Universidad de La Laguna
Escuela Superior de Ingeniería y Tecnología
Departamento de Ingeniería Industrial

ESCUELA SUPERIOR DE INGENIERÍA
Y TECNOLOGÍA

Trabajo de Fin de Grado

Desarrollo e implementación de un controlador multiplataforma para el Scorbot ER-4U

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Estudiante: Yolanda Mercedes Gimeno Rodríguez
Estudiante: José Luis Pérez Pérez

Tutor: Fernando Luis Rosa González
Cotutor: Iván Rodríguez Méndez

9 de julio de 2020

Universidad de La Laguna

Escuela Superior de Ingeniería y Tecnología
Departamento de Ingeniería Industrial

IMPRESO DE AUTORIZACIÓN DEL TRABAJO DE FIN DE GRADO POR LOS TUTORES

Curso 2019/2020

D. Fernando Luis Rosa González, **con D.N.I.** 43611314-W, **como tutor** y **D.** Iván Rodríguez Méndez **con D.N.I.** 43380782-E **como contutor de los estudiantes Dña.** Yolanda Mercedes Gimeno Rodríguez y **D.** José Luis Pérez Pérez **en el Trabajo de Fin de Grado titulado**

Desarrollo e implementación de un controlador multiplataforma para el Scorbot ER-4U

damos nuestra autorización, acreditada por la firma electrónica de este documento, para la presentación y defensa de este proyecto, a la vez que confirmamos que los estudiantes han cumplido con los objetivos generales y particulares que lleva consigo la realización del mismo.

La Laguna, a 9 de julio de 2020

Agradecimientos

Agradecer en especial a nuestro contutor, Iván Rodríguez Méndez, por ofrecernos la oportunidad de trabajar en una idea que había estado barajando y que ha resultado en este proyecto. Gracias por la predisposición que desde el primer momento nos has brindado, guiándonos y aconsejándonos para llegar al objetivo marcado.

Al personal docente por su dedicación y esfuerzo a lo largo de estos años, formándonos y convirtiéndonos así en profesionales. Al personal de administración y servicios por siempre resolver y asistir nuestras peticiones con amabilidad.

Gracias a nuestros familiares y amigos, son quienes nos dieron apoyo para comenzar esta aventura y ánimos para continuar en los momentos de debilidad.

También dar gracias a nuestros queridos amigos, compañeros que nos brindó la carrera y amistades que se asentaron tras muchas horas de estudio y trabajo juntos. Sin ustedes esto no sería posible hoy.

A ti Yolanda, por apostar por este proyecto que en sus inicios se veía lejano conseguir. Soñar en solitario no es tan divertido como hacerlo acompañado. Por este y por futuros proyectos, gracias.

A ti Jose, gracias por hacer de este proyecto una experiencia grata, por confiar en mi para realizar de forma conjunta este proyecto, por sacar lo mejor de mi en cada momento y por ser mi apoyo incondicional.

Universidad de La Laguna
Escuela Superior de Ingeniería y Tecnología
Departamento de Ingeniería Industrial

ESCUELA DE SUPERIOR DE
INGENIERÍA Y TECNOLOGÍA

Trabajo de Fin de Grado

Desarrollo e implementación de un controlador multiplataforma para el Scorbot-ER4U

TOMO I

Memoria

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Estudiante: Yolanda Mercedes Gimeno Rodríguez
Estudiante: José Luis Pérez Pérez

Tutor: Fernando Luis Rosa González
Cotutor: Iván Rodríguez Méndez

9 de julio de 2020

Índice general

Índice de figuras

Índice de tablas

Resumen

El significado de la palabra ingeniería engloba todos aquellos conocimientos que puedan orientarse hacia la creación de nuevas herramientas. Sin embargo, no solo se debe emplear la ingeniería en el desarrollo de nuevos utensilios, sino sobretodo en el desarrollo de métodos que permitan dar un nuevo uso a tecnologías que hayan quedado obsoletas. Este proyecto defiende este ideal al implementar una primera versión de un programa de control multiplataforma para un brazo robótico. Como punto de referencia, se ha elegido para el desarrollo de este trabajo un modelo de brazo robótico cuyas restricciones de uso han hecho que se deje a un lado como herramienta de trabajo, el Scorbot 4R-EU. El programa de control para este modelo de brazo robótico solo es ejecutable sobre sistemas operativos *Windows XP* y su extensión de uso a otras plataformas puede dar gran versatilidad a este tipo de equipos, sobre todo en el ámbito educativo. A lo largo de este proyecto se expondrá tanto el desarrollo de las funcionalidades como de la parte gráfica del programa.

Abstract

The meaning of the word engineering encompasses all those knowledge that can be oriented towards the creation of new tools. However, engineering should not only be used for the development of new utensils, it should also be used to the development of methods that make it possible to give new use to technologies that have become obsolete. This project defends this ideal by implementing a first version of a multi-platform control program for a robotic arm. As a point of reference, it has been choosen for the development of this work a robotic arm model in order of its restrictions of use which have led it to be set aside as a working tool, the Scorbot 4R-EU. The control program for this robotic arm is only executable on Windows XP operating systems and its extension to other platforms can give great versatility to this type of equipment, especially in the educational field. Throughout this project, both the development of the functionalities and the graphic part of the program will be exposed.

Capítulo 1

Introducción

1.1. Róbotica

Se tiene conocimiento de los inicios de lo que conocemos ahora como robótica con los tele-manipuladores, desarrollados por el ingeniero americano Raymond C. Goertz del Argonne National Laboratory en 1948. Estos instrumentos son controlados por un operario que realiza las operaciones necesarias desde un manipulador maestro en una zona segura y el manipulador esclavo reproduce las acciones realizadas. Estos dispositivos fueron desarrollados para evitar la exposición de los operarios ante elementos radiactivos en la industria nuclear. Su uso en entornos peligrosos y hostiles para el aumento de la protección y seguridad del operario son algunas de las atractivas características que hizo a estos aparatos revolucionar la industria. Es tal la magnitud de su evolución hasta la actualidad, que gracias a la precisión alcanzada se comienzan a ver robots cirujanos (ver Figura 1.1) dirigidos por profesionales que afirman poder llegar a realizar operaciones imposibles con el método tradicional.

Figura 1.1: Robot destinado a la cirugía

"La sustitución del operador por un programa de ordenador que controlase los movimientos del manipulador dio paso al concepto robot" [7]. Este nuevo concepto de manipulador, dio lugar a una transformación de la industria. Las fábricas del sector automovilístico vieron en esto un impulso a la automatización de la plantas de ensamblaje. La competitividad entre compañías dio lugar a robots capaces de obrar en casi cualquier entorno y tipo de labor.

Según la Asociación de Industrias Robóticas (RIA), un robot industrial es un manipulador multi-funcional re-programable, capaz de mover materias, piezas, herramientas o dispositivos especiales, según trayectorias variables, programadas para realizar tareas diversas.

