Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

DISEÑO DE UN DRIVER PARA ROS PARA EL CONTROL DE UN BRAZO ROBÓTICO TINKERKIT BRACCIO

Alumno: Pereda Bados, Joaquin

Director: Casquero Oyarzabal, Oskar

Curso: 2019-20

Fecha: Bilbao, 10 de febrero de 2020

Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

Resumen

A lo largo de este Trabajo de Fin de Grado, hemos desarrollado el control de un brazo robótico Tinkerkit Braccio, el cual se controla por medio de una placa Arduino MEGA 2560. Generalmente el brazo se controla desde el entorno del Arduino, pero en este trabajo vamos a controlarlo a través de ROS, para lo que vamos a diseñar un driver para Arduino.

Palabras clave: robot, brazo robótico, Arduino, ROS.

Laburpena

Gradu Amaierako Lan honetan, Tinkerkit Braccio beso robotikoakontrolatukodugu. Beso hau Arduino MEGA 2560 plaka baten bidezkontrolatzen da. Normalean, Arduino programazio ingurunearen bidez garatzen da, baina proiektu honetan ROS sistema eragilearen bidez kontrolatuko dugu. Horretarako Arduinorentzako driver bat diseinatuz.

Gako-hitzak: robot, beso robotiko, Arduino, ROS.

Abstract

Thorough this Final Degree Project, we developed the software to control the Tinkerkit Braccio robotic arm. This arm is controlled by an Arduino MEGA2560 board. It is usually controlled using the Arduino IDE, but we are going to design a driver to control it using ROS.

Keywords: robot, robot arm, Arduino, ROS.

Índice

Índice de figuras

Índice de tablas

1.- Introducción

En un comienzo, la adopción de la automatización de la industria por parte de las empresas se centró en conseguir sistemas aislados de forma que se lograsen automatizar diferentes fases de un proceso. De ahí se pasó, al principio de la década de los 80, a tratar de que esos sistemas se integrasen formando un conjunto, puesto que el ambiente de competencia actual requiere que la fabricación deje de considerarse una actividad aislada y que incorpore todos los elementos necesarios para que la empresa funcione como una entidad única. Entre las principales razones para adoptar la automatización por parte de las empresas podemos destacar las ventajas que representa para los sectores con un tipo de producción por lotes, así como las diferentes tecnologías, técnicas y metodologías que mejoran la flexibilidad y productividad de los procesos de producción, logrando de esa forma una calidad óptima y homogénea en los productos fabricados además de reducir los costes y mejorar la eficiencia de la producción [1].

Hoy en día, los robots son una pieza fundamental en la automatización de la industria. A pesar de las dificultades para establecer una definición formal de robot industrial, Debido en gran medida a la diferencia conceptual entre el mercado euro-americano y el mercado japonés podríamos definir a un robot industrial, según la Organización Internacional de Estándares (ISO), como un “manipulador multifuncional reprogramable con varios grados de libertad, capaz de manipular materias, piezas, herramientas o dispositivos especiales según trayectorias variables programadas para realizar tareas diversas”. Entre los tipos de robots industriales se encuentran los robots con control por computador. Éstos se basan en la programación textual, para la cual el programador utiliza el terminal del computador, el cual cuenta con un lenguaje específico compuesto por varias instrucciones con las cuales se controla el robot, sin necesidad de interactuar con el brazo. De esta forma, se hace posible la programación de diferentes robots desde un mismo terminal, así como se facilita la opción de replicar las acciones de un robot [2].

2.- Contexto

Hay varios tipos de arquitectura de robots, entre los cuales destacan los siguientes:

- Robots de configuración cartesiana: Disponen de tres movimientos prismáticos (P , P , P) y su espacio de trabajo sería un tetraedro.

- Robots de configuración cilíndrica: Su base se mueve de forma rotacional, mientras que en los otros ejes se mueven de forma prismática (R , P , P). De esa forma, su área de trabajo tiene forma cilíndrica.

- Robots de configuración esférica: Las dos primeras articulaciones son de tipo rotacional, mientras que la tercera es de tipo prismático (R, R, P). Esto propicia un espacio de trabajo con forma esférica.

- Robots de configuración rotacional o articulada: Todas las articulaciones son de tipo rotacional, lo cual permite ordenar movimientos a cada articulación para llevar a cabo diferentes tipos de movimientos, proporcionando un amplio espacio de trabajo a costa de una mayor complejidad en el control.

El robot *Tinkerkit Braccio*, el cual puso en venta Arduino, se puede catalogar dentro del último grupo. Este brazo puede ser montado de diversas formas, de modo que se adapte a los gustos del usuario. *Braccio* permite también el manejo de diferentes objetos, desde unas pinzas hasta una cámara o una célula fotovoltaica [3].

El brazo será controlado por una placa Arduino, la cual se conectará a los servomotores del brazo por medio del *shield* que viene con el propio brazo [4].

Este trabajo de fin de grado nace de la búsqueda de una alternativa programática para el control del brazo robótico. La finalidad es utilizar el sistema operativo ROS (Robot Operating System), que es un sistema operativo bajo la licencia open source que provee a los desarrolladores de software aplicaciones para robots herramientas y diferentes librerías para facilitar la abstracción del hardware, el control de los dispositivos y la administración de paquetes entre otras cosas [5], para permitir el control del brazo robótico mediante su sistema de nodos y tópicos.

Con esto, se trata de desarrollar una forma más sencilla de controlar el brazo para así poder utilizarlo en entornos menos especializados (como podría ser el ámbito académico).

3.- Objetivos del trabajo

La principal finalidad de este trabajo de fin de grado es diseñar y desarrollar un *driver* para Arduino que permita controlar el *Braccio* de una forma sencilla. Para ello, habrá que cumplir los siguientes objetivos secundarios:

- Conocimiento del sistema operativo ROS, así como los comandos básicos para la creación de paquetes, y creación y edición de los diferentes tipos de nodos.

- Conocimiento del funcionamiento y uso de las diferentes librerías de las que dispone ROS.

- Conocimiento del lenguaje de programación Python para crear los nodos funcionales de ROS.

- Conocimiento de programación en la plataforma Arduino.

4.- Beneficios aportados por el trabajo

Para hablar de los beneficios que aporta este trabajo, vamos a catalogarlos en tres grupos, los cuales son beneficios sociales, beneficios técnicos y beneficios económicos:

- Como beneficios sociales del trabajo, podemos destacar que creando un driver de Arduino que permita el control del brazo robótico mediante ROS, se podría fomentar de cierto modo el uso de ambos en ámbitos académicos y de ocio, creando un acercamiento de estos a un grupo más amplio de personas.

- Respecto a los beneficios técnicos, el hecho de controlar el brazo mediante ROS abre un amplio abanico de posibilidades, a la vez que aporta numerosas ventajas, como la facilidad para comunicarse con otros dispositivos que utilicen la misma plataforma.

- Por otra parte, no se aprecia ningún beneficio económico en este proyecto.

5.- Requisitos

5.1.- Requisitos del sistema

Para llevar a cabo este trabajo, hay varios requisitos que hay que cumplir:

- Se utilizará *Tinkerkit Braccio*, un brazo robótico de Arduino, el cual dispone de seis servomotores que permiten el movimiento rotacional de las seis articulaciones del robot.

- Para controlar el *Braccio*, se utilizará la placa Arduino MEGA 2560.

- Para llevar a cabo los movimientos se hará uso del sistema operativo ROS. Éste está diseñado principalmente para su uso en Linux pero mayormente para Ubuntu. Por ello, se utilizará la versión 16.04 LTS de Ubuntu puesto que es una versión estable y muy utilizada del sistema operativo.

5.2.- Requisitos de funcionalidad

Los requisitos funcionales son las especificaciones de funcionamiento del robot. En este caso, se requiere de la capacidad para realizar dos tipos de movimientos. Por un lado está el movimiento simple, el cual consiste en indicar una posición a la que el robot debe moverse desde su posición actual, mientras que por el otro está el movimiento de trayectoria. Este último tipo es bastante similar al anterior, pero en vez de una posición se le indican una serie de puntos y el robot debe seguir la trayectoria formada por éstos, partiendo desde su posición actual y terminando en el último punto indicado. Para ello, se han implementado las siguientes funciones:

- Un tópico mediante el cual se lleven a cabo movimientos simples (movimiento a una posición).

- Un tópico que permita el movimiento de trayectoria.

6.- Antecedentes

En este apartado, vamos a hablar de los antecedentes de este trabajo. En un proyecto anterior cuyo título es "*Diseño y desarrollo de un paquete ROS para el control del brazo manipulador Tinkerkit Braccio*" [6] en el que se abordó el mismo problema, se optó por enviar la información necesaria para el movimiento del brazo mediante el puerto serie desde el nodo funcional en ROS. Para ello, se desarrolló un protocolo para gestionar el envío de información y gracias a éste, se interpretan los datos recibidos en el puerto serie en la placa Arduino que controla el brazo.

