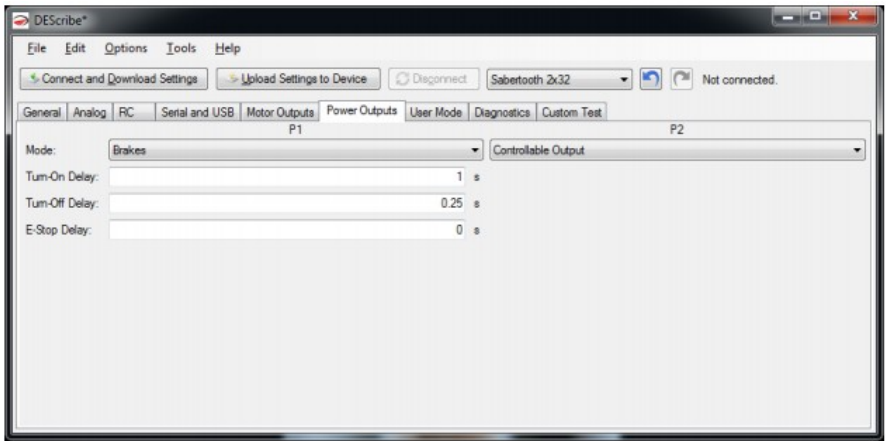


Figura 2.2.11: DEScribe software

2.2.4 Control manual

El control manual es una de las características con las que cuenta el sistema, se implementó para que fuera posible ejercer un controlar manual sobre el vehíiculo sin necesidad de hacer uso del control automático del sistema, esta característica se implementó debido a que existen situaciones y/o espacios en los que no es viable hacer uso del control automático.

En cuanto su diseño, (Figura 2.2.12) dispone de un Joystick de doble precisión a partir del cual se puede extraer tanto la dirección del avance como la velocidad deseada por el usuario.

Como se podrá recordar de capítulos anteriores, el sistema de control manual forma parte de uno de los dos operadores que conforman este proyecto, más concretamente del *operador de control*, éste estaba formado por dos elementos, un *subsistema maestro* encargado de gobernar los motores y un *subsistema esclavo*, el cual se pasará a tratar a continuación.

El control manual tiene por función mantener una copia del estado del joystick acondicionada, y proveer de esta copia al *subsistema maestro* cuando este lo requiera, para de esta forma poder adecuar el comportamiento de los motores a la velocidad deseada.

Para esto, el *subsistema esclavo* deberá de realizar constantemente la siguiente rutina:

- Capturar la información de los potenciómetros
- Filtrar el ruido de la señal
- Modular la información
- Preparar el tipo de la información para el envío

Cuando reciba una petición mediante el bus I_2C por parte del *subsistema maestro*, este le contestará con la última información acondicionada.

Esta comunicación se realiza a través de un bus de datos I_2C establecido entre el subsistema de control maestro y el esclavo.

Capítulo 3
Robot Operating System

3.1 Introducción a ROS

ROS, del ingles Robot Operating System es un framework de código abierto destinado al desarrollo de software para robots a partir de la reutilización de código, provee de los servicios que cabría esperar de un sistema operativo, tales como son la abstracción del software, el control de dispositivos de bajo nivel y los intercambios de mensajes entre otros muchos.

Proporciona además librerías y herramientas para con ellas escribir, compilar y ejecutar código a través de múltiples computadoras mediante un modelo publisher - subscriber

Con la ejecución de ROS se genera una red P2P a la cual se conectan diferentes ejecutables para, empleando los canales de comunicación que ROS suministra, compartir información entre ellos.

Cabe destacar que pese a su nombre, no se trata de un sistema operativo en si mismo, sino de un metasistema operativo que presta determinados servicios haciendo uso de herramientas que le proporciona el sistema operativo desde el que se ejecuta.

3.2 Conceptos

Existen conceptos que resultan de una importancia enorme cuando estas trabajando con ROS.

- ***Master:*** Elemento central de la red de ROS, permite la búsqueda y el registro de nodos. Sin este elemento no se podrían encontrar los nodos entre si, ni tampoco intercambiar mensajes.

- ***Nodos:*** Los nodos son ejecutables que se comunican con otros nodos mediante tópicos. Un nodo puede ser escrito en diferentes lenguajes de programación (C++, Python, Java, ...).

Los nodos pueden convertirse en publishers de un tópico en caso de querer recibir información de ese canal, del mismo modo, pueden registrarse como subscriber de un tópico concreto si desean enviar información por él.

- ***Mensajes:*** Dos nodos se comunican por medio del paso de mensajes. Los mensajes no son más que estructuras de datos que combinan tipos de datos básicos y otras estructuras incrustadas en su interior.

- ***Tópicos:*** Se tratan de canales a partir de los cuales los nodos pueden comunicarse haciéndose subscriber o publisher del mismo.

Son un canal de comunicación destinado a ser empleado para envíos asíncronos de datos.

Pueden existir múltiples nodos tanto publishers como subscribers concurrentemente para un mismo tópico, al igual que un solo nodo puede ser publisher y subscriber de más de un tópico.

- ***Servicios:*** Los servicios al igual que los tópicos son un medio de comunicación, aunque al contrario que estos, los servicios son empleados para comunicaciones síncronas entre nodos.

3.3 Entendiendo ROS

Todo comienza con roscore. Se trata de un conjunto de nodos y herramientas indispensables para la correcta ejecución de las funcionalidades y características de ROS.

Cuando roscore es ejecutado, éste se hace cargo de lanzar el nodo Master, quien permite la búsqueda y el registro de otros nodos.

Una vez se ha lanzado el nodo Master, se pueden comenzar a ejecutar el resto de los nodos que conforman nuestro sistema.

Si un nodo se comunica con el resto de nodos enviando mensajes mediante un tópico (Figura 3.3.1), es el nodo Master el que se encarga tanto de informar al nodo publisher (emisor) de la ubicación en donde se espera el mensaje, como de transmitir a todos los nodos subscriber (receptores) de dicho tópico donde acceder a la información.

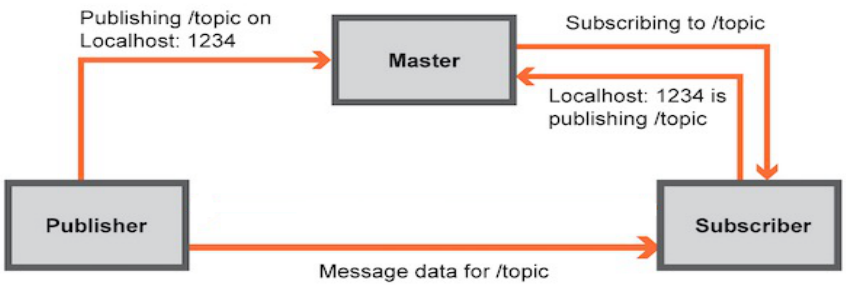


Figura 3.3.1: Entendiendo ROS

3.4 Utilización de ROS en el proyecto

A continuación se procederá a explicar el papel que cumple ROS en el proyecto. Su función, tal y como se detalla puede ser comprendida como dos tareas, que una vez realizadas aportan como resultado un mapa del entorno

Tal y como se profundizará en próximos capítulos, la distribución de ROS instalada y preparada para la raspberry Pi3 model B, ha sido complementada con los paquetes *rosserial-arduino*, el cual proporciona de un medio de comunicación serial directo con el microcontrolador , y *sweep-ros*, paquete gracias a el cual obtenemos los datos de distancias capturados por el Sweep.