En la actualidad encontramos robots manipuladores con diferentes morfologías y aplicaciones. Las aplicaciones más comunes dentro de la indus-

tria pueden ser la soldadura, trabajos repetitivos o ensamblaje. También pueden ser útiles para añadir ergonomía a los trabajadores de planta. Como vemos en las imágenes de la Figura 1.2 el manipulador que cuenta con un sistema de visión y una ventosa por efector final, entrega al operario las piezas disminuyendo a este movimientos repetitivos.

Figura 1.2: Robot trabajando a mano con el personal

1.2. Objetivos

El proyecto objeto de esta memoria parte de la idea de crear herramientas de libre distribución y acceso que permitan redefinir las funcionalidades de equipos ya obsoletos.

El brazo robótico Scorbot ER-4U es una versión de brazo antropomórfico cuya característica destacable en relación a este proyecto es que requiere de un sistema operativo Windows XP para operar. Esta condición implica tener un equipo dedicado para trabajar con él, lo que supone, para un laboratorio, disponer de más recursos en sentido de espacio y software.

Con este Trabajo de Fin de Grado se pretende:

1. Establecer las bases para el control de un brazo robótico.

2. Realizar un programa de control multiplataforma.

3. Orientarlo al ámbito educativo.

En las siguientes páginas se expondrá el proceso llevado a cabo para generar un sistema de comunicación con el brazo robótico Scorbot ER-4U, así como el desarrollo del software resultante.

1.3. Cronología del desarrollo

El punto de partida de cualquier proyecto comprende la planificación temporal y distribución de los distintos aspectos que lo conforman. En el caso de este proyecto de desarrollo de software, el trabajo se puede fraccionar en cuatro fases, cuyo organigrama se detalla en la el apartado A.1 del Anexo A.

- **FASE 0:** Estudio y establecimiento de la comunicación.
- **FASE 1:** Desarrollo del protocolo de comunicación.
- **FASE 2:** Generación de trayectorias e interfaz de usuario.
- **FASE 3:** Optimización del programa y redacción de la memoria.

El inicio de este proyecto viene dado por una etapa de documentación y aprendizaje autónomo sobre los protocolos de comunicación USB y la conectividad entre dispositivos. Establecer la conexión entre la controladora USB del brazo robótico y nuestro dispositivo de control fue el principal objetivo de esta **Fase 0**.

Conseguido esto, se inicia la **Fase 1** donde se busca mantener esa conexión en el tiempo. Es aquí donde se crea el concepto de hilo de sincronismo, siendo este la parte del programa encargada de mantener una comunicación continua con la controladora. Establecida la sincronización entre dispositivos, el siguiente hito es el control sobre los elementos que

accionan el movimiento físico del manipulador, los motores. Se da por finalizada la fase al completar efectivamente los movimientos de los 5 ejes que permite el brazo robótico.

Teniendo ya implementada la funcionalidad básica de movimientos articulares simples, en la **Fase 2** se pasa a implementar un control del robot manipulador mediante trayectorias. Para ello, es requisito tener una posición de referencia, también conocida como *home*. Definida la referencia, se desarrolla un modelo cinemático basado en el método de Denavit – Hartenberg para resolver el problema de la cinemática inversa, cuyo resultado permitirá obtener las variaciones a ejecutar en el robot para alcanzar el objetivo definido en el plano cartesiano.

De forma paralela al desarrollo del control de movimientos mediante trayectorias, se ha llevado a cabo el diseño de la interfaz gráfica de usuario que pondrá en conjunto las diferentes herramientas de control del Scorbot ER-4U.

Finalizada la interfaz de usuario, se da comienzo a la **Fase 3** cuyo fin consiste en la optimización y comprobación del programa y en la redacción y documentación del trabajo descrito.

1.4. Distribución de la memoria

La presente memoria se divide en tres partes diferenciadas: Memoria, Pliego de Condiciones y Anexos.

En primer lugar, la **Memoria** comprende todo el planteamiento seguido a lo largo del desarrollo del proyecto así como los resultados obtenidos. De los capítulos 2 al 5 se detallan los componentes por separado de mayor relevancia que intervienen en el desarrollo de las posteriores etapas. Éstas se corresponden con los capítulos 6 a 11 donde se desglosa la implementación de cada sección que compone el programa final.

En segundo lugar, se presenta el **Pliego de condiciones y presupuesto** donde se detallan los condicionantes a los que se debía acoger el desarro-

llo del proyecto desde el principio, así como un desglose del presupuesto necesario para llevar acabo el proyecto.

Finalmente aparece un apartado de **Anexos** en el que se adjunta el cronograma del proyecto, conceptos y razonamientos que afianzan los desarrollos realizados a lo largo de la Memoria, pero que no tienen impacto directo sobre la misma.

Capítulo 2

Herramientas y metodología

2.1. Herramientas

2.1.1. Sistemas operativos

Distribución Linux

Sistema operativo Linux Mint 19.2 Tina [2], distribución de GNU/Linux basada en Debian y Ubuntu con el kernel 4.15.0-54-generic. Su fácil instalación la convierten en un sistema perfecto para realizar un proyecto de estas características.

Windows XP

Sistema operativo Windows XP Profesional, versión de 2002 con Service Pack 3.

VirtualBox

Entorno de virtualización de sistemas operativos en código abierto, bajo los términos de la Licencia Pública General de GNU(GPL), desarrollado

tanto para empresas como para particulares. Se hace uso de la versión 6.1.6 r137129.

2.1.2. Análisis de datos

Wireshark

Version 2.6.10 Wireshark. Software con el que monitorizar el tráfico a través de cualquier vía de comunicación, en este caso para monitorizar la comunicación entre la controladora USB y el ordenador. Es a través de este programa como se consigue determinar como interactúan ambos equipos.

Libre Office Calc

Programa de hoja de cálculo de código abierto para la visualización de los datos extraidos desde el Wireshark. Se usan archivos con extensión CSV. La versión del programa empleada es la 6.0.7.3

Python

Lenguaje con el que se ha construido la funcionalidad y estructura del programa [4], así como los filtros de análisis de datos y la unión entre el código del programa y la interfaz de usuario. La versión de Python utilizada es la 3.6.9 [3][1].

2.1.3. Diseño de la interfaz gráfica

Qt Designer

Es una herramienta de Qt [5] para el diseño y desarrollo de interfaces gráficas basado en el uso de Widgets. Versión utilizada 5.9.5. El soporte

gráfico para el desarrollo de interfaces que ofrece esta herramienta la convierten en un recurso muy útil a la hora de crear una interfaz gráfica de usuario con elementos básicos.

2.2. Metodología

El desarrollo del proyecto sigue una dinámica de trabajo por etapas con la idea de realizar una evolución progresiva mediante el alcance de hitos. En la Figura A.1 del Anexo A se desglosan las etapas en las que se divide el proyecto junto con los objetivos de cada una de ellas, formando en conjunto el desarrollo del trabajo llevado a cabo. A continuación se describe con más detalle la línea de progresión de trabajo adoptada a lo largo del desarrollo del proyecto.