En nuestro caso, se ha "decidido" hacer uso de la librería *rosserial*, la cual permite la integración de la placa Arduino en la red de nodos y tópicos de ROS. De esta forma, no es necesario enviar los datos para el control del brazo a través del puerto serie y por ende tampoco un protocolo, si no que se publicarán en un tópico desde el que la placa Arduino leerá los mensajes.

7.- Análisis de alternativas

A la hora de llevar a cabo el proyecto, se han valorado diversas alternativas en algunos aspectos y se han tomado varias decisiones. Para ello, se han tenido en cuenta varios factores (independientes para cada una de las decisiones), los cuales se han valorado del 1 (peor) al 4 (mejor), teniendo cada uno de esos factores una ponderación acorde a su importancia para el proyecto.

7.1.- Lenguaje de programación

Debido a que para este proyecto se va a utilizar el sistema operativo de ROS, es necesario elegir un lenguaje compatible. Se han valorado los lenguajes Python y C++, puesto que son los dos lenguajes para los que hay una mayor documentación en la wiki de ROS [5].

En este caso, los factores o criterios a valorar serán: la facilidad de aprendizaje, el tiempo de desarrollo, la robustez del código y la complejidad de la sintaxis.

- Tiempo de aprendizaje: Debido a las limitaciones de tiempo para llevar a cabo el trabajo, este apartado es muy importante, siendo preferible un tiempo de aprendizaje corto que uno largo.

- Tiempo de desarrollo: Como con el criterio anterior, se valora un tiempo de desarrollo corto, el cual a su vez permita un mayor tiempo para el análisis y solucionar los posibles problemas que pudieran aparecer.

- Robustez del código: Una mayor robustez del código permite una mayor estabilidad del programa, de forma que el código tenga más capacidad de hacer frente a un error mientras se está ejecutando.

- Complejidad de la sintaxis: La facilidad de lectura y comprensión del código por parte de otras personas es también un punto a tener en cuenta, pero no tan importante como los anteriores.

Analizando cada uno de los lenguajes según los criterios arriba especificados:

- C++ : Se trata de un lenguaje muy robusto y estable, razón por la cual es uno de los más utilizados. Por otra parte, su complejidad lo convierten en un lenguaje para nada fácil de aprender. Ello conlleva un tiempo de aprendizaje largo. Además, el tiempo de desarrollo es relativamente largo también, y su sintaxis compleja.

- Python: Python podría considerarse uno de los lenguajes más sencillos y fáciles de aprender. Por ende, tiene un tiempo de desarrollo y aprendizaje cortos, así como una sintaxis muy sencilla, a costa de perder robustez de código.

Tabla 1: Análisis de las características de las alternativas del lenguaje de programación

Criterio	**Ponderación**	**C++**	**Python**
Tiempo de aprendizaje	4	1	4
Tiempo de desarrollo	3	2	3
Robustez	2	4	1
Complejidad de la sintaxis	1	2	4
Total		20	31

Observando la tabla, podemos concluir que la mejor opción para llevar a cabo el proyecto sería utilizar el lenguaje Python.

7.2.- Distribución de ROS

A pesar de que utilizar ROS se encuentra entre los requisitos indispensables del proyecto, hay diferentes distribuciones entre las que elegir (en este caso vamos a valorar ROS Kinetic, ROS Lunar y ROS Melodic). Para ello, hay que tener en cuenta el nivel de soporte, el tamaño de la comunidad y la compatibilidad con la versión 16.04 de Ubuntu:

- Soporte: El soporte es algo fundamental, ya que a mayor nivel de soporte, más actualizaciones habrá y más fácil será la solución de algunos problemas.

- Comunidad: Una comunidad más amplia implica una mayor probabilidad de obtener ayuda en caso de tener algún problema durante el desarrollo del proyecto.

- Compatibilidad con Ubuntu 16.04: La versión 16.04 de Ubuntu se encuentra entre los requisitos del sistema, por lo que también es algo indispensable para este proyecto.

Analizando cada una de las distribuciones mencionadas según los criterios arriba especificados:

- ROS Kinetic: ROS Kinetic Kame (también llamada ROS Kinetic) es una versión LTS (Long Term Support) lanzada en 2017. Por ello, dispondrá de soporte hasta el año 2021. Su amplio periodo de soporte la convierte en la versión con la comunidad más amplia y es compatible con Ubuntu 16.04.

- ROS Lunar: ROS Lunar Loggerhead (o ROS Lunar) es otra versión lanzada en 2017. Ésta, al no ser LTS, tendrá soporte solo hasta 2019. Ello implica que la comunidad es limitada. También es compatible con la versión 16.04 de Ubuntu.

- ROS Melodic: ROS Melodic Morenia (o ROS Melodic) es la última distribución de ROS. Por ello tiene soporte hasta 2023, y una comunidad en crecimiento pero no es compatible con Ubuntu 16.04

Tabla 2: Análisis de alternativas de distribución de ROS

Criterio	**Ponderación**	**ROS Kinetic**	**ROS Lunar**	**ROS Melodic**
Soporte	3	3	2	4
Comunidad	3	4	2	3
Compatibilidad Ubuntu 16.04	5	1	1	0
Total		26	17	21

Viendo los valores de la tabla, se utilizará la distribución ROS Kinetic.

7.3.- Tipos de mensajes utilizados

Para trabajar con ROS hay que utilizar mensajes. Éstos pueden ser o bien algunos de los mensajes estándar de ROS o bien un tipo de mensaje propio. En este apartado se tendrán en cuenta la documentación disponible y la adaptabilidad del mensaje a las necesidades del proyecto:

- Documentación: Ya que los mensajes van a ser utilizados por los usuarios del programa, es muy importante que haya suficiente documentación.

- Adaptabilidad: Aquí se valorará la idoneidad del tipo de mensaje para utilizar el programa.

Para este apartado, se tendrán en cuenta por un lado los mensajes proporcionados por ROS y por el otro un mensaje creado exclusivamente para el proyecto.

- Mensajes proporcionados por ROS: Al ser mensajes que son parte de las librerías básicas de ROS, disponen de una buena documentación. Respecto a la adaptabilidad, algunos de los campos quedarían vacíos, lo cual podría llevar a alguna confusión. Entre éstos, deberíamos destacar los mensajes *trajectory_msgs/JointTrajectory* [7] y *trajectory_msgs/JointTrajectoryPoint* [8], puesto que son los más utilizados para este tipo de situaciones.

- Mensaje propio: Comenzando por la adaptabilidad, crear un mensaje específico para el proyecto permite un manejo más fácil debido a que no hay ningún campo sobrante. Por el lado de la documentación, al ser un mensaje propio, no cuenta con ninguna documentación en la wiki. Además, solucionar los posibles problemas relacionados con el tipo del mensaje es más difícil.

Tabla 3: Características de los tipos de mensajes

Criterios	**Ponderación**	**Mensajes de ROS**	**Mensaje Propio**
Documentación y compatibilidad	4	3	1
Adaptabilidad	3	2	4
Total		18	16

A pesar de que las dos opciones podrían ser perfectamente viables, se ha decidido utilizar los mensajes proporcionados por ROS, puesto que se ha creído más importante facilitar la comunicación con otros robots que la adaptabilidad del tipo de mensaje.

8.- Análisis de la solución propuesta

Una vez descritos los requisitos y elegidos tanto el lenguaje de programación como la distribución y los mensajes de ROS a utilizar, se puede proceder a analizar la solución propuesta para cumplir los objetivos.

8.1.- Apartado de hardware

Primero procederemos a explicar el funcionamiento y a analizar las diferentes partes del hardware.

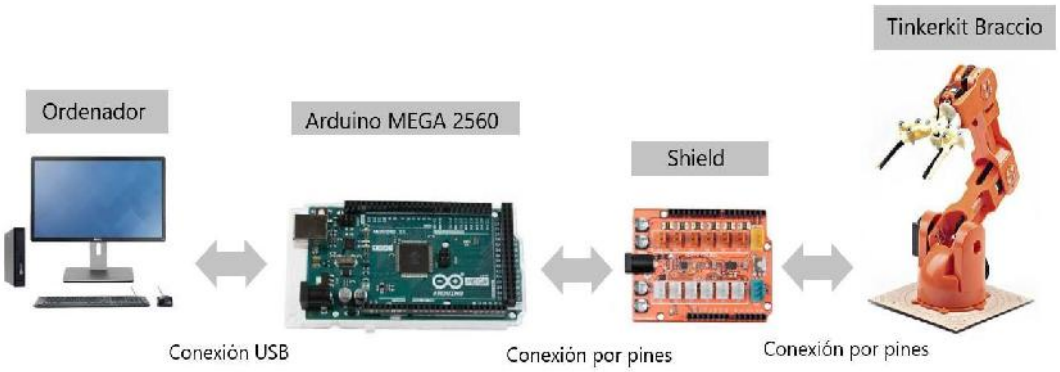

Figura 1: Estructura de comunicación del hardware del entorno de ejecución

En la figura se pueden ver por un lado el brazo robótico *Tinkerkit Braccio* con su *shield*, por otro la placa *Arduino MEGA 2560* y por último un ordenador. El brazo robótico se conectará a la placa Arduino mediante el *shield* que viene con el propio brazo. Este *shield* es una pieza indispensable, puesto que además de que el Arduino no tiene potencia suficiente para mover los seis servomotores del brazo, cada uno de los servomotores requiere de una conexión de tres pines (alimentación, tierra y control), mientras que la placa solo dispone de una salida de alimentación y una de tierra. La placa Arduino, por otro lado, se conectará al ordenador mediante un cable USB, utilizando el puerto serie.