Por un lado, en lo que respecta a la primera tarea a desempeñar por ROS (Figura 3.4.1) en el proyecto es la de proporcionar funcionalidades y medios al *subsistema maestro*, el cual forma parte del *operador de control*, que es quien tiene la función de elaborar y publicar los mensajes de odometría a través del tópico ***/odometry*** y los de registros o debug por medio de ***/log_ardu***.

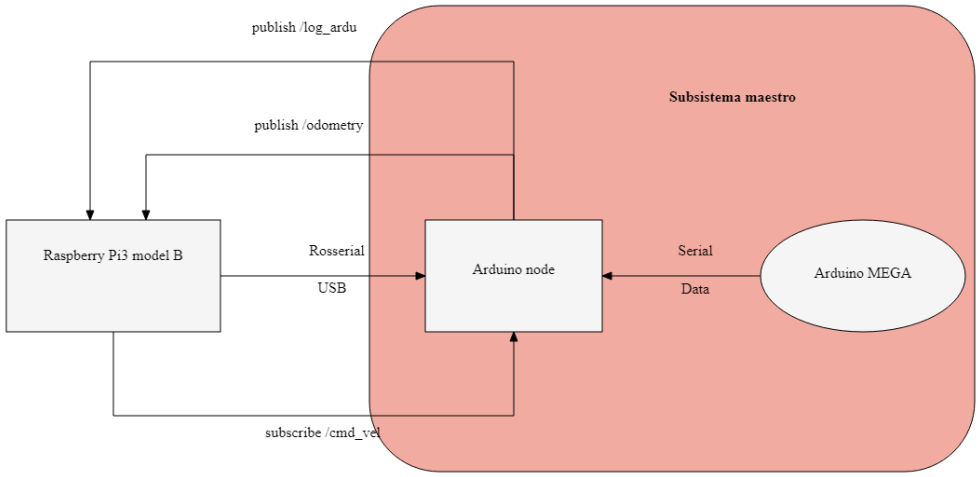


Figura 3.4.1: ROS part1

El *subsistema maestro*, al mismo tiempo, es un nodo subscrito al topic ***/cmd_vel*** a partir del cual recibe mensajes desde el *operador de planificación* con la velocidad deseada para el vehículo.

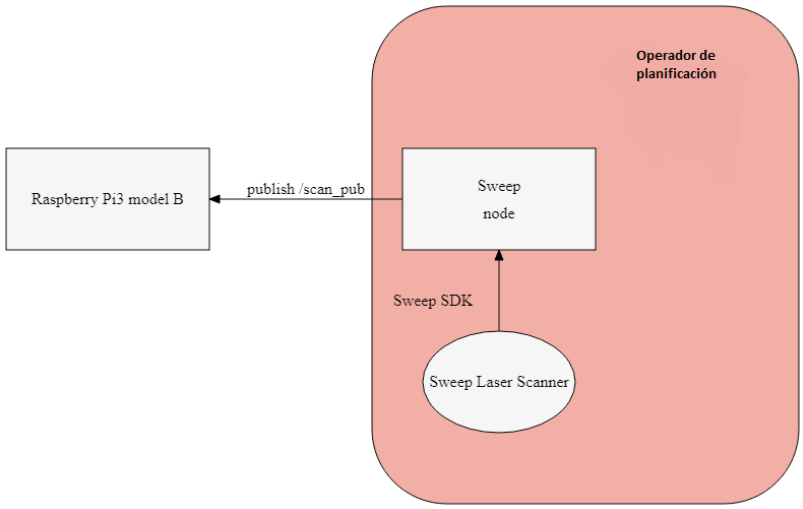


Figura 3.4.2: ROS part2

Por otro lado y en paralelo a la realización de la primera tarea se lleva a cabo la segunda (Figura 3.4.2), cuyo fin es el de, prestando apoyo al *operador de planificación*, contrastar la información odométrica recibida mediante el tópico ***/odometry*** con objeto de ratificar la veracidad de la información obtenida, para de este modo evitar la incoherencia de datos producidos por patinazos, resbalamientos y/o a causa del terreno. Con el fin de obtener este contraste, se emplean los datos capturados mediante el Sweep laser scanner incorporado en la silla de ruedas. Gracias a el cálculo de la diferencia existente entre ambas medidas podemos determinar el posicionamiento relativo de nuestro sistema, lo que permite elaborar el mapa

Un grafo del flujo que represente de forma simple las comunicaciones efectuadas por los elementos que conforman el sistema mediante los nodos de ROS sería el correspondiente a la Figura 3.4.3.

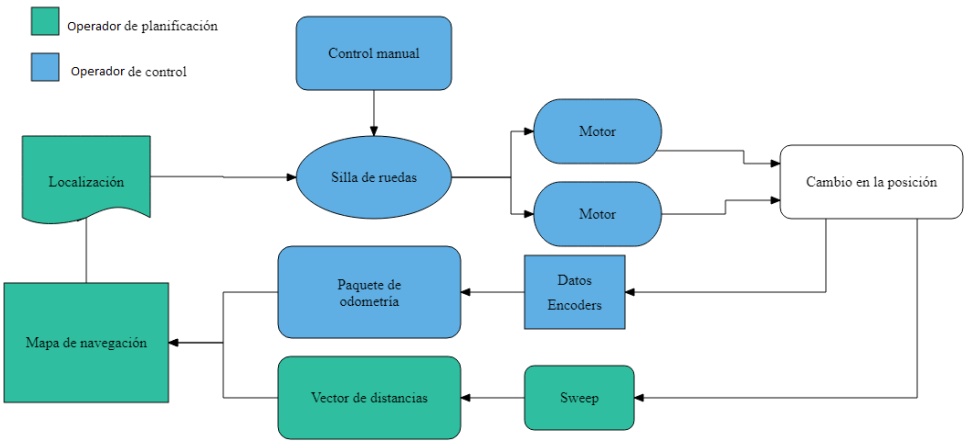


Figura 3.4.3: Grafo del sistema

Tal y como se puede observar tanto el *operador de control* como el *operador de planificación* del sistema emplean los medios que ROS proporciona para poder realizar el intercambio de datos entre los nodos.

En cuanto a la distribución, se puede percibir como mientras que el *operador de planificación* requiere de los datos emitidos por el *operador de control* para poder ejecutarse, el *operador de control* por su parte puede funcionar tanto independientemente del *operador de planificación* empleando las órdenes emitidos por el usuario mediante el subsistema de control manual, como aplicando los comandos enviados por el *operador de planificación* sirviéndose las funcionalidades de ROS.

De este modo se consigue paralelizar la ejecución de los elementos lógicos del sistema, gracias a la distribución planteada es muy sencillo separar el control manual del control automatizado de la silla de ruedas, manteniendo siempre en ejecución los procesos del *operador de planificación*.

Capítulo 4
Raspberry Pi

Tal y como se explicó en el apartado “Hardware” perteneciente al capitulo 2, la plataforma escogida para dar soporte al *operador de planificación* será una Raspberry Pi 3 model B.