En primer lugar se realiza un estudio de la comunicación entre el programa Scorbase y la controladora USB. La idea de este primer análisis es encontrar secuencias de mensajes asociadas al proceso de conexión entre la controladora USB y el programa que permitan generar un algoritmo estándar para realizar la conexión con el dispositivo. Además se hace patente la necesidad de implementar un segundo algoritmo de ejecución cíclica que mantenga la conexión abierta, de forma que el programa y la controladora USB se mantengan sincronizados, tal como se representa en la Figura 2.1.

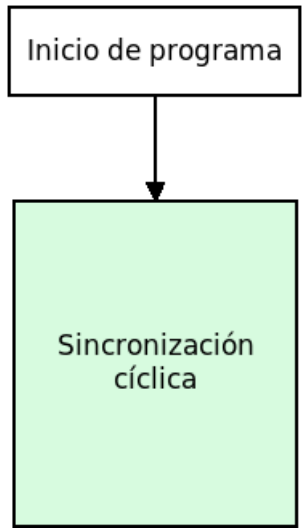


Figura 2.1: Comunicación y sincronización inicial

Adicionalmente, durante el análisis inicial también se aprecia que la comunicación con la controladora USB implica por un lado el envío constante de mensajes desde la aplicación que indiquen al dispositivo que el programa sigue conectado y por otro lado el envío de mensajes que constituyen las órdenes que la controladora USB debe llevar acabo. La presencia de esta doble funcionalidad lleva a desarrollar e implementar un programa con una estructura de dos hilos de ejecución paralela, tal como se puede apreciar en la Figura 2.2, donde uno será el encargado de mantener la sincronización con la controladora y el otro será el encargado de transmitir los mensajes de órdenes.

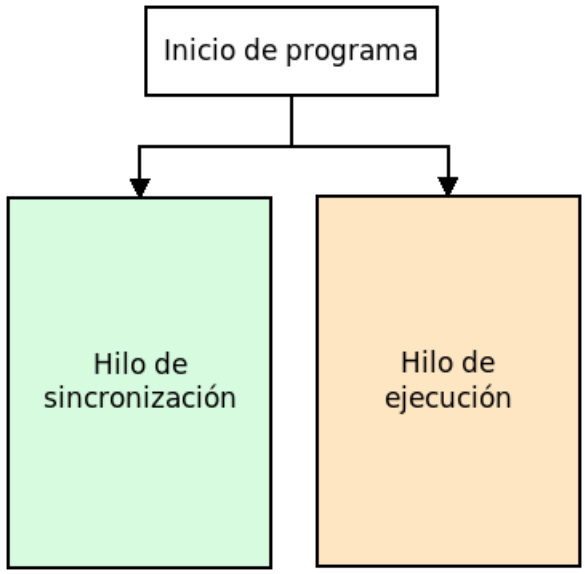


Figura 2.2: Estructura inicial del programa

Teniendo creada una herramienta que permite enviar órdenes a la controladora USB, se inicia el análisis de los mensajes necesarios para llevar acabo el accionamiento de los distintos motores que componen el brazo robótico. Con el fin de facilitar la lectura y comprensión de los mensajes de comunicación, se han desarrollado un conjunto de filtros en lenguaje Python con los que filtrar los datos y organizarlos según el criterio más adecuado, así como realizar representaciones gráficas que faciliten la comprensión de los mismos. Los filtros utilizados vienen descritos en el Anexo C. El resultado de este proceso es la implementación de un algoritmo que genera secuencias de mensajes capaces de indicar a la controladora USB qué acciones debe llevar acabo y cómo ha de hacerlo, como el movimiento del motor de la base o el del codo.

El siguiente paso es el desarrollo de una herramienta que facilite la interacción entre el programa y el usuario. Es aquí donde aparece la ne-

cesidad de implementar una interfaz gráfica de usuario donde se pongan a disposición del usuario todas las funcionalidades desarrolladas para el control y accionamiento del robot manipulador. Así pues, aparece un tercer elemento dentro de la estructura del programa, tal como puede verse en la Figura 2.3, que interactúa con el hilo de ejecución para la realización de las acciones indicadas por el usuario.

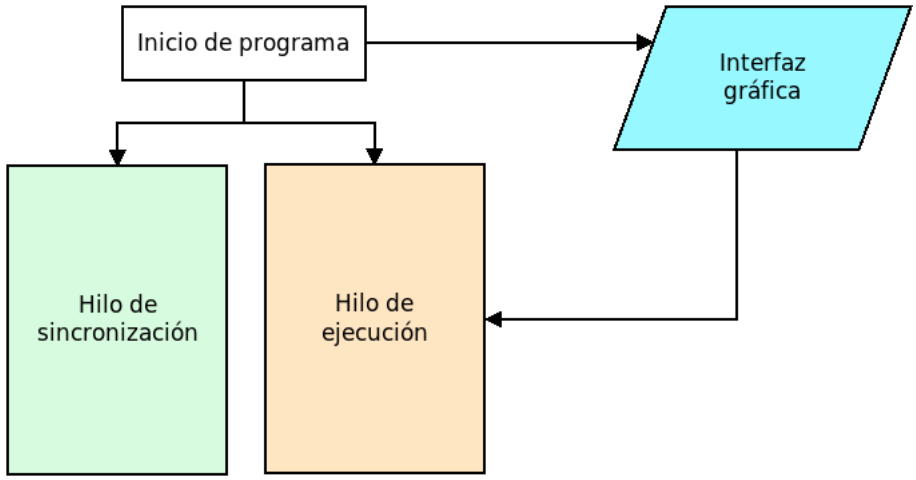


Figura 2.3: Implementación de la interfaz gráfica en el programa

El siguiente paso es el desarrollo de un algoritmo que permita al robot buscar la posición *home*, funcionalidad que se considera básica para un programa de estas características por la relación que tiene con el desarrollo de funcionalidades más complejas como la generación de trayectorias, que requieren de un punto de referencia desde el que realizar los movimientos de un brazo robótico de forma controlada.

De esta forma, a partir de aquí solo resta añadir al programa tantas funcionalidades como se deseen. En este caso se le ha implementado un generador de trayectorias mediante cinemática inversa que permite al robot realizar movimientos de mayor complejidad. El desarrollo del proyecto

desemboca por tanto en un programa de control funcional para el brazo robótico Scorbot 4R-EU basado en Python y de ejecución multiplataforma, cuya estructura se resume en la Figura 2.4.