Debido a que no es necesario para el diseño y la programación y a la gran dificultad de transporte para trabajar con él, el brazo robótico no se añadirá al proyecto hasta el final del mismo. Por este motivo y porque el único puerto serie de la placa Arduino estará ocupada por la librería rosserial, se va a emular un segundo puerto serie haciendo uso de un adaptador TTL-USB que se conecta tanto a la placa Arduino como al ordenador.

Figura 2: Estructura de comunicación del hardware del entorno de desarrollo

8.2.- Apartado de software

En este apartado analizaremos el software, el cual se compone de tres bloques:

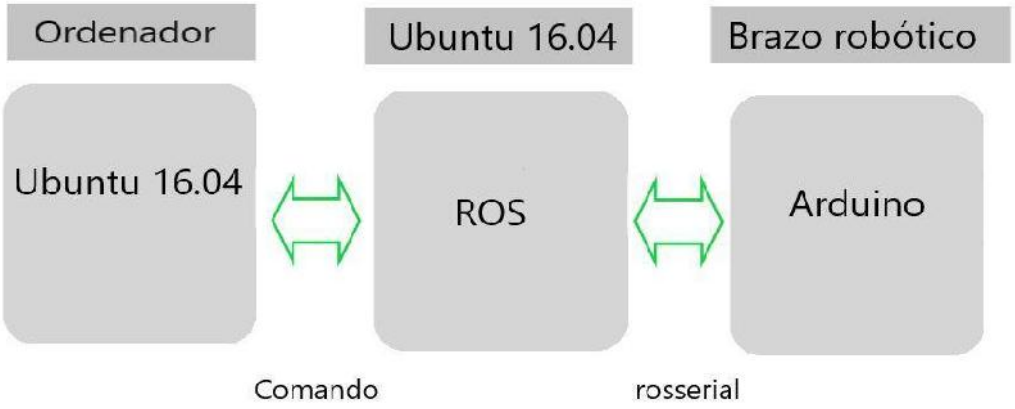

Figura 3: Estructura del software

Como se puede apreciar en la figura, por un lado se encuentra Ubuntu (que representa al usuario) y por el otro Arduino (que se encarga del control del robot), siendo ROS el nexo entre ambos.

Figura 4: Funcionamiento del software

En la parte de ROS se encuentran los nodos subscriptores y publicadores que se encargarán de llevar a cabo las comprobaciones pertinentes para ver si el mensaje enviado es válido, y en caso afirmativo publicarlo en otro tópico desde el que lo leerá el programa en Arduino.

En el Arduino, se leerán los mensajes de los tópicos de la segunda capa y serán procesados para con ellos darle al brazo la orden de movimiento correspondiente.

9.- Diseño

En este apartado se procederá a describir tanto el hardware como el software que componen el proyecto.

9.1.- Hardware

Como ya se ha mencionado anteriormente, el hardware es una faceta importante de este proyecto. Por ello, vamos a comenzar a detallar los diferentes componentes que forman la parte del hardware. Estos componentes son el brazo robótico *Tinkerkit Braccio*, la placa *Arduino MEGA 2560* y el adaptador *TTL-USB*.

El brazo robótico *Tinkerkit Braccio*, el cual consta de seis servomotores, se vende desmontado para que el usuario lo monte acorde a sus necesidades. En este caso ha sido montado en su forma estándar, permitiendo así el movimiento de todos los servomotores simultáneamente [3].

Figura 5: Brazo robótico *TinkerkitBraccio*

Las especificaciones del brazo son las siguientes:

Tabla 4: Especificaciones del brazo robótico Tinkerkit Braccio

Peso	792 g
Rango máximo de distancia de operación	80cm
Altura máxima	52cm
Anchura de la base	14cm
Anchura de la pinza	9cm
Longitud del cable	40cm
Capacidad de carga	Con distancia de operación de 32cm: 150gr
	Con la configuración mínima de *Braccio*: 400g

Al ser un brazo, la forma más fácil de nombrar los servomotores es asociándolos a su correspondiente articulación en el cuerpo humano.

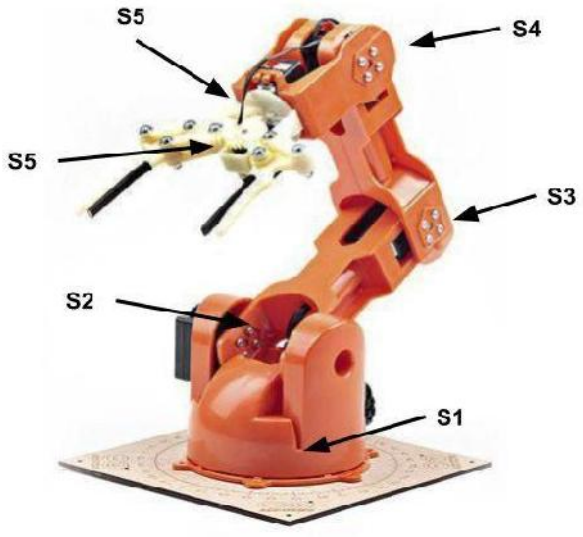

Figura 6: Servomotores del *Tinkerkit Braccio*

El movimiento de los diferentes servomotores es el siguiente:

Tabla 5: Nombres y movimiento de las articulaciones del robot

Nombre en la imagen	**Nombre del servomotor**	**Movimiento (en grados)**
S1	*Base*	0-180º
S2	*Shoulder* (Hombro)	15 - 165º
S3	*Elbow* (Codo)	0 - 180º
S4	*Wrist_ver* (Muñeca vertical)	0 - 180º
S5	*Wrist_rot* (Rotación de muñeca)	0 - 180º
S6	*Gripper* (Pinza)	10 - 73º (abierta-cerrada)

Además del *Braccio*, disponemos de una placa *Arduino MEGA 2560*. La placa se encargará de enviar las diferentes señales de control a los servomotores del brazo, permitiendo de esa forma que estos se muevan. Las características de la placa se muestran en la siguiente tabla [9]:

Tabla 6: Especificaciones de la placa *Arduino MEGA2560*

Microcontrolador	ATmega2560
Tensión de trabajo	5V
Tensión de entrada recomendada	7 - 12V
Límite de tensión de entrada	6-20V
Pines digitales I/O	54 (15 de los cuales proveen de salida PWM)
Pines de entrada analógica	16
Corriente DC por pin I/0	20mA
Corriente DC por pin 3.3V	50mA
Memoria Flash	258KB (de los cuales 8KB son utilizados por el *bootloader*)
SRAM	8KB
EEPROM	4KB
Velocidad del reloj	16MHz
LED_BUILTIN	13
Longitud	101.52mm
Anchura	53.3mm
Peso	37g

Para terminar el apartado hardware, es necesario mencionar el adaptador *TTL-USB*. Puesto que la placa Arduino dispone de un solo puerto serie y este será utilizado por rosserial para conectar la placa a la infraestructura de ROS, es necesario emular un segundo puerto serie en la placa, para lo que nos ayudaremos de este adaptador.

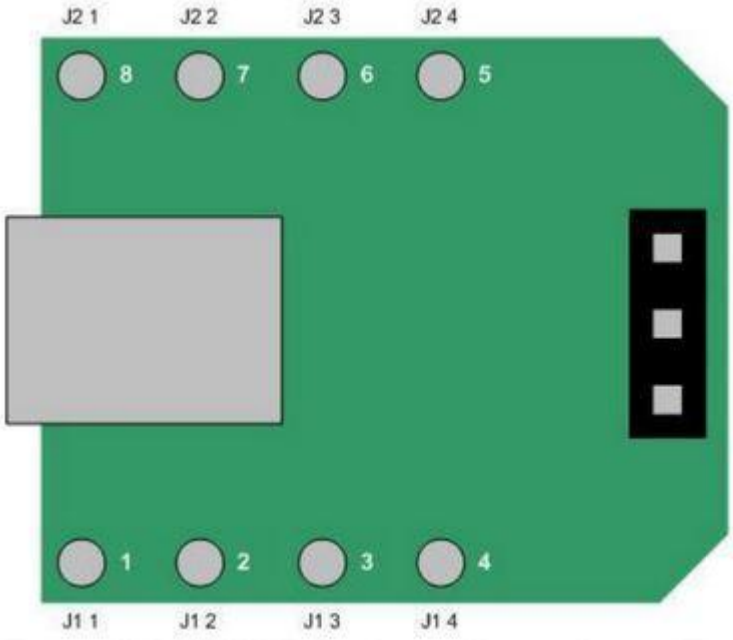

Figura 7: Pines del adaptador *TTL-USB*

El puerto USB del adaptador se utilizará para conectarlo al ordenador, mientras que los pines 7 y 8 (receptor y transmisor) del adaptador, se conectarán a los pines correspondientes de la placa Arduino (14 y 15 respectivamente). De esta forma se creará una conexión serie entre la placa y el adaptador, emulando un segundo puerto serie en la placa.