Se trata de un microprocesador que nos ofrece unas características muy competitivas tal y como se puede observar en el apartado especificado, (Tabla 2.2) además de contar con una activa comunidad de usuarios siempre dispuesta a colaborar.

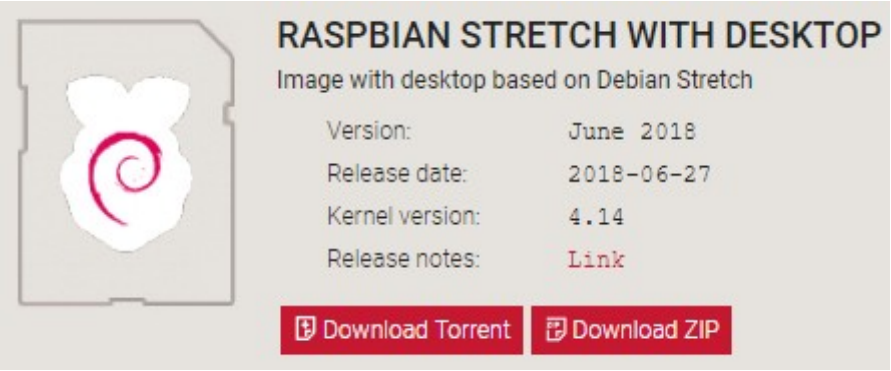
4.1 Instalación del sistema operativo.

La raspberry Pi 3 model B cuenta con multitud de sistemas operativos y gestores de contenido listos para instalarlos en ella, desde completas versiones de Linux hasta reducidas versiones de Windows pasando por una infinidad de gestores de contenido diferentes.

Finalmente y tras la valoración de varias opciones se determinó que sería Raspbian el sistema operativo escogido para hacerse cargo del *operador de planificación*, esta decisión se tomo en torno a la capacidad del sistema operativo de aprovechar al máximo el potencial del microprocesador y la compatibilidad del mismo con aplicaciones y herramientas de otras distribuciones de Linux.

La instalación del sistema operativo requiere únicamente de una SSD de al menos 8GB, los pasos a seguir para instalar el SSOO se estipulan a continuación:

1. Descargar la imagen del SSOO a instalar, en el caso de nuestro sistema un Raspbian versión 4.14 (Figura 3.1.1)



2. Descomprimir el fichero y copiar el contenido del mismo en la SSD previamente formateada.

3. Introducir la tarjeta SSD en la Raspberry Pi 3 e iniciar el microprocesador, el proceso de instalación comenzara automáticamente.

4. Una vez finalizada la instalación procedemos a la desinstalar y eliminar todas las aplicaciones y herramientas innecesarias, esto se efectúa debido al poco espacio de almacenamiento del que disponemos.

5. A continuación faltaría por instalar y preparar la distribución ROS que se desee en la Raspberry Pi para su uso.

6. Finalmente se completan las funcionalidades que ROS aporta mediante paquetes que distribuye la comunidad.

4.2 Instalación de ROS

Una vez el microprocesador esta listo para comenzar a trabajar con él, procedemos a instalar la distribución Kinetic ROS.

1. Se configuran los repositorios y claves

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/s
ources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F7178
15A3895523BAEEB01FA116
```

Figura 4.2.1: Setup ROS Repositories

2. Se confirma la actualización del package index

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Figura 4.2.2: Package index

3. Se instala la distribución Kinetic ROS

```
sudo apt-get install ros-kinetic-desktop-full
```

Figura 4.2.3: Installation

4. Se inicializa rosdep