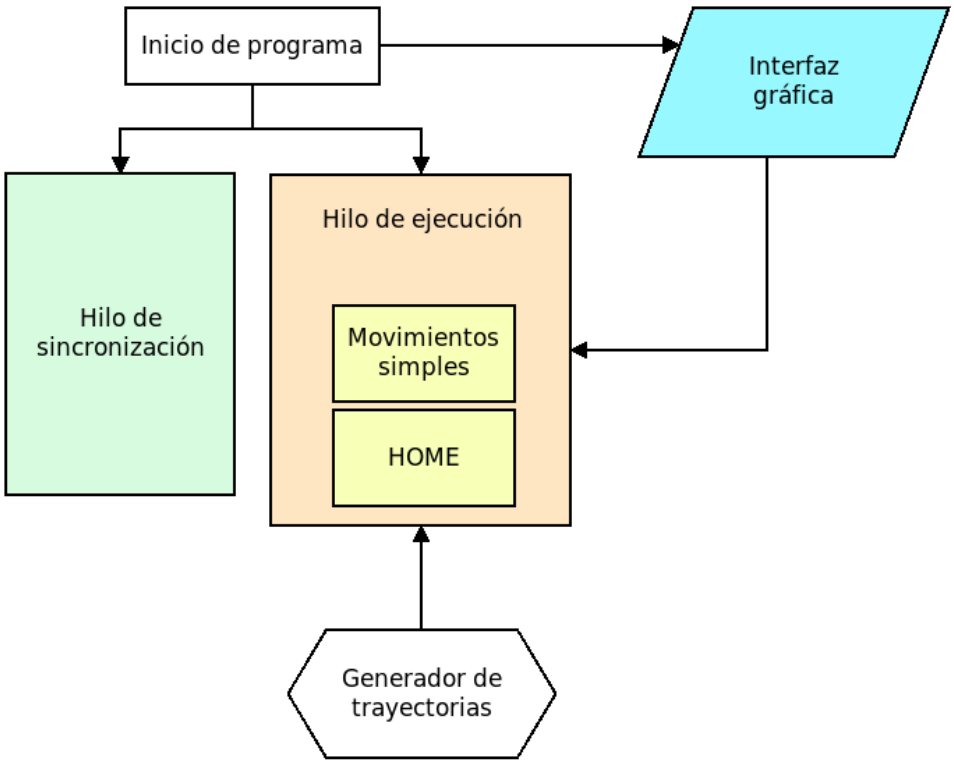


Figura 2.4: Estructura general del programa

Capítulo 3

Plataforma de desarrollo

Para el desarrollo de este proyecto, se ha tomado como referencia de equipo de trabajo al Scorbot ER-4U. Es un modelo de robot manipulador cuyo diseño está orientado al entorno académico. Prueba de ello es la carcasa de estilo abierto que presenta, permitiendo observar las distintas piezas mecánicas que componen el robot durante su funcionamiento.

La forma de control por defecto del robot es mediante el programa Scorbase, de Intelitek Inc. Este software está diseñado para funcionar sobre el sistema operativo Windows XP y permite realizar un uso básico del brazo robótico para introducir a los estudiantes en la robótica industrial. También cuenta con un modo Avanzado de uso que permite un mayor control y la realización de pre-configuraciones del robot.

El equipo se compone del brazo robótico Scorbot ER-4U y una controladora USB de la misma familia de robots Scorbot. Ambos equipos se comunican mediante un cable interface de 62 pines tipo-D, véase Figura 3.1. La instalación es completada por un ordenador conectado vía cable USB a la controladora USB [10].

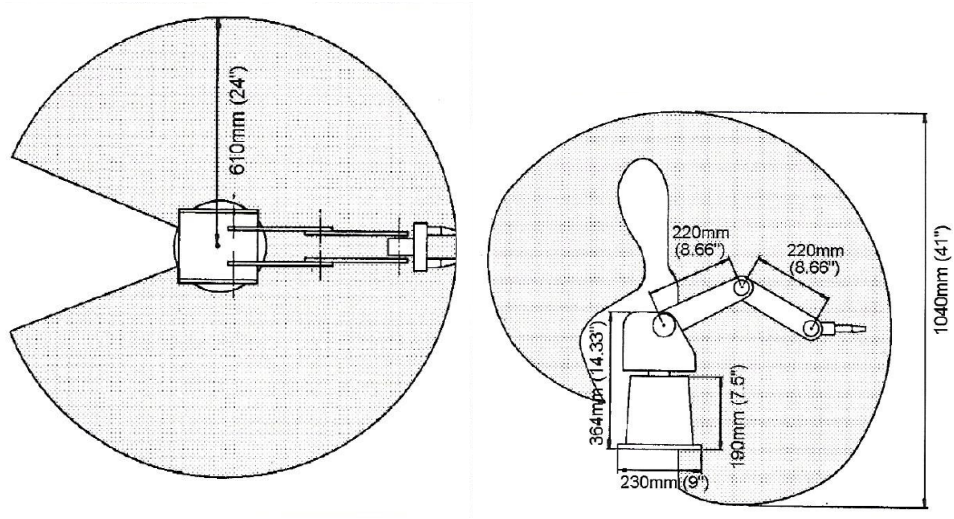
Figura 3.1: Conexión de los equipos

3.1. Estructura del robot

Se trata de un manipulador antropomórfico de 5 GDL (Grados de Libertad)[1] que está formado por 6 motores, todos de corriente continua y con transmisión mecánica mediante correas y engranajes. [10]

Dentro del mundo de la robótica de manipulación, se suele comparar la configuración de los movimientos de un robot con las articulaciones anatómicas, es por esto que haremos referencia a los diferentes *links* del robot como, articulaciones hombro (2), codo (3), muñeca (4 y 5) a excepción, a la cadera (1) se hará referencia con el nombre de base. Encontramos que las articulaciones son de tipo rotación y con 1 GDL. A excepción del resto, la muñeca es una articulación cilíndrica con 2 GDL. Cada una de ellas tiene lo que se conoce como una zona de trabajo, siendo esta el espacio físico que la mecánica del propio robot le permite alcanzar con cada movimiento. En las Figuras 3.2a y 3.2b se representan las zonas de trabajo del Scorbot 4R-EU que se complementan con la tabla 3.1, donde se recogen los límites mecánicos de cada movimiento.

[1] Se conoce como GDL a los movimientos que puede realizar una articulación respecto a la anterior de forma independiente.



(a) Área de trabajo de la base [10] (b) Área de trabajo hombro-codo [10]

Figura 3.2: Áreas de trabajo del manipulador

Eje	Tipo de movimiento	Ángulo
Eje 1	Rotación de la base	310º
Eje 2	Rotación del hombro	130º/-35º
Eje 3	Rotación del codo	± 130º
Eje 4	Inclinación de la pinza	± 130º
Eje 5	Giro de la pinza	± 570º

Tabla 3.1: Especificaciones de los movimientos

3.2. Motores

Las elementos motrices que componen el robot, ver Figura 3.3, son de tipo servomotor de corriente continua cuyo sentido del movimiento viene

dado por la polaridad de la tensión aplicada. Para el movimiento de cada articulación hay dedicado un motor, a excepción de la muñeca que, al ser capaz de realizar movimientos de *pitch* y *roll* sobre el eje de la propia articulación, requiere de dos motores dedicados (etiquetas 4 y 5 de la Figura 3.3). La pinza por su parte viene controlada por un servo motor (etiqueta 6 de la Figura 3.3) que le permite realizar los movimientos de apertura y cierre. La Tabla 3.2 complementa la Figura 3.3 asociando cada movimiento con su motor correspondiente.