Figura 8: Estructura de comunicación del hardware del entorno de desarrollo

9.2.- Software

El apartado de software es el más importante del diseño. Por consiguiente, es indispensable conocer el funcionamiento de los diferentes apartados que lo componen, así como de la función que desempeñarán en el proyecto. Los componentes son, por una parte ROS, por otro la librería rosserial, y por último el software de Arduino. El código del proyecto está accesible en:

https://github.com/JoaquinPereda/TinkerkitBraccio-ROS-Driver

9.2.1.- Sistema Operativo ROS

9.2.1.1.- Función que desempeñará

El sistema operativo ROS es el que se encarga de la infraestructura, permitiendo enviar los diferentes mensajes. Además de eso, mediante su sistema de nodos y tópicos se reciben los mensajes y se comprueba que sean correctos, antes de transmitirlos al Arduino.

9.2.1.2.- Marco teórico

ROS (Robot Operating System) es un sistema operativo que se ejecuta sobre Ubuntu. Éste dispone de un gran número de librerías, así como de tipos de mensajes y aporta gran facilidad a la hora de diseñar software orientado al control de robots. Para comprender mejor su funcionamiento, es necesario explicar los tres puntos más importantes, que son los tipos de archivos con los que trabaja, los comandos básicos para trabajar con ROS y por último la comunicación.

- Tipos de archivos:

  Hay principalmente dos tipos de archivos en ROS:

- Por un lado tenemos los paquetes, los cuales son la unidad básica de organización. En éstos se recoge el código de los diferentes nodos y servicios que componen el paquete, así como de los mensajes que se hayan creado exclusivamente para él.

- Los mensajes, por el otro lado, son los archivos que se envían mediante ROS. Como bien hemos mencionado anteriormente, en cada paquete habrá varios archivos de mensajes en los que se especificarán los campos que componen el mensaje, así como el tipo de dato de cada uno de ellos. Es importante conocer la estructura de cada mensaje, puesto que cada nodo podrá trabajar con uno.

- Comandos básicos:

  - Una vez instalado ROS y antes de hacer cualquier otra cosa, habrá que ejecutar el comando *source* en la terminal de Ubuntu, de forma que ROS pueda acceder a los diferentes paquetes que lo componen. Además de ello, para poder utilizar cualquiera de los paquetes que haya en el espacio de trabajo habrá que utilizar el comando *source* con el archivo *devel/setup.bash* que se encuentre dentro de dicho espacio de trabajo.

```
source /opt/ros/kinetic/setup.bash

source catkin_ws/devel/setup.bash
```

Figura 9: Comando *source* de ROS

-

  - Para crear un paquete, habrá que ejecutar el comando *catkin_create_pkg*. A este comando hay que pasarle como parámetros el nombre del paquete y las dependencias para éste. En nuestro caso, las dependencias serán rospy y std_msgs.

```
catkin_create_pkg [nombre del paquete] [dependencias]

catkin_create_pkg paquetePrueba rospy std_msgs
```

Figura 10: Comando para crear paquetes en ROS

- Después de crear el paquete, usando el comando *catkin_make* se pueden compilar todos los paquetes que haya dentro del espacio de trabajo en el que se ejecute. Después de editar o crear algún nodo dentro de un paquete, tendremos que utilizar el comando *catkin_make* para compilarlo, y así poder utilizar y probar los cambios realizados.

- Para ejecutar un nodo, tenemos que utilizar el comando *rosrun*. Los parámetros necesarios por este comando son el nombre del paquete y el archivo del nodo que se quiere ejecutar.

Cabe destacar que para ejecutar el comando *rosrun* es necesario haber ejecutado en una terminal el comando *roscore*, el cual pone en marcha toda la infraestructura de ROS (master).

```
rosrun [nombre del paquete] [nombre del archivo]

rosrun paquetePrueba nodoPrueba.py
```

Figura 11: Comando para lanzar un nodo

- También disponemos del comando *rosnode*, el cual aporta varias utilidades para trabajar con nodos [10].

  - '*rosnode info*': Esta opción nos permite ver la información de un nodo concreto.

  - '*rosnode kill*': Sirve para terminar un nodo activo.

  - '*rosnode list*': Este comando muestra todos los nodos que están activos.

- '*rosnode cleanup*': Con esta función podemos purgar la información de registro de los nodos inalcanzables.

- Por último, vamos a mencionar el comando *rostopic.* Este comando nos aporta diferentes funcionalidades para trabajar con tópicos. Entre ellas, las más importantes son las siguientes:

  - '*rostopic echo*': Esta opción muestra por consola los mensajes que lleguen al tópico que hayamos elegido.

  - '*rostopic info*': Con esta opción, como con '*rosnode info*', se puede ver la información de un tópico.

  - '*rostopic list*': Sirve para listar los tópicos activos.

  - '*rostopic type*': Esta opción nos muestra el tipo de mensajes con el que el tópico elegido trabaja.

  - '*rostopic pub*': Es la función para publicar mensajes en un tópico.

```
jonkin@jonkin-VirtualBox:~$ rostopic list
/debug
/info
/rosout
/rosout_agg
/simpleMovement
/simpleMovementArduino
jonkin@jonkin-VirtualBox:~$ rostopic type /simpleMovement
trajectory_msgs/JointTrajectoryPoint
```

Figura 12: Funcionamiento del comando rostopic list

- Comunicación:

  La comunicación se lleva a cabo gracias a tres elementos:

- Por un lado tenemos el elemento maestro (también conocido como ROS Master [11]). Este elemento es la pieza fundamental del sistema, puesto que se encarga de permitir la comunicación entre los nodos. Su función principal es la de permitir a los nodos publicar o suscribirse a los tópicos que estos soliciten, de forma que se establezcan las comunicaciones entre los nodos.

- Por otro lado se encuentran los nodos. Éstos son programas capaces de llevar a cabo diferentes acciones. Entre ellos hay dos tipos bastante diferenciados.

Primero, tenemos a los subscriptores. Los nodos subscriptores se encargan de recibir los mensajes de los tópicos a los que están subscritos, para lo que utilizan una programación basada en eventos, de forma que esperan a que el tópico reciba un mensaje para después leerlo y trabajar con ello.

Además de subscriptores, los nodos también pueden ser publicadores. Un nodo publicador funciona de forma secuencial y su función dentro del sistema de comunicación de ROS es la de publicar mensajes en tópicos, para que los nodos subscriptores puedan leerlos.

- Por último, se encuentran los tópicos. Los tópicos son un canal que comunica a los diferentes nodos publicadores (puesto que puede haber más de un nodo publicando en el mismo tópico) con los diferentes nodos subscriptores.

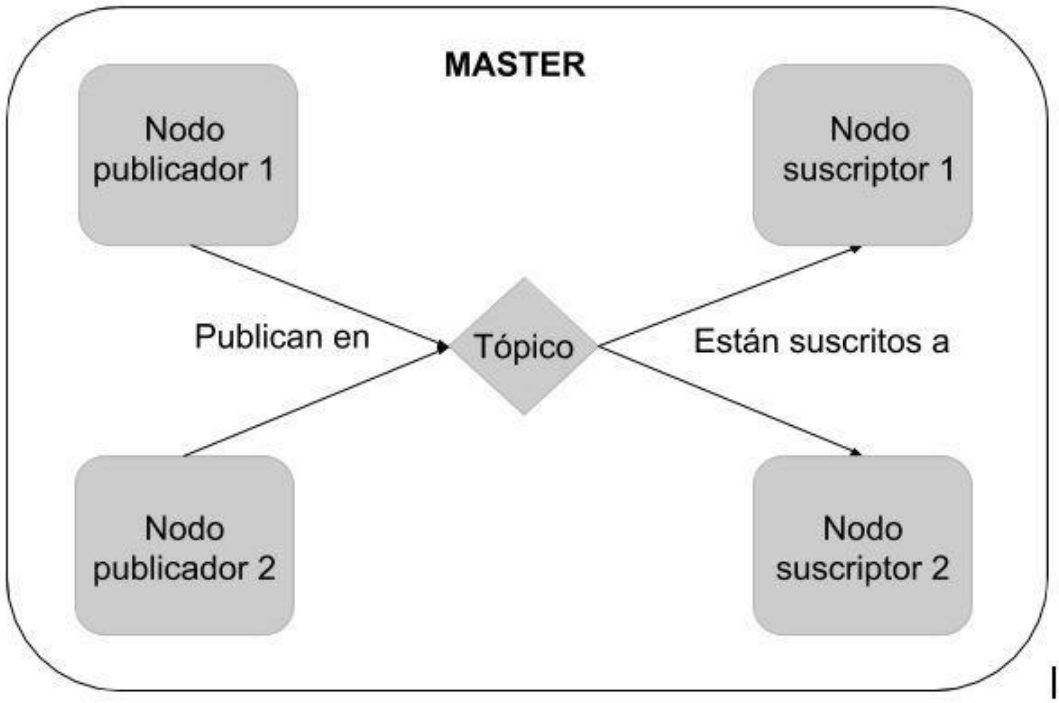

Figura 13: Comunicación de ROS

- No se puede terminar este apartado sin mencionar los mensajes, puesto que también son una pieza fundamental en la comunicación. Los mensajes son los datos que se transmiten entre los diferentes nodos. Cabe destacar que en cada tópico solo se puede manejar un tipo de mensaje, por lo que para enviar varios tipos de mensajes sería necesario crear varios nodos publicadores, con sus respectivos tópicos y los correspondientes nodos subscriptores que lean los mensajes de cada uno de esos tópicos.