```
sudo rosdep init
rosdep update
```

Figura 4.2.4: Initialize rosdep

5. Configuración del entorno

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

Figura 4.2.5: Enviroment variables

Al finalizar estos pasos, ROS kinetic estará instalado en el microprocesador, únicamente faltará por instalar los siguientes paquetes para tener el *operador de planificación* preparado para el despliegue.

- *Rosserial-arduino*

Brinda multitud de funcionalidades a Arduino IDE y aporta un valioso protocolo de comunicación entre Arduino y Raspberry Pi, empleando para ello los enlaces seriales del bus UART al que se conectan ambos elementos.

- *Sweep-ros*

Este paquete requiere de la librerías pertenecientes al kit de desarrollo de software de Sweep (Sweep-SDK) para poder ejecutarse, y ofrece la oportunidad de poder comunicar el lasser Scan y ROS.

Capítulo 5
Sistema de Mapeo

En este capítulo se hará énfasis en el sistema de mapeo, éste está destinado a elaborar un mapa del terreno circundante al vehículo durante su recorrido.

5.1 Descripción general

En un comienzo se generalizará la descripción del sistema, para con el tiempo ir especificando cada una de las tareas que se realizan, junto con el tipo de comunicación que establece cada componente.

Para comenzar, tal y como se ha tratado en el capítulo 2, existen dos requisitos para elaborar el mapa del terreno que rodea a un sistema móvil, uno de ellos es conocer las distancias que rodean al sistema, mientras que el otro es el de mantenerse informado del desplazamiento efectuado por él.

Como se recordará, el vector de distancias obtenido es producto del barrido laser que realiza el Sweep Laser Scanner incorporado al proyecto, mientras que los datos que determinan los movimientos realizados por la silla de ruedas, se obtienen mediante los paquetes de odometría emitidos de forma regular por el *operador de control*, quien genera estos paquetes en base a la información capturada del sistema sensorial al que esta conectado.

Finalmente, conociendo la distancia y la velocidad a la que se mueve la silla de ruedas gracias a los paquetes odométricos, y recibiendo paralelamente información acerca de las distancias que rodean a la silla de ruedas mientras se desplaza es posible generar el mapa.

5.2 Preparación

Esta preparación será de una vital importancia para poder convertir datos en un mapa que represente el entorno cercano al que circula la silla de ruedas.

En lo que respecta a la preparación requerida para que el sistema pueda manejar la inmensa cantidad de información que se envían los nodos entre ellos será necesario hacer uso de las comunicaciones que ROS provee.

Para ello se deberá de ejecutar:

- **roscore**
    - Necesario para dar soporte a ROS. La ejecución debe de mantenerse en segundo plano mientras los nodos requieran de funcionalidades de ROS como la comunicación.
- **rosrun rosserial_python serial_node.py** ***/dev/ttyS3***
    - Implementación en Python del lado del servidor de rosserial.

    Gestiona la configuración, publicación y suscripción de un dispositivo, el cual esté conectado al puerto USB */ttyS3*, con rosserial habilitado.

- ***roslaunch sweep_ros view_sweep_laser_scan.launch***
    - Herramienta a partir de la cual se pueden lanzar con facilidad múltiples nodos de ROS así como establecer parámetros para estos nodos.

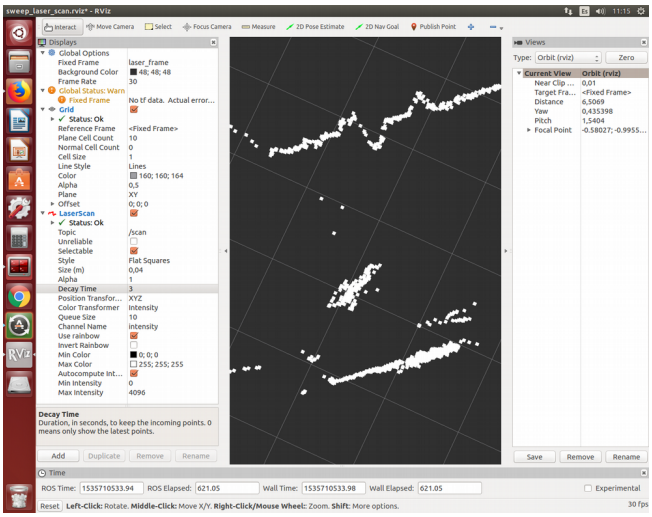


Figura 5.2.1: Rviz visualización

Con la aplicación de estos comandos, se estarán ejecutando múltiples nodos de forma simultanea, siendo posible que compartan información entre ellos con el fin de que construyan juntos el mapa del entorno,

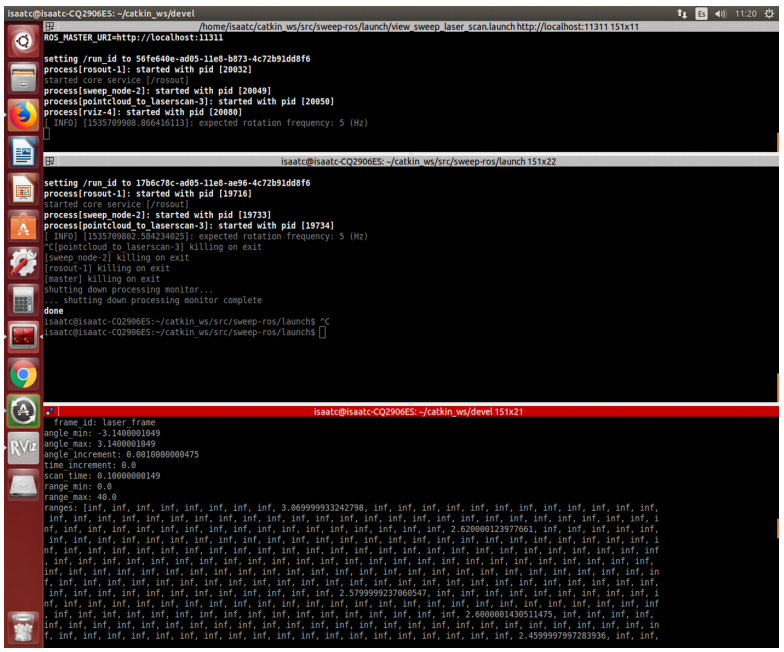


Figura 5.2.2: Ejecución de nodos ROS

Capítulo 6
Plan de trabajo

La gestión del trabajo correspondiente a este proyecto fue diseñada siguiendo los fundamentos de la metodología ágil, este modelo de desarrollo permite una alta adaptabilidad ante problemas que pudieran manifestarse durante la realización de ampliaciones en el sistema, al mismo tiempo que permite evaluar y probar cada uno de los módulos antes y después de su despliegue, lo que permite corregir errores que no fueron previstos en la etapa de planificación.

Durante la etapa de planificación del proyecto se determinó que las principales fases o módulos que se debían de completar para obtener un producto eran las siguientes:

Tarea	Definición
(1)	Estudio y definición de requisitos.
(2)	Diseño e instalación de la electrónica encargada de los motores.
(3)	Diseño y montaje de piezas 3D en la silla.
(4)	Montaje y configuración del controlador de motores.
(5)	Montaje de sistema sensorial de la silla.
(6)	Diseño e instalación de un sistema de control manual.
(7)	Despliegue de ROS y diseño de la infraestructura lógica.
(8)	Preparación de la Raspberry Pi3
(9)	Instalación y configuración de Sweep en ROS.

Tabla 6 1: Fases del desarrollo

Con el fin de llevar a cabo con éxito el desarrollo del sistema en el tiempo establecido se elaboró un cronograma o diagrama de Gannt, para con él poder distribuir las tareas dentro del intervalo de tiempo del que se dispone.

Tarea	**Días**	**Mes 1** (Feb)	**Mes 2** (Mar)	**Mes 3** (Abril)	**Mes 4** (May)	**Mes 5** (Jun)	**Mes 6** (Jul)	**Mes 7** (Agos)
(1)	25							
(2)	30							
(3)	20							
(4)	25							
(5)	13							
(6)	25							
(7)	20							
(8)	15							
(9)	30							

Tabla 6 2: Cronograma

Capítulo 7
Presupuesto

En este capítulo se llevara a cabo un estudio económico del proyecto, enumerando tanto los gastos asociados al desarrollo del sistema como el coste de los materiales empleados y de la mano de obra.

7.1 Gastos directos

Los gastos directos son aquellos que se asocian con el producto de manera directa. Dentro de este conjunto de gastos algunos de los más habituales son:

- Materias primas.
- Mano de obra directa.

7.1.1 Gastos en material

A continuación se enumerara el coste asociado a los materiales necesitados para la construcción del sistema, entre los cuales cabe destacar:

- Silla de ruedas eléctrica. (Tabla 6.1)
- Tornillería variada. (Tabla 6.1)
- Electrónica básica. (Tabla 6.2)
- Elementos eléctricos. (Tabla 6.3)
- Cableado.

Gastos en elementos mecánicos			
Concepto	**Coste**	**Unidades**	**Coste Total**
Silla de ruedas F35R2 de “Sunrise Medical”	*2000€*	*1*	*2000€*
DIN 7985 M4-14	*0.16€*	*15*	*2.4€*
DIN 439	*0.15€*	*15*	*2.25€*
DIN 7380 M6-16	*0.15€*	*10*	*1.5€*
DIN 7973 ST3.5-13	*0.15€*	*10*	*1.5€*
DIN EN ISO 7046	*0.16€*	*10*	*1.6€*
DIN 6921 M5-12	*0.10€*	*4*	*0.4€*
DIN 439 PASO FINO	*0.15€*	*4*	*0.6€*
Soportes varios (bridas, abrazaderas,...)	*1.5€*	*3*	*4.5€*
Placas de PVC rígido 5mm	*6€*	*1*	*6€*
		Total	*2020.75€*

Tabla 7.1 1: Gastos en materiales mecánicos

Gastos en electrónica básica			
Concepto	**Coste**	**Unidades**	**Coste Total**
Portafusibles de bayoneta	*7€*	*1*	*7€*
Regleta de conexión	*4€*	*1*	*2.7€*
Cableado (variado)	*0.4 (m)€*	*4(m)*	*1.6€*
Manguito Termoretractil (variado)	*0.60 (m)€*	*2(m)*	*1.2€*
		Total	*12.5€*

Tabla 7.1 2: Gastos en materiales eléctricos

Gastos en elementos electrónicos			
Concepto	**Coste**	**Unidades**	**Coste Total**
Sabertooth 2x32	*120€*	*1*	*120€*
HEDS-5600	*40€*	*2*	*80€*
Arduino UNO	*20€*	*1*	*20€*
Arduino MEGA	*35€*	*1*	*35€*
Raspberry Pi3 model B	*37€*	*1*	*37€*
SDC8gb Tarjeta micro SD	*10€*	*1*	*10€*
Fusible 50 Amph	*10€*	*4*	*40€*
Adaptador de corriente RaspberryPi3	*8€*	*1*	*8€*
Sweep Laser Scanner “Scanse”	*300€*	*1*	*300€*
		Total	*650€*

Tabla 7.1 3: Gastos en elementos electrónicos

Una estimación del coste total destinado a los materiales a emplear durante el desarrollo del sistema quedaría determinado por la siguiente estructura(Figura 6.4)

Resumen de Gastos materiales	
Concepto	**Coste Total**
Gastos en elementos mecánicos	*2020.75€*
Gastos en electrónica básica	*12.5€*
Gastos en elementos electrónicos	*650€*
IVA	*563.48€*
Total	*3333.73€*

Tabla 7.1 4: Resumen de los gastos en materiales

7.1.2 Gastos en Herramientas e informática

Seguidamente se enumerara el coste asociado a las herramientas necesarias para la construcción del sistema.

Gastos en Herramientas de trabajo			
Concepto	**Coste (€/h)**	**Unidades**	**Coste Total**
Materiales de laboratorio (tester, fresadora,...)	*4€*	*20*	*80€*
Herramientas varias (desoldador, pelacables,...)	*1€*	*20*	*20€*
Dispositivos electrónicos (osciloscopio, fuente,...)	*1.78€*	*40*	*71€*
PC	*1.5€*	*550*	*825€*
		Total	*995€*

Tabla 7.1 5: Gastos en herramientas de trabajo

Gastos en software	
Concepto	**Coste Total**
Linux debian	*0€*
Arduino IDE	*0€*
Raspbian	*0€*
DEScribe	*0€*
ROS	*0€*
Total	*0€*

Tabla 7.1 6: Gastos en software

Una estimación del coste total destinado a las herramientas, los equipos y el software a emplear durante el desarrollo del sistema quedaría

determinado por la siguiente estructura(Figura 6.7)

Resumen de Gastos en herramientas	
Concepto	**Coste Total**