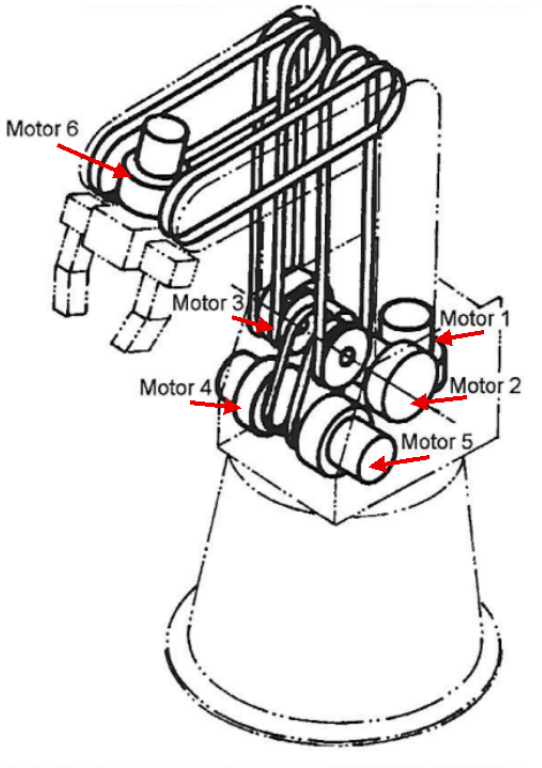


Figura 3.3: Distribución de los motores

Motor	Tipo de movimiento
Motor 1	Rotación de la base
Motor 2	Rotación del hombro
Motor 3	Rotación del codo
Motor 4	Movimiento de la muñeca
Motor 5	Movimiento de la muñeca
Motor 6	Accionamiento de la pinza

Tabla 3.2: Relación entre motor y movimiento asociado

3.3. Encoders

En el extremo de cada motor, tal y como se puede observar en la Figura 3.4, se halla un encoder de tipo incremental óptico cuya finalidad es llevar un control cerrado de los movimientos del motor en cuestión.

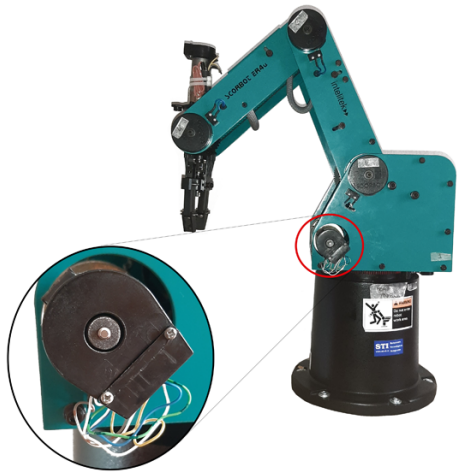
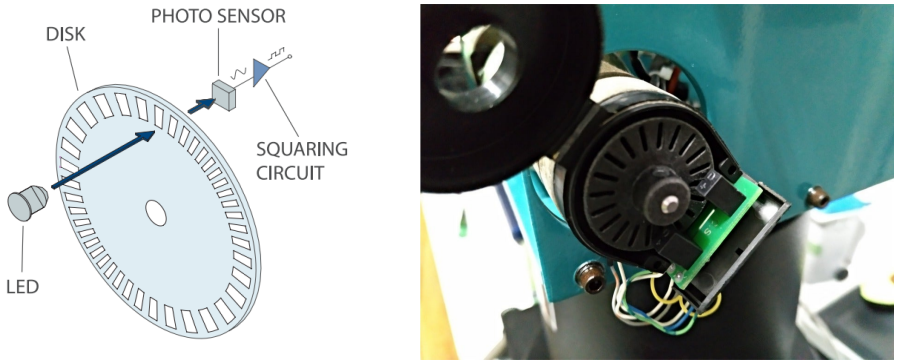


Figura 3.4: Detalle de un encoder

Este dispositivo nos permite calcular el ángulo, velocidad y aceleración del motor al que está asociado mediante la cantidad de impulsos que recoge el foto-transistor. En las Figuras 3.5a y 3.5b, se observa con detalle la estructura que conforma a los enconders del robot.



(a) Interior de un encoder óptico (b) Encoder óptico del Scorbot ER-4U

Figura 3.5: Estructura interna de un encoder óptico

3.4. Micro-interruptores

El Scorbot ER-4U dispone de cinco micro-interruptores, uno sobre cada eje. Estos serán un elemento clave para realizar determinadas funciones, la función *home* entre ellas, que se detalla en el capítulo 8. En la Figura 3.6 se define la situación de cada micro-interruptor, donde el 1 se corresponde con la base, el 2 con el hombro, el 3 con el codo, el 4 se corresponde con el movimiento de *pitch* y el 5 con el movimiento de *roll*:

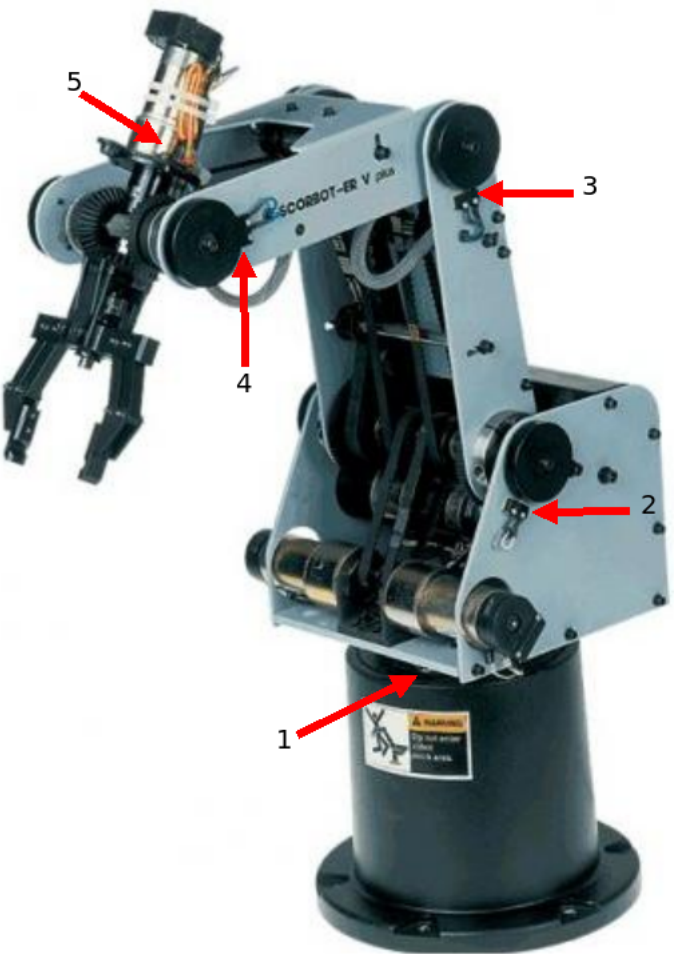


Figura 3.6: Situación de los micro-interruptores

El funcionamiento de estos elementos consiste básicamente en la activación por parte de la articulación del botón que compone al micro-interruptor. Tal como puede apreciarse en la Figura 3.7, el eje está dotado con una pestaña que en una determinada posición activa el sensor.