9.2.1.3.- Aspectos prácticos

Una vez mencionados tanto la función que desempeñará dentro del proyecto como los aspectos teóricos más importantes del sistema operativo ROS, pasaremos a mostrar la forma en que se utilizará.

Antes de comenzar a explicar el diseño de los diferentes nodos, hay que comprender el tipo de mensaje que se va a utilizar. Al disponer de dos tipos de movimiento (de trayectoria y simple), utilizamos dos tipos de mensajes diferentes.

- Para el primero, se utilizan mensajes de tipo JointTrajectory. Este es un tipo de mensaje complejo, el cual consta de tres campos:

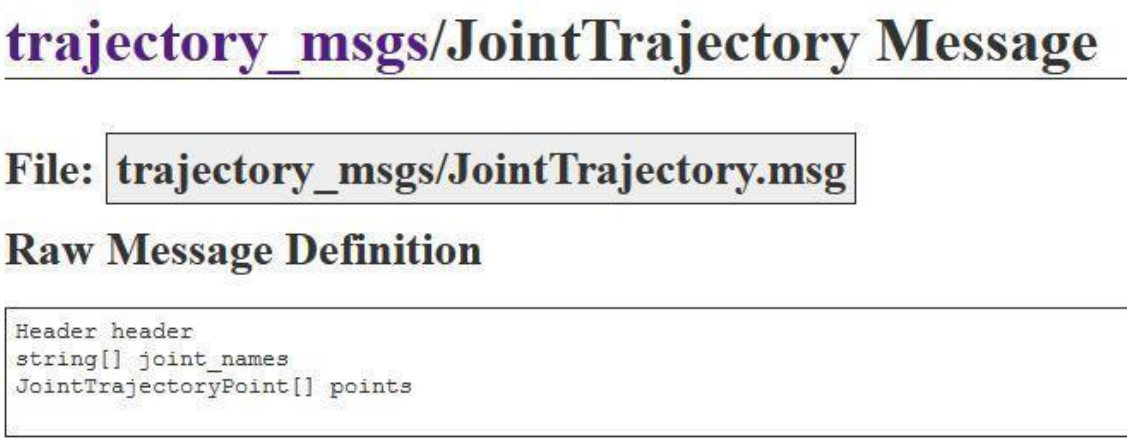

Figura 14: Estructura del mensaje *JointTrajectory*

El primer campo, llamado *header*, es una cabecera. La cabecera está formada por tres campos simples. Puesto que en este proyecto dejaremos la cabecera vacía, de forma que ROS la rellene por defecto, no vamos a tratar sus detalles.

Después, tenemos el campo *joint_names*. Este campo, el cual es un *array* de *strings*, sirve para nombrar las articulaciones del brazo. Se utilizará para controlar el orden de los valores para las diferentes articulaciones en el campo *positions* de cada punto.

Por último está el campo *points*. Se trata de un *array* de mensajes de tipo *JointTrajectoryPoint*. Este mensaje está formado por cuatro *arrays* de *floats* y un mensaje de tipo duration. Entraremos más en detalle en el siguiente punto, puesto que es el tipo de mensaje que se utilizará para el movimiento simple.

- Para el movimiento simple, utilizaremos mensajes de tipo *JointTrajectoryPoint*. Como hemos mencionado en el punto anterior, los mensajes de este tipo están compuestos por cuatro *arrays* de *floats* y un mensaje de tipo *duration*:

trajectory_msgs/JointTrajectoryPoint Message

File: trajectory_msgs/JointTrajectoryPoint.msg

Raw Message Definition

```
# Each trajectory point specifies either positions[, velocities[, accelerations]]
# or positions[, effort] for the trajectory to be executed.
# All specified values are in the same order as the joint names in JointTrajectory.msg

float64[] positions
float64[] velocities
float64[] accelerations
float64[] effort
duration time_from_start
```

Figura 15: Estructura del mensaje *JointTrajectoryPoint*

El primer *array*, el cual se llama *positions*, es el que en este proyecto utilizaremos para designar la posición a la que deberá moverse cada una de las articulaciones del brazo. Los *arrays velocities* y *accelerations* no los vamos a utilizar, pero se pueden utilizar para designar la velocidad y aceleración del movimiento de cada articulación. Mientras tanto, el *array effort* se utilizaría en vez de los dos anteriores *arrays*.

Nosotros, en cambio, utilizaremos el campo *time_from_start*. Este campo es un mensaje de tipo *duration* y está formado por un atributo *sec* y un atributo *nsec*. Utilizaremos el campo *sec* para designar la velocidad a la que se realizará el movimiento designado en el *array positions*.

```
ros::Duration::Duration ( int32_t _sec,
                          int32_t _nsec
                        )
```

Figura 16: Estructura del mensaje *duration*

Una vez explicados los mensajes a utilizar, hay que mencionar que tanto el campo *joint_names* como los campos *positions* y *time_from_start* tienen unas

limitaciones respecto a los valores que pueden tener. El valor de *time_from_start* oscilará entre 10 y 30 mientras que en el *array joint_names* tienen que aparecer los nombres de todas las articulaciones. El orden de éstos es indiferente, pero debe coincidir con el orden de los valores del *array positions* (en el movimiento simple se tendrá en cuenta el orden de la tabla). Los valores válidos se muestran en la siguiente tabla:

Tabla 7: Nombres y movimientos de las articulaciones

Articulación	**Nombre en *joint_names***	**Rango de valores en *position***
Base	*base*	0 - 180
Hombro	*shoulder*	15 - 165
Codo	*elbow*	0 - 180
Muñeca (Vertical)	*wrist_ver*	0 - 180
Muñeca (Rotación)	*wrist_rot*	0 - 180
Pinza	*gripper*	10 - 73 (abierta - cerrada)

Después de describir los mensajes y las limitaciones de éstos, podemos pasar a hablar sobre los nodos que compondrán el proyecto. Al igual que con los tipos de mensaje, será mejor mencionar por un lado los nodos y tópicos relacionados con el movimiento simple y por otro hacer lo propio con el movimiento de trayectoria.

- Para gestionar el movimiento simple tenemos un nodo subscriptor y un publicador. Ambos estarán escritos en lenguaje Python, por lo que deberemos importar la librería de ROS para Python; *rospy* . También debemos importar la librería correspondiente al tipo de mensaje a utilizar, que en este caso será *JointTrajectoryPoint*. Se importará el archivo de mensaje *String* del paquete *std_msgs* puesto que se utilizará en los nodos publicadores complementarios *info* y *debug*.

```python
#!/usr/bin/env python

import rospy

from std_msgs.msg import String
from trajectory_msgs.msg import JointTrajectoryPoint
```

Figura 17: Importaciones del nodo para movimientos simples

Después de importar las librerías necesarias, tendremos que definir la función *callback* a la que se llamará cuando se publique un mensaje en el tópico asociado. Esta función tiene un funcionamiento sencillo. Lee la información del mensaje recibido y valida que los valores dados en *positions* y en el campo *sec* de *time_from_start* son correctos. En caso de que así sea, publicará el mensaje en el tópico */simpleMovementArduino*, así como un mensaje informativo en el tópico */info*. Si por el contrario el mensaje es incorrecto, se publicará un mensaje en el tópico */info* con detalles sobre el error encontrado.