Gastos en herramientas de trabajo	*995€*
Gastos en software	*0€*
Total	*995€*

Tabla 7.1 7: Resumen de los Gastos en herramientas

7.1.3 Gastos en Mano de obra

Por último se enumerara el coste asociado a la mano de obra necesaria para el desarrollo y la construcción del sistema.

Para ello se realizó una evaluación de el sueldo medio en España de un ingeniero informático con un nivel de experiencia medio.

Concepto	**Coste Total**
Sueldo neto de un ingeniero	*26000€*
Seguridad social	*8000€*
Total	*34000€*
Sueldo/h	*20€/h*

Tabla 7.1 8: Gastos en Mano de obra

Teniendo en cuenta que el desarrollo del proyecto ha requerido de 6 meses para completarse, y estimando que al mes existen 150 horas laborales se deduce que el coste en mano de obra será de aproximadamente 3000€ al mes, 18000€ en total.

7.1.4 Resumen de Gastos directos

Resumen de Gastos directos	
Concepto	**Coste Total**
Materiales	*3333.73€*
Herramientas e informática	*995€*
Mano de obra	*18000€*
Total	*22330€*

Tabla 7.1 9: Resumen de los Gastos en herramientas

7.2 Gastos indirectos

Los gastos indirectos son aquellos que no se asocian con el producto de manera directa, sino que están relacionados con los recursos consumidos durante el desarrollo del producto final, tal y como son la electricidad o el agua.

Resumen de Gastos indirectos	
Concepto	**Coste Total**
Coste de la electricidad empleada	*150€*
Total	*150€*

Tabla 7.2 1: Resumen de los Gastos indirectos

7.3 Coste completo del proyecto

Concepto	Coste Total
Costes directos	*22330€*
Costes indirectos	*150€*
Total	*22480€*

Tabla 7.3 1: Resumen de los Gastos totales

Capítulo 8
Conclusiones y líneas futuras

Para finalizar, en este capítulo se expondrán las conclusiones a las que se llegaron durante el desarrollo del proyecto, al igual que se presentarán alternativas para mejorar la funcionalidad del mismo.

Se comenzará con unas conclusiones generales, seguidamente se continuará con las conclusiones especificas de los objetivos.

8.1 Conclusiones

Una de las conclusiones más claras que se obtuvo del proceso de diseño y de construcción, fue la enorme diversidad de metodologías y herramientas que se pueden emplear para realizar un sistema robótico similar a este, tanto en lo que respecta a actuadores y sensores, como al software y el hardware utilizado.

Gracias a esto, se adquirió una valiosa experiencia ganada durante las fases de diseño, montaje e instalación del proyecto. Esta experiencia fue complementada con los conocimientos adquiridos tanto durante el grado, como de la documentación previa al desarrollo y del desarrollo en si mismo. De esta manera se puede considerar que el poyecto ha aportado mucho al autor, quien ha visto potenciados sus conocimientos sobre electrónica, cinemática o robótica general, y ha adquirido otros nuevos como ROS.

En lineas generales, los objetivos y/o requisitos del proyecto han sido completado exitosamente.

El sistema es capaz de desplazarse de forma automática o manual, esto es posible gracias a la electrónica y los sensores empleados para gobernar los motores y obtener información del avance y dirección de estos, esta característica unida a la utilización del Sweep proveen de toda la información necesaria para elaborar el mapa, en tiempo real, del entorno que rodea al sistema.

Es por todo esto por lo que se considera que la problemática principal del proyecto ha sido resuelta y verificada.

8.2 Lineas futuras

En este apartado se expondrán las alternativas o mejoras para una posible modificación del proyecto con el fin de potenciar y añadir funcionalidades.

Tal y como se observa en los objetivos del proyecto, su finalidad es la de realizar un mapa del entorno circundante al sistema, una mejora simple sería la de añadir localización ademas de mapeo a las características de este haciendo uso de técnicas de localización y modelado simultáneos. Se trata de una modificación que unicamente requeriría de un sustituto para el hardware empleado en el *operador de planificación*, dado que la Raspberry Pi3 model B no dispone de la potencia necesaria para dar soporte a esta modificación.

Una alternativa a la utilización de Arduino en el sistema es diseñar un modelo de microcontrolador propio en base a los micrcontroladores utilizados, aprovechando sus características open source and hardware con el fin de disminuir los costes y aumentar la eficiencia.

Otra posible mejora sería la de diseñar y montar un modulo que combine giroscopio y acelerómetro, con el se podría obtener una comparativa para con los datos proveídos por sistema sensorial e incrementar la información odométrica emitida al *operador de planificación*.

Capítulo 9
Summary and Conclusions

This chapter presents the conclusions reached during the development of the project, as well as alternatives to improve its quality.

For starters, we will delve into some general thoughts about the project, and then continue with the specific conclusions.

9.1 Conclusions

One of the clearest conclusions that was obtained from the design and construction process was the enormous diversity of methodologies and tools that can be used to make a robotic system similar to this one, both in terms of actuators and sensors as well as software and hardware.

Valuable experience was acquired during the design, assembly and installation phases of the project. This experience was complemented with the knowledge obtained during the degree, to the documentation done prior to the development and also to the development itself. Therefore this project has contributed a lot to the author, who has seen his knowledge about electronics, kinematics or general robotics strengthened, and has acquired new ones such as ROS.

The system is able to move automatically and manually, this is possible thanks to the electronics and sensors used to govern the engines and to gather information regarding the progress and direction of these, this feature, coupled with the use of the Sweep provide all the information necessary to create a map of the environment that surrounds the system in real time.

Hence, in accordance to these results, the author concludes that the main hindrances of the project has been resolved and the proper functions have been verified.

9.2 Sumary

In this section we review some alternatives, improvements or possible modifications for the project in order to enhance and add functionalities.

As it is stated in the objectives section, the purpose is to make a map of the environment surrounding the system. A simple improvement would be to include location in addition with the mapping of the characteristics of the environment,therefore using simultaneous localization and modeling techniques. It is a modification that would only require a substitute for the hardware used in the *planning operator*, since the Raspberry Pi3 model B does not have the necessary power to support this modification.

An alternative to the use of Arduino in the system is to design its own model of its own micrcontroller based on the microcontrollers used, taking advantage of its open source and open hardware features in order to reduce costs and increase efficiency.

Another possible improvement would be to design a module that combines gyroscope and an accelerometer, with which a comparison could be obtained with the data provided by the sensory system and thus increase the odomometric information issued to the *planning operator*.

Capítulo 10
Códigos

10.1 Código del operador de control

10.1.1 Subsistema maestro

Código ejecutándose en el microcontrolador modelo Arduino MEGA

El código perteneciente al subsistema maestro es el encargado de múltiples tareas, desde recibir información por su sistema sensorial como de hacer uso de actuadores en relación a los comandos recibidos entre otros muchos.

Para empezar se puede es importante percatarse como se emplean interrupciones para recoger los datos emitidos por los encoders.

Estos datos son los que se usaran mas adelante tanto como realiamentación para el control PID de velocidad como para generar y publicar el paquete odométrico emitido mediante ROS

El calculo, tanto del algoritmo PID como de la odometría se realizan de forma periódica para asegurar la fiabilidad de los datos, tal y como se puede observar al final del código. Su emisión se realiza en una relación de 2 a 1 por el coste operacional requerido para la generación del paquete odométrico.

Para gobernar los motores se emplea comunicación serial con el controlador de motores, por lo que se simplifica la tarea de manejo de los motores.

La comunicación serial con el subsistema esclavo se realiza mediante un bus I_2C, es por esto que se emplean bytes para su emisión, pese a que al recibir los datos se requiere de una operación de conversión de tipos.

Las publicaciones y subcripciones a los topicos de ROS se realizan en este código tras la definición de variables.