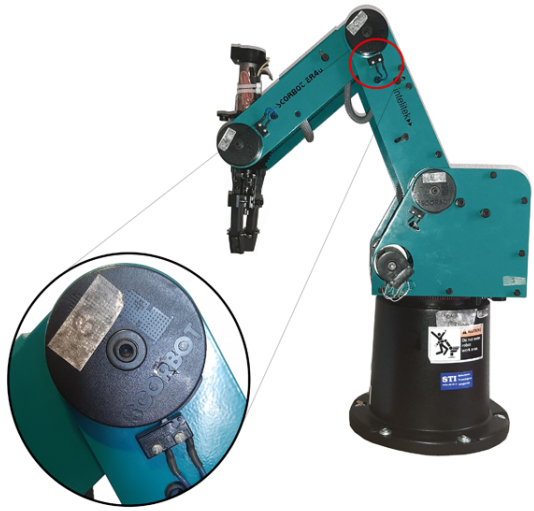
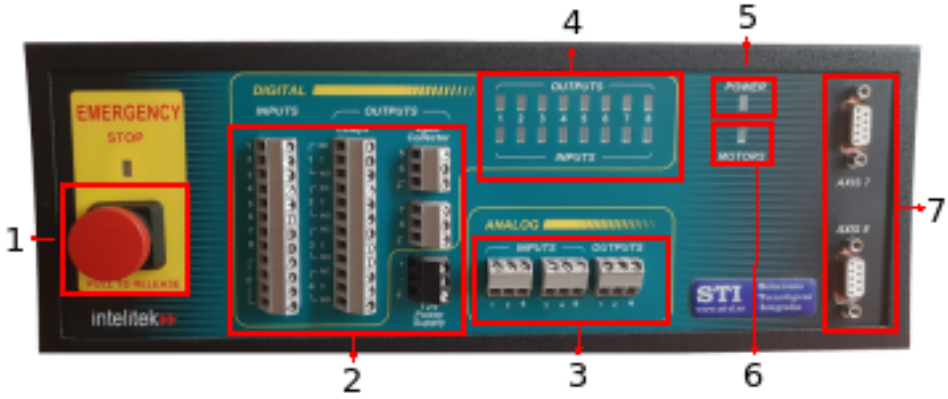


Figura 3.7: Detalle de un micro-interruptor

3.5. Controladora USB

La controladora USB es el elemento que rige la comunicación entre el robot y el ordenador, así como el control con unidades periféricas adicionales al sistema clásico del robot. La conexión física entre el robot y la controladora se realiza mediante un cable de 62-pines tipo-D de alta densidad(conector 1 de la imagen de la Figura 3.8b), y el ordenador se enlaza con la controladora con un cable USB(conector 2 de la imagen de la Figura 3.8b).



(a) Vista delantera

(b) Vista trasera

Figura 3.8: Controladora USB

Este instrumento es capaz de controlar los motores del robot mediante señal PWM y leer las señales de los encoders y micro-interruptores. Además también es capaz de ampliar su funcionalidad haciendo uso de las entradas/salidas analógicas y digitales, conectores 3 y 2 respectivamente de la imagen de la Figura 3.8a, para añadir sensores o actuadores adicionales

o hacer uso de los conectores que permiten la conexión de dispositivos periféricos motorizados de SCORBOT mediante los conectores del elemento 7 de la imagen de la Figura 3.8a.

En el panel frontal de la controladora USB se dispone de una serie de indicadores LED, de los cuales el principal y de mayor interés en la primera fase del proyecto es el led *POWER*(número 5 en la imagen de la Figura 3.8a). Este led indica al usuario si hay una conexión establecida entre el programa y la controladora USB. Por otro lado tenemos el LED de motores, elemento 6 de la imagen de la Figura 3.8a, indica si se está dando tensión o no a los motores. Este se activa por defecto cuando se inicia por primera vez el programa o al habilitar los motores desde la interfaz de usuario. Se apagará siempre que se presione la parada de emergencia(número 1 de la imagen de la Figura 3.8a), se deshabiliten desde programa o se cierre la aplicación. Por último, la controladora USB dispone de un conjunto de 16 LEDs, número 4 de la imagen de la Figura 3.8a, para indicar el estado de las 8 entradas y las 8 salidas digitales.

3.6. Software de control. Scorbase 5.6

El programa Scorbase versión 5.6 es el software de control por defecto para el uso del Scorbot 4R-EU. Para su utilización es necesario el montaje de un ordenador con sistema operativo Windows XP/Vista/7. Es por ello que en la actualidad se ha quedado obsoleto este software.

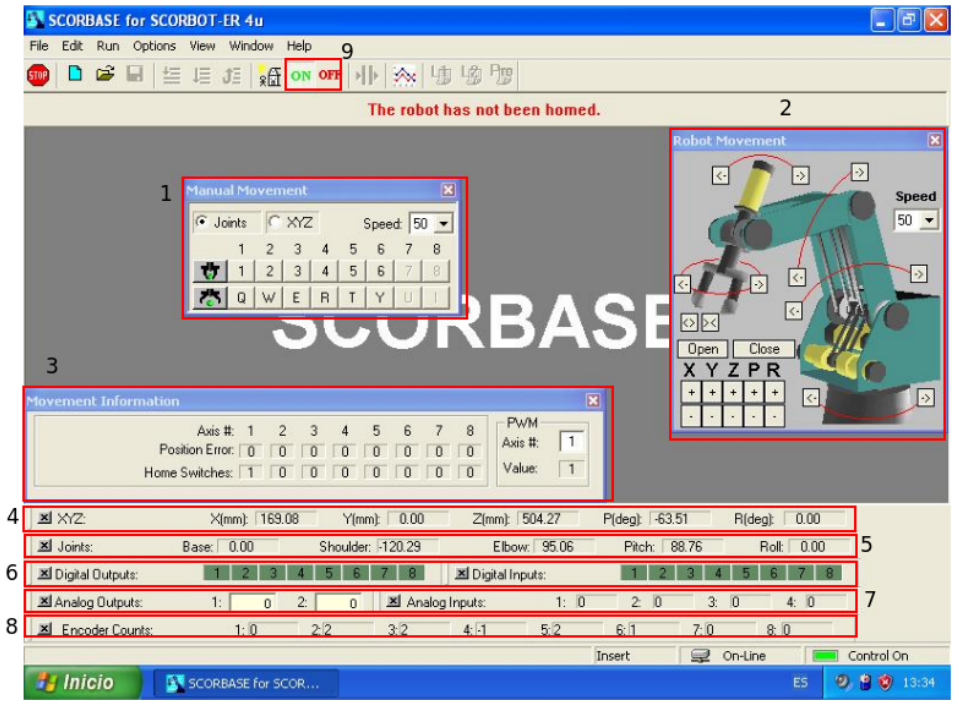


Figura 3.9: Vista de la interfaz gráfica de Scorbase 5.6

En la imagen de la Figura 3.9, se pueden apreciar las diferentes prestaciones que ofrece el paquete de control de Scorbase. Entre ellas, haremos una breve descripción de las más destacadas:

– **Search Home:** Ofrece el procedimiento hacia la posición predefinida como *home*, donde cada eje transita hasta su posición de inicio por separado. Esta posición se consigue cuando los todos los microinterruptores están activos.