Para que todo funcione, hay que definir tanto el nodo subscriptor como el publicador y asignarlos a sus respectivos tópicos.

```python
def __init__(self):
    self.topicName = 'simpleMovementArduino'
    self.pubSMmsg = rospy.Publisher( self.topicName, JointTrajectoryPoint , queue_size = 5)

    rospy.Subscriber("simpleMovement", JointTrajectoryPoint , self.callbackSM)

    debug("Simple Movement node initialized.")
    info( False , "Simple Movement node is on.")
```

Figura 18: Definición de los nodos para el movimiento simple

El funcionamiento del código se explica mediante el siguiente diagrama:

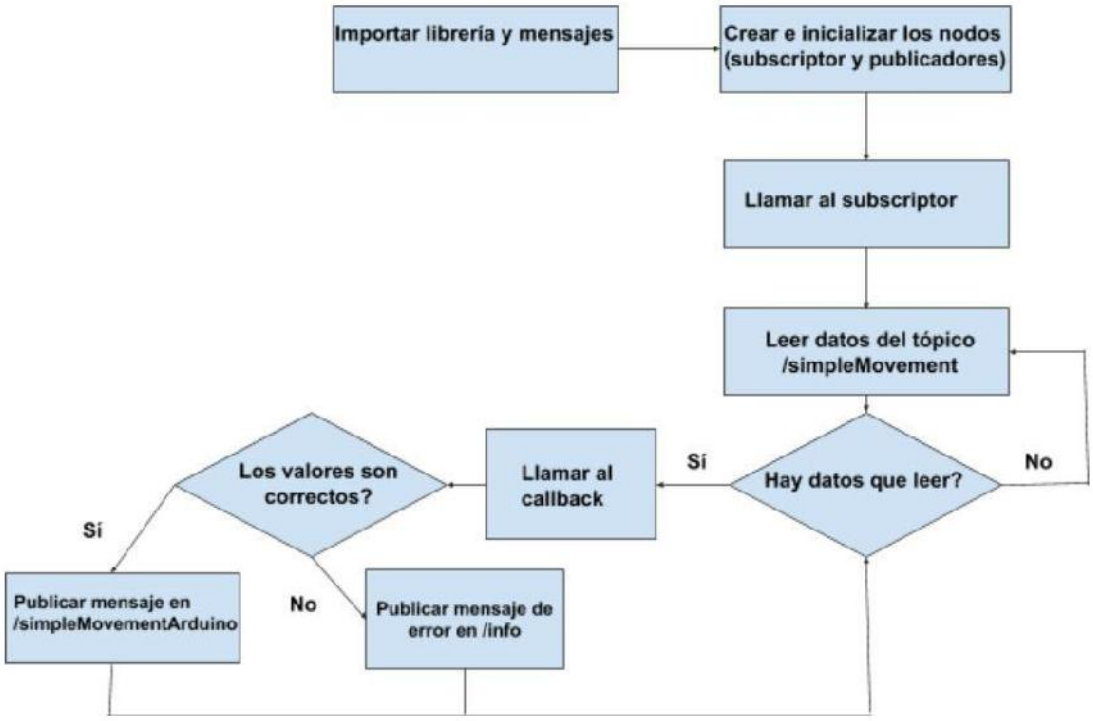

Figura 19: Funcionamiento nodo de movimiento simple

- Para el movimiento de trayectoria, al igual que con el movimiento simple, tenemos un nodo subscriptor y un publicador. Las librerías a importar son las mismas, salvo que debemos importar también el archivo para el tipo de mensaje *JointTrajectory*, puesto que será el que reciba el nodo subscriptor.

Tanto la función *callback* como el funcionamiento de este nodo son bastante parecidos a los del nodo de movimiento simple. La única diferencia consiste en que en este nodo, al tratarse de un mensaje del tipo *JointTrajectory*, habrá que comprobar que los valores del campo *joint_names* sean correctos, así como la información de cada uno de los puntos del campo *points*.

Cabe mencionar que el tópico al que estará subscrito se llama */trajectoryMovement* y el tópico donde se enviará el mensaje de tipo *JointTrajectory* en caso de que sea correcto será */trajectoryMovementArduino*.

9.2.2.- Librería *rosserial*

9.2.2.1.- Función que desempeñará

La función que desempeñará la librería *rosserial* es simple. Nos permitirá que la placa Arduino se conecte con el sistema operativo ROS, lo cual a su vez nos permitirá crear un nodo subscriptor directamente en Arduino. Eso evitará que tengamos que crear un protocolo para enviar los datos a la placa.

9.2.2.2.- Marco teórico

La librería *rosserial* define un protocolo para multiplexar diferentes tópicos y servicios a través de un puerto serie o un web socket, así como para enviar mensajes.

Para su funcionamiento, se requiere de un nodo en la máquina *host*, de forma que el protocolo serie pueda conectarse a la red general del sistema operativo ROS.

9.2.3.- Arduino

9.2.3.1.- Función que desempeñará

Arduino será una parte indispensable del proyecto, puesto que su función es la de controlar el brazo robótico. Esta parte del código se centrará en el control directo del brazo, leyendo la información de los mensajes recibidos para llamar a la función de movimiento del brazo con los valores adecuados.

9.2.3.2.- Marco teórico

Arduino es un lenguaje de programación el cual está basado en *C++*. Antes de comenzar a explicar como lo hemos implementado en el proyecto comentaremos la estructura y algunos de los comandos que vamos a utilizar [12].

Respecto a la estructura, los programas en Arduino están compuestos siempre por mínimo dos funciones, las cuales son indispensables.

- La función *setup( )* será la función en la que se inicializarán los pines, así como el puerto serie, y en la que se harán las declaraciones de las variables. Se ejecuta una única vez al inicio del programa, y es obligatoria, a pesar de que no haya ninguna declaración que hacer.

- La función *loop( )*, por otra parte, se ejecutará después de *setup( )*, de forma cíclica. Es la función que contendrá la mayor parte del código del programa.

Una vez comentada la estructura básica de Arduino, convendría mencionar algunas de las funciones que se van a utilizar en el proyecto:

- *Serial.begin(rate)*: Esta función se encarga de abrir el puerto serie con la velocidad especificada en el parámetro *rate*. Nosotros utilizaremos el valor 9600, puesto que es la más utilizado a la hora de abrir una comunicación con el ordenador. Debido a que no abriremos el puerto serie más de una vez, llamaremos a esta función dentro de la función *setup( ).*

- *Serial.println(data)*: Al igual que la función *Serial.print(data)*, se encarga de imprimir los datos a través del puerto serie. La diferencia entre ambas es que la primera envía un carácter *new line* cada vez que se la llama.

- *NodeHandle.initNode( )*: Esta función, la cual ejecutaremos tan solo una vez, inicializará el nodo de ROS, para que podamos después subscribirlo a uno o más tópicos.

- *NodeHandle.subscribe(subscriber)*: Inicia el suscriptor enviado como parámetro al tópico definido al crear el primero.

- *NodeHandle.spinOnce( )*: Coge el primer mensaje de la cola de mensajes recibidos y llama a la función *callback* correspondiente. Es importante llamar a la función con la suficiente frecuencia para perder el menor número de mensajes posibles, puesto que el tamaño de la cola es limitado.

9.2.3.3.- Aspectos prácticos

A la hora de implementar el código de Arduino para este proyecto, hay que tener en cuenta dos situaciones. En la primera situación no se utilizará el brazo robótico, sino que se simulará a través del adaptador *TTL-USB* como se muestra en la *Figura2*, mientras que en la segunda situación se controlará directamente el brazo.

- En la primera situación, necesitaremos simular un puerto serie software a través del adaptador *TTL-USB*. Para ello tendremos que abrir el puerto serie, lo cual haremos dentro de la función *setup( )*. Asimismo, en la misma función inicializaremos el objeto *NodeHandle*, así como los dos nodos subscriptores de los que dispondrá el programa. Para ello, primero hemos creado las dos funciones *callback*, una para cada uno de los nodos, las cuales explicaremos más adelante.

```
void setup()
{

  Serial3.begin(9600);

  nh.initNode();

  nh.subscribe(subSM);
  nh.subscribe(subTM);

}
```

Figura 20: Función *setup* del driver de Arduino

En la función *loop( )*, tan solo se llamará a *nh.spinOnce( )* y después se hará un *delay* de 10 milisegundos. De esta forma, se gestionará uno de los mensajes

recibidos cada 10 milisegundos (algo más contando con el tiempo de ejecución de la función *callback* correspondiente), lo cual es un tiempo razonable.

Las funciones *callback* son muy parecidas entre sí, y también son muy parecidas a las funciones *callback* de los nodos escritos en Python. Ambas comienzan haciendo una comprobación de que los valores recibidos en el mensaje son correctos. A pesar de haber hecho una comprobación en el nodo de Python, al poder publicar los mensajes directamente en los tópicos */simpleMovementArduino* y */trajectoryMovementArduino* directamente desde ROS, es mejor hacer una pequeña comprobación, para asegurarnos de que el movimiento a realizar es posible. Una vez comprobado, se procederá a extraer los valores necesarios del mensaje, y se enviarán los valores por el puerto serie software.

- La segunda situación es prácticamente idéntica a la primera. La diferencia está en que deberemos incluir las librerías necesarias para controlar el brazo, así como sustituir la función de iniciar el puerto serie por la de inicializar el brazo. Además, llamaremos a la función *Braccio.ServoMovement(step delay, base, shoulder ,elbow , wrist vertical, wrist rotation, gripper)* con los valores que en la primera situación se enviaban por el puerto serie software para que el brazo se mueva.

Para comprender mejor el funcionamiento de las funciones *callback*, tenemos el siguiente diagrama de flujo, que muestra tanto la función para el movimiento simple como para el movimiento de trayectoria.