```
/
*************************************************************************
*****
*
* ROS_motor.ino
*
*************************************************************************
******
*
* AUTORES
*   Antonio Sanjuán Prieto
*
* FECHA
*   01/08/2018
*
* DESCRIPCION
*   Código ejecutándose en el microcontrolador modelo Arduino MEGA.
*   Función:  encargado de llevar a cabo operaciones directamente
relacionadas con los sensores y actuadores de la silla, emite paquetes de
odometria al operador de planificación.

          #include <Wire.h>
          #include <SoftwareSerial.h>
          #include <Sabertooth.h>
          #include <ros.h>
          #include <ros/time.h>
          #include <geometry_msgs/Twist.h>
          #include <geometry_msgs/Quaternion.h>
          #include <std_msgs/String.h>
          #include <nav_msgs/Odometry.h>
```

```
#include "types.h"

//Pinnes
 //channel-encoder
#define EncPinLA 2
#define EncPinLB 3
#define EncPinRA 18
#define EncPinRB 19
 //Encoder
volatile bool _EncLASet;
volatile bool _EncLBSet;
volatile bool _EncLAPrev;
volatile bool _EncLBPrev;

volatile bool _EncRASet;
volatile bool _EncRBSet;
volatile bool _EncRAPrev;
volatile bool _EncRBPrev;

volatile long _EncTicksL;
volatile long _EncTicksR;
bool direc;

 //Motor

int Tx_pin = 6;            // Arduino Transmit Pin to Sabertooth S1
int Rx_pin = 5;            // Arduino Receive Pin to Sabertooth S27
float last_outputL= 0.0;
float last_outputR= 0.0;
//SABERTOOTH
SoftwareSerial SWSerial(Rx_pin, Tx_pin); // RX, TX on pin 2 (to S1).
Sabertooth ST(128, SWSerial); // Address 128, and use SWSerial as the
serial port.

//Vars
 //ROBOT CARACTERISTICAS
float wheel_r = 0.16; //m
float tpv = 1024.0;        //el encoder me da 1024 ticks por vuelta (ticks
por vuelta)ENCODER
float wheel_perimeter = 2.0 * 3.14 * wheel_r;
float tpm = tpv / wheel_perimeter;          //Ticks por metro
float wheeltrack = 0.51;

  //PID
```

```
float kp = 0.065;
float ki = 0.0005;  //0.15
float kd = 0.003;   //1
bool auto_control = false;  //default control is Manual

  //limits
float MIN_OUT = -127.0;
float MAX_OUT = 127.0;
float float_PWM = 0.6;

  //Timmers
unsigned long _loop_functionPID_time_ = 50;   // (50/1.000
=0.05segundos)
unsigned long _current_time_;              //tiempo actual
unsigned long _lastPID_time_= 0;                 //ultimo tiempo
unsigned long _loop_functionODO_time_ = 1000; // (1000/1000 =
1segundo)
unsigned long _lastODO_time_ = 0;

  //ROS
char baseFrame[] = "/base_link";
char odomFrame[] = "/odom";
  //Struct de odometria
OdometryData odometryData;
  //Struct de PID
WheelPid PidL;
WheelPid PidR;
  //Creamos el handle node
ros::NodeHandle nh;
  //Publisher para registro de logs
std_msgs::String str_msg;
  //Publisher para odometria
nav_msgs::Odometry odom_msg;
ros::Publisher odomPub("odometry_", &odom_msg);
  //Debug Publisher
std_msgs::String log_msg;
ros::Publisher chatter("log_ardu_", &log_msg);

//FUNCTIONS2
int EncParser(bool prev_,bool set_){
 if(prev_ == set_){
 return 1;
 }
 return -1;
```

```
}

void debounceCountL(){
 _EncLBSet = digitalRead(EncPinLB);
  _EncLASet = digitalRead(EncPinLA);

  _EncTicksL += EncParser(_EncLAPrev, _EncLBSet);

  _EncLAPrev = _EncLASet;
  _EncLBPrev = _EncLBSet;
}

void debounceCountR(){
  _EncRBSet = digitalRead(EncPinRB);
  _EncRASet = digitalRead(EncPinRA);

  _EncTicksR += EncParser(_EncRAPrev, _EncRBSet);

  _EncRAPrev = _EncRASet;
  _EncRBPrev = _EncRBSet;
}

float VelToTicks(float v){
    float aux = (v * tpv) / wheel_perimeter;
    return(_loop_functionPID_time_ * aux) / 1000.0;  //(los 100 es porun
segundo)
}

void selector_vel(float _x, float _th){
  float x = _x;
  float th = _th;
  float velo_right;
  float velo_left;
  if(x == 0 && th == 0){
    velo_right = 0.0;
    velo_left = 0.0;
  }
  else if(x == 0 && th != 0){
    velo_right = th;
    velo_left = (-1.0)*th;
  }
  else if(x != 0 && th == 0){
    velo_right = x;
    velo_left = x;
```

```
  }
  else if(x != 0 && th != 0){
    velo_right = x + th;
    velo_left = x + (-1.0)*th;
  }