– **Manual Movement:** Permite controlar el robot de manera directa. Cuando está seleccionado el modo *joints*, al hacer *click* o presionar

las teclas correspondientes del teclado, se mueve un eje del robot. En el modo *XYZ*, presionar un botón se traduce en el movimiento del robot a lo largo del eje correspondiente. Elemento 1 de la imagen de la Figura 3.9.

- **Robot Movement:** Permite llevar un control en los modos *XYZ* y *joints*, al igual que el *Manual Movement* pero exclusivamente mediante pulsaciones en los botones del cuadro de diálogo. Elemento 2 de la imagen de la Figura 3.9.

- **Status Bars:** Muestran la posición del efector final en coordenadas cartesianas, el ángulo de cada articulación, el estado de las salidas y entradas digitales y analógicas y el valor de lectura de los encoders. Elementos 4, 5, 6, ,7 y 8 de la imagen de la Figura 3.9.

- **Movement Information:** Indica en caso de error o pulsación de micro-interruptor, el eje al que corresponde dicha señal. Elemento 3 de la imagen de la Figura 3.9

- **ON/OFF:** Activa o desactiva el servocontrol de los ejes. *Control On* solo está disponible cuando se establezca la conexión con la controladora USB y el *Control Off* no permitirá la ejecución de comandos de movimiento. Elemento 9 de la imagen de la Figura 3.9.

Capítulo 4

Adquisición y análisis de datos

Parte esencial para el desarrollo de un proyecto de estas características es situarlo dentro de un marco teórico a partir del cúal comenzar a trabajar. En este caso se establece dentro de la comunicación USB entre el Scorbot EU-4R y el programa Scorbase.

Para la adquisición de datos se hace uso del programa Wireshark, ejecutado sobre linux, en paralelo a la ejecución del programa Scorbase, lanzado en una máquina virtual con sistema operativo Windows XP. El resultado de la captura de datos, es un archivo CSV con toda la información que se ha detectado en la comunicación USB, clasificada por orden de llegada. En el Anexo F se encuentra en detalle los componentes básicos de dicha comunicación, los cuales nos permiten establecer el origen y destino de los dispositivos, entre otras especificaciones. La información de los paquetes de datos al ser capturada se organiza de la siguiente manera:

- **Sequence**: Indica el orden en el que han sido capturados los mensajes.

- **Type**: Indica el tipo del paquete capturado. Puede ser un paquete de control, uno de escritura, lectura...

- **Time**: Indica el tiempo que ha transcurrido desde que se inició la captura hasta que se capturó el mensaje.
- **Request**: Indica el tipo de transferencia que se ha realizado para enviar o recibir el paquete.
- **I/O**: Indica el origen del paquete capturado.
- **EP**: Indica el endpoint de origen del paquete.
- **IRP**: Es un paquete conocido como I/O Request Packets.
- **Status**: Indica el estado del IRP.
- **Data**: Información de comunicación que puede ser una orden o una respuesta a una orden. No todos los paquetes llevarán información de este tipo asociada.
- **Length**: Indica el tamaño del paquete Data del mensaje.

La columna **data** contiene todas las órdenes enviadas desde el programa hacia la controladora y las respuestas que ésta última devuelve al *host*. Mediante la información del *endpoint* se puede determinar el origen y el destino.

La longitud de cada elemento **data** es de 128 bytes codificados en hexadecimal y agrupados por pares que se pueden separar en distintos grupos según analicemos los mensajes de escritura o de lectura. A la hora de analizar la información, es importante destacar que ésta se ordena de forma distinta, dentro del mensaje, según su origen.

Para facilitar el análisis, la información es filtrada con la finalidad de eliminar aquella que sea irrelevante en esta parte del proceso. Se hace uso del filtro *filter_SCB.py*, ver Anexo C, que separa los mensajes de escritura de los de lectura observando la columna *endpoint* del fichero original.

Para el análisis de los paquetes necesarios para controlar los movimientos de cada articulación, las capturas de datos se clasifican por motor, es decir, en cada captura de datos solo se realizan movimientos con una sola articulación desde el Scorbase para tener información específica de cada una.

Como resultado de este proceso, se obtienen numerosos documentos con miles de paquetes cuya información relevante viene codificada en hexadecimal y expresada en una misma sección del mensaje: **data**. Las herramientas de análisis se centran en el desglose de esta columna del documento.

Capítulo 5

Tipos de mensajes

El control del brazo robótico es posible gracias a la información que es intercambiada entre el programa y la controladora USB. Desde el punto de vista del programa, los mensajes que son enviados desde el propio programa son mensajes de escritura mientras que los mensajes que son enviados al programa son de lectura. En ambos casos, la información viene estructurada dentro de cada mensaje de una forma determinada y única para cada tipo. La estructura y análisis de estos paquetes se recoge en el capítulo 4.

5.1. Mensajes de escritura

El componente **data** de los mensajes de escritura se compone de tres grupos:

- **Grupo 1**: Primer doble byte del mensaje de escritura. Consiste en un par de bytes que se usan como contador para llevar un control sobre la concatenación de mensajes. Este contador llega al valor máximo de FF, con el doble byte hexadecimal. Con cada mensaje, se incrementa en uno su valor y cuando alcanza su valor máximo, el

contador se reinicia a 1, permitiendo tener una referencia cíclica de los datos.

– **Grupo 2**: Del segundo doble byte al duodécimo. Orden a cumplir por la controladora. En el grupo 2, el par de bytes más importante es el correspondiente al de la quinta posición. En ellos se indica qué orden es la que se envía y, por tanto, cuál es la acción a llevar a cabo. El listado de órdenes identificadas e implementadas en el programa resultante de este proyecto están desglosados en la Tabla E.1 del Anexo E. El formato de este grupo cambia según la orden que se esté transmitiendo.

– **Grupo 3**: Del decimotercer doble byte al cuadragésimo cuarto. Valor de la escritura para el encoder asociado a cada motor correspondiente a la posición del mismo. A su vez, los datos de cada motor se dividen en conjuntos de cuatro dobles bytes, dándonos un resultado de 8 secciones: base, hombro, codo, motor 1 de la muñeca, motor 2 de la muñeca, pinza y dos adicionales que pueden ser conectados a la controladora como actuadores periféricos del robot. Cada sección define el estado de un motor dando la posición del encoder con los dos primeros dobles bytes de la sección y el signo con los dos segundos. La lógica de estos datos es explicada en la sección 7.2.