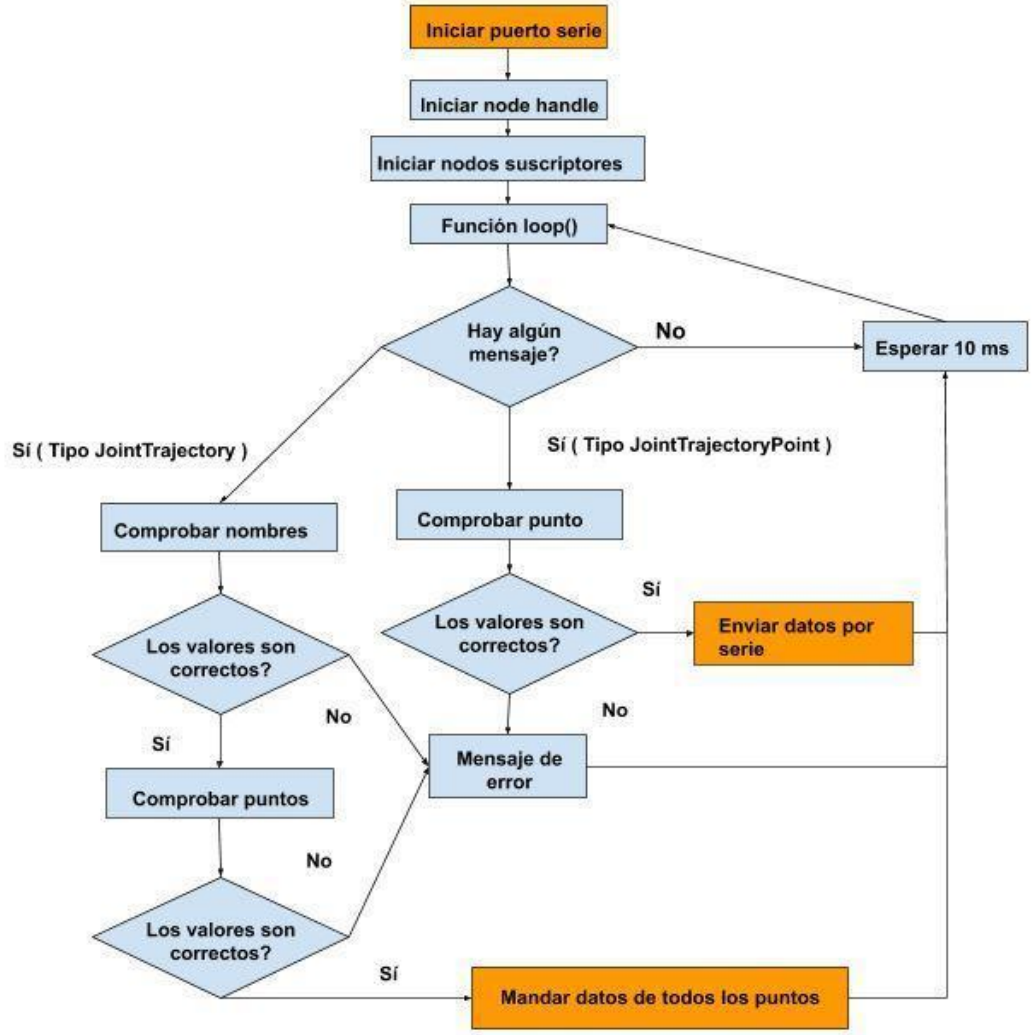

Figura 21: Funcionamiento movimiento simple y de trayectoria en Arduino

Como se puede ver, el funcionamiento de ambos es bastante similar, siendo el movimiento de trayectoria un poco más complejo que el movimiento simple. Los cuadros en naranja son aquellos que en la segunda situación serán sustituidos por las funciones relacionadas con el brazo robótico, siendo el primero la inicialización y los otros dos el movimiento.

10.- Análisis de resultados

Una vez terminado el proyecto, es hora de revisar los objetivos propuestos en el apartado 3 para analizar si se han cumplido o no. Como bien habíamos definido, el objetivo principal era crear un *driver* para Arduino que permitiera controlar el brazo robótico *Tinkerkit Braccio* mediante el sistema operativo ROS, con ayuda de la librería *rosserial*. Tal como hemos mostrado en el apartado de diseño, ese objetivo se ha cumplido.

Con la finalidad de cumplir el objetivo principal, se establecieron unos objetivos secundarios, los cuales pasaremos a comentar:

- Se han adquirido los conocimientos necesarios de ROS para diseñar un *driver* que permita la conexión entre ROS y el brazo robótico.

- Se han adquirido los conocimientos necesarios para utilizar la librería *rosserial* de forma que el *driver* esté conectado con la red de ROS.

- Se han diseñado dos nodos en Python, que son capaces de leer los mensajes de un tópico, validan que la información sea correcta y luego los publican en otro tópico diferente.

- Se han adquirido los conocimientos necesarios del lenguaje de programación Arduino para poder crear un *driver* que controle el brazo a partir de un mensaje de ROS.

11.- Plan de proyecto y planificación

Después de concluir el diseño del proyecto, es hora de comenzar a crear un plan de trabajo en el cual habrá que explicar las tareas a llevar a cabo, así como el grupo que se encargará de ello.

11.1.- Descripción del equipo

En la siguiente tabla se mostrará el equipo encargado de este proyecto, el cual está formado por dos personas, así como el cargo desempeñado por cada uno.

Tabla 8: Nombres y cargos de los integrantes del equipo del proyecto

Miembro	Cargo desempeñado
Oskar Casquero Oyarzabal	Director del proyecto
Joaquin Pereda Bados	Ingeniero junior

11.2.- Fases y tareas

El trabajo a realizar para llevar a cabo el proyecto se divide en varios paquetes, los cuales a su vez estarán divididos en varias tareas y cuentan con un tiempo límite para completarse. Además de una tabla con los datos más importantes de cada uno de los paquetes, se explica cada uno de ellos con más detalle, mencionando las tareas que los componen.

Tabla 9: Paquetes de trabajo del proyecto

Paquete de trabajo	Nombre	Comienzo	Final
Paquete de trabajo 1	Gestión de proyecto	Día 1	Día 104
Paquete de trabajo 2	Análisis de alternativas	Día 1	Día 10
Paquete de trabajo 3	Preparación de los equipos	Día 10	Día 12
Paquete de trabajo 4	Formación	Día 12	Día 44
Paquete de trabajo 5	Desarrollo	Día 44	Día 94
Paquete de trabajo 6	Documentación	Día 94	Día 104

- Paquete de trabajo 1: Gestión del proyecto:

En este paquete se organiza y ejecuta correctamente el proyecto. Con ese fin, se encarga de coordinar el resto de paquetes, de forma que el proyecto se pueda terminar dentro del plazo establecido.

Responsable: Oskar Casquero Oyarzabal

Participante: Joaquin Pereda Bados

- Paquete de trabajo 2: Análisis de alternativas:

En este paquete se buscan y estudian diferentes alternativas para aquellos aspectos del proyecto que no están especificados dentro de los requisitos del sistema, de forma que se pueda encontrar la opción más apropiada

Responsable: Oskar Casquero Oyarzabal

Participante: Joaquin Pereda Bados

Duración: 10 días

- Paquete de trabajo 3: Preparación de los equipos:

Esta fase es la encargada de la instalación del software necesario. Comienza por instalar el sistema operativo Ubuntu en el ordenador, para después hacer los mismo con el sistema operativo ROS, el cual se instala sobre Ubuntu. Una vez terminado eso, hay que instalar el lenguaje de programación *Python* así como los paquetes necesarios en Arduino y la librería *rosserial*.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 2 días

- Paquete de trabajo 4: Formación:

Aquí los participantes adquieren los conocimientos necesarios para llevar a cabo el desarrollo del proyecto.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 32 días

- Tarea 1: Familiarización con Ubuntu:

En esta tarea se conocen los comandos básicos del sistema operativo necesarios para poder llevar a cabo las siguientes tareas.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 2 días

- Tarea 2: Familiarización con Python:

Puesto que Python será el lenguaje programación escogido para desarrollar una parte importante del proyecto, es necesario conocer su funcionamiento y sintaxis para poder trabajar.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 10 días

- Tarea 3: Familiarización con ROS:

Esta tarea se centra en conocer el sistema operativo ROS. Para ello hay que aprender su estructura y el funcionamiento del sistema de nodos y tópicos, además de como programarlos.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 6 días

- Tarea 4: Familiarización con Arduino:

Al igual que Python, Arduino es una parte indispensable, puesto que es la forma en que se controla el brazo robótico. Por ello, en esta tarea se aprenden los conceptos de Arduino necesarios para controlar el brazo.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 10 días

- Tarea 5: Familiarización con *rosserial*:

Por último y una vez completadas las anteriores tareas, nos centramos en la librería *rosserial*, la cual nos permite conectar la placa Arduino con la red de ROS, y por ende con el resto del proyecto.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 4 días

- Paquete de trabajo 5: Desarrollo:

Esta fase engloba el diseño requerido para llevar a cabo el proyecto.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 50 días

- Tarea 1: Identificación de los tópicos y nodos necesarios:

Para comenzar con el desarrollo, se piensa cuántos nodos se necesitan para llevarlo a cabo, además del tipo de estos y del mensaje o mensajes a utilizar.

Responsable: Joaquin Pereda Bados

Participante: Joaquin Pereda Bados

Duración: 3 días

- Tarea 2: Diseño y programación de los nodos en Python:

Una vez definidos los nodos a crear, pasamos a programarlos en Python.