  PidL.SpeedWish = velo_left;
  PidR.SpeedWish = velo_right;
  //Convertimos esta velocidad
  PidL.tpc = VelToTicks(PidL.SpeedWish);
  PidR.tpc = VelToTicks(PidR.SpeedWish);
}

void MsgToVel(const geometry_msgs::Twist& msg){
  float x = msg.linear.x;
  float th = msg.angular.z;
  float velo_right;
  float velo_left;
  if(x == 0 && th == 0){
        str_msg.data = "Geometry_msgs/Twist -- '[0.0, 0.0, 0.0]' '[0.0, 0.0,
0.0]' ";
    velo_right = 0.0;
    velo_left = 0.0;
  }
  else if(x == 0 && th != 0){
    //GIRO PURO
    str_msg.data = "Geometry_msgs/Twist -- '[0.0, 0.0, 0.0]'
'[??.??, ??.??, ??.??]' ";
        velo_right = th;
    velo_left = (-1.0)*th;
  }
  else if(x != 0 && th == 0){
    //AVANCE PURO
    str_msg.data = "Geometry_msgs/Twist -- '[??.??, ??.??, ??.??]' '[0.0,
0.0, 0.0]'  ";
    velo_right = x;
    velo_left = x;
  }
  else if(x != 0 && th != 0){
    //AVANCE MIXTO
    str_msg.data = "Geometry_msgs/Twist -- '[??.??, ??.??, ??.??]'
'[??.??, ??.??, ??.??]' ";
        velo_right = th;
    velo_left = (-1.0)*th;
```

```
  }

//imprimimos algo de debug
 chatter.publish(&str_msg);
 //Añadimos el parametro de velocidad deseada
 PidL.SpeedWish = velo_left;
 PidR.SpeedWish = velo_right;

 //Convertimos esta velocidad
 PidL.tpc = VelToTicks(PidL.SpeedWish);
 PidR.tpc = VelToTicks(PidR.SpeedWish);
}

//Subscriber al topic
ros::Subscriber<geometry_msgs::Twist> MsgToVelocidad("/cmd_vel",
&MsgToVel);

//funcion para el calculo del PID
void PID_Calc(WheelPid * p){
  float output;
  float Perror, Ierror, Derror;

  //Término proporcional
  float error = p->tpc - (p->Ticks - p->Prev_Ticks);
  Perror = error;
  //Término derivativo
  Derror = (Perror - p->PrevErr);
  //Término integral
  p->sumErr += error;
  Ierror = p->sumErr ;

  //MODULAMOS IERROR
  if(Ierror > 300.0){
    Ierror = 300.0;
  }else if(Ierror < -300.0){
    Ierror = -300.0;
  }

  output = Perror * kp + Ierror* ki;
  p->PrevErr = error;
  p->Prev_Ticks = p->Ticks;
  output += p->output;

  //MODULAMOS
```

```
  if(output < MIN_OUT){
    output = MIN_OUT;
  }else if(output > MAX_OUT){
    output = MAX_OUT;
  }

  //Brakes helper
  if(abs(p->tpc) < 10){
     PidRestart();
     p->output = 0;
  }else{
          p->output = output;
  }
}
//Funcion para refrescar parametros del PID
void RefreshPid(){
    drive_(&PidL, &PidR);

    //IMPORTANTE
    PidL.Ticks = _EncTicksL / 4;
    PidR.Ticks = _EncTicksR / 4;
    odometryData.Enc_Time = millis();
    odometryData.Last_EncRead = nh.now();
    PID_Calc(&PidL);
    PID_Calc(&PidR);
}
//Funcion de avance/retroceso de los motores
void drive_(WheelPid * a,WheelPid * b){
         ST.motor(1,b->output);
         ST.motor(2,a->output);
}
//Custom map function [From int to float]
float custom_map(int value, int _min, int _max, float _Tomin, float
_Tomax){
  float result = 0.0;
  result = (value - _min) * (_Tomax - _Tomin) / (_max - _min) + _Tomin;
  return result;
}
void request_function(){
  byte command_byte[2];              //lo que recibimos por I2C
  int  command_int[2];              //la traduccion que hicimos
  Wire.requestFrom(9,2);             //conectamos con el slave9 y le
pedimos 2 bytes
  command_byte[0] = Wire.read();
```

```
  command_byte[1] = Wire.read();
  for(int i=0;i<2;i++){
    if(command_byte[i] > 127){
        command_int[i] = 256 - (int)command_byte[i];
        command_int[i] *= -1;
    }else{
        command_int[i] = command_byte[i];
    }
  }
  selector_vel(custom_map((-1)*command_int[0],(-100.0),100.0,(-
3.0),3.0),custom_map((-1)*command_int[1],(-100.0),100.0,(-3.0),3.0));
}
//Refrescar datos de odometria
void RefreshOdometry(){
  //obtener ultima medida de los encoders
  double dt = (odometryData.Enc_Time -
odometryData.Prev_Enc_Time)/1000.0;
  //Refrescamos el valor
  odometryData.Prev_Enc_Time = odometryData.Enc_Time;

  //distancia recorrida por las ruedas
  double distanceLWheel = (PidL.Ticks - odometryData.Last_EncL) /
tpm;
  double distanceRWheel = (PidR.Ticks - odometryData.Last_EncR) /
tpm;
  //Guardamos los anteriores valores del PID
  odometryData.Last_EncL = PidL.Ticks;
  odometryData.Last_EncR = PidR.Ticks;

  double _DistAver = (distanceLWheel + distanceRWheel) / 2.0;
//media de la distancia lineal
  double _AngleRot = (distanceRWheel - distanceLWheel) / wheeltrack;

  double LinearVel = _DistAver / dt;
  double AngularVel = _AngleRot / dt;

  if(_DistAver != 0){
    double dx = cos(_AngleRot) * _DistAver;
    double dy = -sin(_AngleRot) * _DistAver;

    odometryData.x_linear += (cos(odometryData.z_angular)* dx -
sin(odometryData.z_angular) * dy);
    odometryData.y_linear += (sin(odometryData.z_angular)* dx
+cos(odometryData.z_angular) * dy);
```

```
  }
  if(_AngleRot != 0){                    //Si ha habido firo
    odometryData.z_angular += _AngleRot;
  }


    //expresamos el giro como quaternion
    geometry_msgs::Quaternion quaternion;
    quaternion.x = 0.0;
    quaternion.y = 0.0;

    quaternion.z = sin(odometryData.z_angular / 2.0);
    quaternion.w = cos(odometryData.z_angular / 2.0);

    //Generamos el msg
    odom_msg.header.frame_id = odomFrame;
    odom_msg.child_frame_id = baseFrame;
    odom_msg.header.stamp = odometryData.Last_EncRead;
    odom_msg.pose.pose.position.x = odometryData.x_linear;
    odom_msg.pose.pose.position.y = odometryData.y_linear;
    odom_msg.pose.pose.position.z = 0;
    odom_msg.pose.pose.orientation = quaternion;
    odom_msg.twist.twist.linear.x = LinearVel;
    odom_msg.twist.twist.linear.y = 0;
    odom_msg.twist.twist.linear.z = 0;
    odom_msg.twist.twist.angular.x = 0;
    odom_msg.twist.twist.angular.y = 0;
    odom_msg.twist.twist.angular.z = 0;