Responsable: Oskar Casquero Oyarzabal

Participante: Joaquin Pereda Bados

Duración: 20 días

- Tarea 3: Diseño y programación del *driver* de Arduino:

Después de crear los nodos de ROS, creamos el *driver* de Arduino, que es el que se encarga de controlar el brazo robótico

Responsable: Oskar Casquero Oyarzabal

Participante: Joaquin Pereda Bados

Duración: 25 días

- Tarea 4: Conexión de la placa Arduino con el brazo robótico:

Para finalizar el desarrollo, sustituimos la parte del código de Arduino que utilizamos para la validación por las funciones correspondientes de la librería del brazo robótico, para así poder controlarlo.

Responsable: Oskar Casquero Oyarzabal

Participante: Joaquin Pereda Bados

Duración: 2 días

- Paquete de trabajo 6: Documentación:

Para finalizar el proyecto, se debe crear un documento que nos sirva para detallar todo el proceso realizado, de forma que pueda servir también como un manual de referencia o una ayuda para otros proyectos realizados con posterioridad.

Responsable: Oskar Casquero Oyarzabal

Participante: Joaquin Pereda Bados

Duración: 10 días

12.- Diagrama Gantt

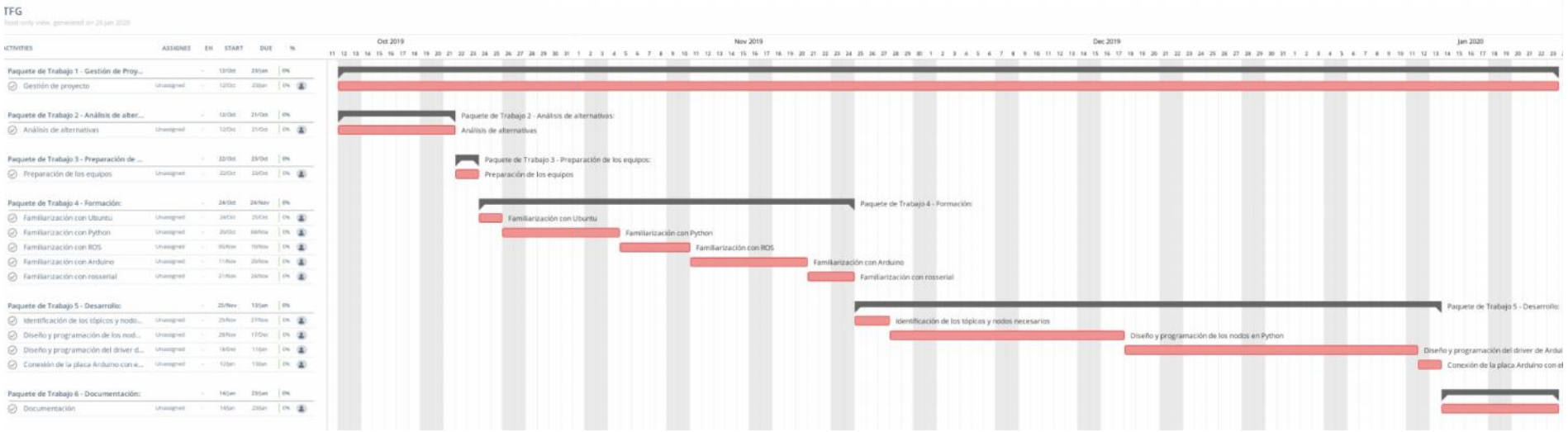

Figura 22: Diagrama Gantt del proyecto

13.- Presupuesto

Tras mostrar las tareas a realizar y la duración de cada una de ellas, pasamos a valorar el aspecto económico. Para ello, se calculan dos presupuestos distintos.

El primer presupuesto recoge los costes de desarrollo del proyecto, mientras el segundo se centra en los costes de ejecución. Para los costes de desarrollo se tienen en cuenta las horas que los integrantes del equipo necesitan para adquirir los conocimientos necesarios para poder desarrollar una solución, mientras en los costes de ejecución se cuenta el desarrollo, así como los costes del brazo robótico, la placa Arduino y el adaptador *TTL-USB*.

Tabla 10: Presupuesto de costes de desarrollo

Concepto	**Unidades**	**Coste unitario**	**Número de unidades**	**Coste total**
Horas internas				8.950,00€
Director	horas	60.00€	50	3.000,00€
Ingeniero junior	horas	35.00€	170	5.950,00€
Amortizaciones				44,00€
Ordenador	horas	0.20€	220	44,00€
Costes directos				8.994,00€
Costes indirectos		*10% de los directos*		899.40€
Subtotal				9.893,40€
Imprevistos		*10% del subtotal*		989,34€
Total				10.882,74€

Dentro de los costes de desarrollo, se cuentan también las amortizaciones del equipo informático, debido a que es un proyecto con el fin de desarrollar un software. Además de eso, se destina un 10% de los costes directos para cubrir cualquier gasto indirecto, y un 10% del subtotal para cubrir posibles imprevistos.

Tabla 11: Presupuesto de costes de ejecución

Concepto	**Unidades**	**Coste unitario**	**Número de unidades**	**Coste total**
Horas internas				1.650,00€
Director	horas	60.00€	10	600,00€
Ingeniero junior	horas	35.00€	30	1.050,00€
Amortizaciones				8,00€
Ordenador	horas	0.20€	40	8,00€
Gastos				241,49€
Tinkerkit Braccio		200.00€	1	200,00€
Arduino *Mega 2560*		35.50€	1	35,50€
Adaptadot *TTL-USB*		5.99€	1	5,99€
Costes directos				1.899,49
Costes indirectos		*10% de los directos*		189,94
Subtotal				2.089,43
Imprevistos		*10% del subtotal*		208,94
Total				2.298,37€

Como podemos comprobar, los costes de ejecución son menores que los de desarrollo. Eso se debe mayormente a que el número de horas es significativamente más pequeño al ya disponer los miembros del equipo de los conocimientos necesarios. A estos costes, hay que añadirle el coste del brazo robótico, así como el del adaptador y el de la placa. Seguimos destinando un 10% a los costes indirectos e imprevistos.

14.- Conclusiones

En este último apartado, se habla sobre las conclusiones a las que se ha llegado en este trabajo.

La primera es que el sistema operativo ROS, además de ser un sistema robusto para interconectar diferentes tipos de robots entre sí, es una herramienta que facilita mucho el trabajo en el ámbito. Gran parte de ello se debe a que los conceptos básicos para su uso son fáciles de aprender, además de tener una amplia comunidad y un gran número de tutoriales que facilitan aún más el aprendizaje.

A pesar de que pudiese parecer innecesario, el crear los nodos en Python abre la posibilidad para en futuros proyectos añadir nuevas funcionalidades, como la de guardar diferentes puntos para después poder utilizarlos sin tener que escribir todo el movimiento.

La última conclusión se trata principalmente de que, a pesar de no adquirir los conocimientos necesarios para llevar a cabo un trabajo de características, habiendo hecho un grado en Ingeniería Informática de Gestión y Sistemas de Información, debido a que la robótica ha sido un tema que casi no se ha tratado durante todo el grado, sí que se han adquirido unas bases que permiten, una vez aprendidos los conceptos básicos sobre la plataforma o el lenguaje a utilizar, poder desarrollar una solución al problema planteado de una manera sencilla.

Bibliografía

[1] La automatización de la industria. 1993, de AITIM. Sitio web: https://infomadera.net/uploads/articulos/archivo_2116_17668.pdf

[2] R. González, V. Robots industriales. Sitio web: http://platea.pntic.mec.es/vgonzale/cyr_0204/ctrl_rob/robotica/industrial.htm

[3] Robot TinkerkitBraccio, tienda oficial de Arduino. Sitio web: https://store.arduino.cc/tinkerkit-braccio-robot

[4] Braccio: brazo robótico controlable con Arduino. Sitio web: https://www.mcielectronics.cl/shop/product/braccio-brazo-robotico-controlable-con-arduino-25457

[5] Wiki de ROS en español. Sitio web: http://wiki.ros.org/es

[6] García Zarraga, N. Diseño y desarrollo de un paquete ROS para el control del brazo manipulador Tinkerkit Braccio. Sitio web: http://hdl.handle.net/10810/37061

[7] Definición del mensaje JointTrajectory, documentos de ROS. Sitio web: http://docs.ros.org/kinetic/api/trajectory_msgs/html/msg/JointTrajectory.html

[8] Definición del mensaje JointTrajectoryPoint, documentos de ROS. Sitio web: http://docs.ros.org/kinetic/api/trajectory_msgs/html/msg/JointTrajectoryPoint.html

[9] Placa Arduino MEGA2560, tienda oficial de Arduino. Sitio web: http://store.arduino.cc/arduino-mega-2560-rev3

[10] Comando rosnode, ROS Wiki. Sitio web: http://wiki.ros.org/rosnode

[11] Maestro de ROS, ROS Wiki. Sitio web: http://wiki.ros.org/Master

[12] Ruiz Gutiérrez, J.M. Manual de programación de Arduino. Sitio web: https://arduinobot.pbworks.com/f/Manual+Programacion+Arduino.pdf