  //lo publicamos
    odomPub.publish(&odom_msg);
  }
void PidRestart(){
  PidL.PrevErr = 0;
  PidL.Ierror = 0;
  PidL.output = 0;
  PidL.sumErr = 0;
  PidR.PrevErr = 0;
  PidR.Ierror = 0;
  PidR.output = 0;
  PidR.sumErr = 0;
}

void FullRestart(){
```

```
PidRestart();
 _EncTicksL = 0;
 _EncTicksR = 0;
}

void setup(){
 //serial
 Serial.begin(9600);
 SWSerial.begin(9600);
 ST.autobaud();

 //I2C comunication
 Wire.begin();           //I2C as master

 //pinnes

 //resis pull-up
 pinMode(EncPinLA,LOW);
 pinMode(EncPinLB,LOW);
 pinMode(EncPinRA,LOW);
 pinMode(EncPinRB,LOW);
 //INTERRUPT
 //LEFT
 attachInterrupt(digitalPinToInterrupt(2),debounceCountL,CHANGE);
 attachInterrupt(digitalPinToInterrupt(3),debounceCountL,CHANGE);
 attachInterrupt(digitalPinToInterrupt(18),debounceCountR,CHANGE);
 attachInterrupt(digitalPinToInterrupt(19),debounceCountR,CHANGE);
 nh.initNode();

 nh.advertise(odomPub);       //odometry_
 nh.subscribe(MsgToVelocidad); //vel_cmd
 nh.advertise(chatter);       //log_ardu_
}
void loop(){
 _current_time_ = millis();
 if(_loop_functionPID_time_ + _lastPID_time_ < _current_time_){
 //if manual control is active
 if(auto_control == false){
 request_function();
   }
   RefreshPid();
   _lastPID_time_ = _current_time_;
 }
 if(_loop_functionODO_time_ + _lastODO_time_ < _current_time_){
```

```
    RefreshOdometry();
    _lastODO_time_ = _current_time_;
    Serial.print("-------------------------\n");
  }

  nh.spinOnce();
}
```

10.1.2 Subsistema esclavo

Código ejecutándose en el microcontrolador modelo Arduino UNO

El código perteneciente al subsistema esclavo tiene por objetivo el de recopilar toda la información capturada por el Joystick, y modularizar esta información para que sea posible su emisión y recepción a través del bus. Para esto se satura la entrada del Joystick con el fin de evitar lecturas erróneas, de este modo es posible capturar los datos obtenidos del mando, por ultimo se modulan estos datos para que el rango de valores posibles este dentro del intervalo permitido por la estructura de datos determinada para su emisión, un array de Bytes.

El microcontrolador mantiene en un bucle de captura de datos hasta que el microcontrolador maestro conectado a él mediante el bus le hace una petición, es entonces cuando el subsistema esclavo le responde con la ultima captura de datos modularizados realizada.

```
/
**************************************************************************
*****
*
* Joystic:_emisor.ino
*
**************************************************************************
******
*
* AUTORES
*   Antonio Sanjuán Prieto
*
* FECHA
*   01/08/2018
*
* DESCRIPCION
*   Código ejecutándose en el microcontrolador modelo Arduino UNO
*   Función:   encargado de permitir el control manual de la silla
```

```
//Controlador del Joystic

#include <Wire.h>
#include <SoftwareSerial.h>
/*
 * PINNES
 * 1- VCC
 * 2- LEFT/RIGHT
 * 3- GND
 * 4- FORWARD / BACK
 * 5- FORWARD/BACK
 * 6- VCC = 3.3
 * 7- LEFT/RIGHt
 * 8- UNUSED
 *
 * recordar coger un gnd comun entre ambos arduinos
 * el sda de un arduino maestro es el sda del arduino esclavo
 * el scl de un arduino maestro es el scl del arduino esclavo
 */

//REPARAMOS DESFASE (error en la medida potenciometros)
int desf_x = 1215;
```

```
int desf_y = 1215;

//DECLARACION PINNES
int pin_x = A1;      //ejex -> LEFT/RIGHT (A1)
int pin_y = A0;      //ejey -> FORWARD/BACK (A0)
int pin_x2 = A2;
int pin_y2 = A3;

//VALORES DEL JOYSTIC (locales)
int input_x;       //ejex -> LEFT/RIGHT
int input_y;       //ejey -> FORWARD/BACK
int input_x2;
int input_y2;


//VALORES DE SALIDA  (COMMAND)
int Preoutput_x;
int Preoutput_th;
int output_x;
int output_th;

//función que responde a la peticion del MASTER
void response_function(){

  byte buffer_[2] = {0,0};
  //saturamos
  if(abs(output_x) >= 10){
    //Serial.print("entramos");
    buffer_[0] = output_x;
  }
  if(abs(output_th) >= 10){
    buffer_[1] = output_th;
  }
  Wire.write(buffer_,2);
}

 void setup() {

  //configuracion de pinnes
  pinMode(pin_x, INPUT);
  digitalWrite(pin_x, HIGH);
  pinMode(pin_y, INPUT);
  digitalWrite(pin_y, HIGH);
  pinMode(pin_x2, INPUT);
```

```
 digitalWrite(pin_x2, HIGH);
 pinMode(pin_y2, INPUT);
 digitalWrite(pin_y2, HIGH);

 //serial a 9600baudios
 Serial.begin(9600);

 //comunicacion I2C
 Wire.begin(9);       //start the I2C bus as slave #9
 Wire.onRequest(response_function);    //a la espera de que se le pidan
los datos
}

void loop() {

 input_x = analogRead(pin_x);
 input_x2 = analogRead(pin_x2);
 input_y = analogRead(pin_y);
 input_y2 = analogRead(pin_y2);

 Preoutput_x = (input_y+input_y2 - desf_y);        //eje y (velocidad lineal)
 Preoutput_th = (input_x+input_x2 - desf_x);           //eje x (velocidad
rotacional)

 //mapeamos

 output_x = map(Preoutput_x,(-590),590,(-100),100);
 output_th = map(Preoutput_th,(-590),590,(-100),100);

 }
```

Bibliografía

[1] Arduino

- *https://www.arduino.cc/*

[2] Sabertooth Datasheet

- *https://www.dimensionengineering.com/datasheets/Sabertooth2x32.pdf*

[3] Sabertooth DEScribe

- *https://www.dimensionengineering.com/info/describe*

[4] Sabertooth DIP Switch Wizard

- *https://www.dimensionengineering.com/datasheets/USBSabertoothDIPWizard/*

[5] HEDS-5600 Datasheet

- *http://www.avagotech.com/docs/AV02-1046EN*

[6]Encoder de cuadratura

- http://www.ni.com/tutorial/7109/es/

[7] Raspberry Pi3 model B

- *https://cdn.sparkfun.com/datasheets/Dev/RaspberryPi/2020826.pdf*

[8] Raspbian SO

- *https://www.raspberrypi.org/downloads/raspbian/*

[9] Robot Operating System (ROS)

- *http://www.ros.org/*

[10] Wiki de ROS

- *http://wiki.ros.org/es*

[11] Wiki de Rosserial ROS

- *http://wiki.ros.org/rosserial*

[12] Sweep SDK

- *https://github.com/scanse/sweep-sdk*

[13] Sweep ROS

- *https://github.com/scanse/sweep-ros*

[14] Sweep visualizer

- *http://scanse.io/*