Los bytes restantes no han entrado dentro de los objetivos de este proyecto al corresponderse con las entradas y salidas analógicas y digitales de la controladora.

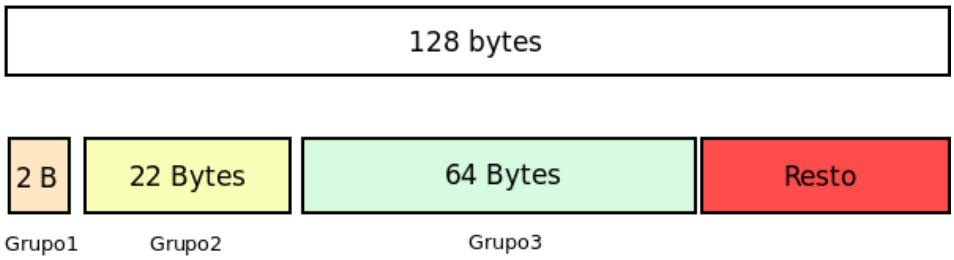


Figura 5.1: Distribución de la información en mensajes de escritura

5.2. Mensajes de lectura

El componente **data** de los mensajes de lectura se compone de cuatro grupos:

- **Grupo 1**: Primer doble byte. Secuencia que indica el mensaje de escritura al que está respondiendo la controladora. Sigue la misma lógica que el grupo 1 de los mensajes de escritura, ya que los primeros son en respuesta a los segundos. Destacar que no todos los mensajes de escritura tienen un mensaje de lectura asociado, solo aquellos cuya respuesta es necesaria para continuar la comunicación con el programa.

- **Grupo 2**: Del segundo doble byte al sexto. Indica la orden a la que está respondiendo el mensaje de lectura. Replica el quinto doble byte del mensaje de escritura al que se está dando respuesta, ya que ese es el doble byte que indica qué orden se está realizando.

- **Grupo 3**: Séptimo doble byte. Indica qué micro-interruptores están activos en el momento en el que se creó el mensaje. Los valores asociados a cada micro-interruptor están desglosados en la Tabla 8.1 de la sección 8.2. Tiene la complejidad de ser el resultado de la suma

hexadecimal de todos los micro interruptores activos en el momento de la lectura, por lo que se deberá tener en cuenta a la hora de determinar qué articulaciones están en su posición de *home*.

– **Grupo 4**: Del vigésimo doble byte al quincuagésimo noveno. Lecturas de los encoders de cada motor que tendrán asignados 5 pares de bytes cada uno. La estructura que rige la información de los motores es igual a los mensajes de escritura: base, hombro, codo, motor 1 de la muñeca, motor 2 de la muñeca, pinza y los dos motores periféricos adicionales que se pueden añadir a la controladora. Dentro de la información de cada motor se tiene que los dos primeros pares de bytes se corresponden con la posición del motor, el tercero se corresponde con el signo del valor de la posición y los dos últimos son bytes de error que reflejan el desfase de la posición del motor entre la secuencia de lectura y la de escritura asociada. Estos son empleados para el control de errores durante la realización de movimientos. En caso de que el *encoder* no detecte el incremento indicado en la escritura anterior, la controladora transmite el desfase entre posición objetivo y posición alcanzada en estos dobles bytes. Esta característica permite realizar un control más exhaustivo de los movimientos del robot y de los errores.

Los bytes restantes no han entrado dentro de los objetivos de este proyecto al corresponderse con las entradas y salidas analógicas/digitales de la controladora.

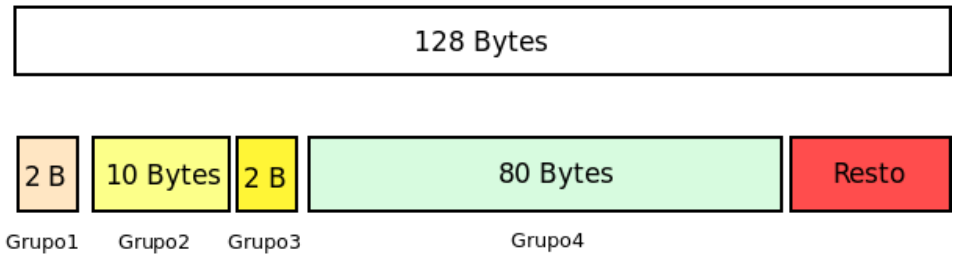


Figura 5.2: Distribución de la información en mensajes de lectura

Capítulo 6

Comunicación: Controladora USB

Siguiendo la estructura de programa del diagrama de la Figura 2.4, la comunicación entre la aplicación y la controladora USB puede dividirse entre los mensajes necesarios para realizar la conexión con la controladora USB, los mensajes enviados por el hilo de sincronismo y los mensajes enviados por el hilo de ejecución.

6.1. Conexión a la controladora USB

La comunicación entre el programa y la controladora USB comienza con el proceso de enumeración, descrito en el Anexo F. Durante este proceso se recopila la información necesaria para crear una vía de acceso a la controladora USB que permita al programa conectarse. La clave para llevar acabo esta conexión es conocer de antemano el *idVendor* y el *idProduct* del dispositivo al que se quiere conectar el programa, ya que estos son los identificadores dentro del *hub* que diferencia un equipo de otro. En el caso de que hubiese más de un dispositivo con los mismos identificadores, habría

que realizar un proceso de identificación más exhaustivo. Ahora el acceso al punto de conexión de la aplicación a la controladora está controlado por el propio programa.

Una vez se establece la configuración que debe seguir el *firmware* de la controladora USB y se han creado los objetos en el programa referente a los *endpoints* de entrada y salida de datos, se realiza un segundo análisis de la comunicación entre el equipo y el Scorbase. El objetivo es encontrar un patrón que se asocie con el momento de encendido de la conexión entre el Scorbase y el robot siguiendo los métodos especificados en el Capítulo 4. El resultado es un conjunto de secuencias de datos que se dividen en tres grupos de paquetes de datos, los dos primeros se corresponden con secuencias predeterminadas y específicas del encendido del robot, mientras que el tercero destaca por su posterior relevancia en apartados más avanzados del proyecto al realizar el encendido de los motores mediante órdenes de control a través de la interfaz gráfica, que se muestra en el Capítulo 10 de la memoria.

Llegados a este punto, el programa ya es capaz de realizar, lo que llamaremos de aquí en adelante, una secuencia de encendido pero no es capaz de mantener una comunicación abierta con la controladora que mantenga ambos extremos de la conexión sincronizados.

6.2. Hilo de sincronismo

Este primer hilo de los dos que componen el programa se considera como la base, ya que es el encargado de mantener activa la comunicación con la controladora USB una vez se ha realizado la conexión inicial y mientras no se está realizando ninguna otra acción.

La funcionalidad del hilo de sincronismo se basa en la ejecución cíclica de una función cuyo objetivo es el envío periódico de un mensaje de sincronismo, cuya estructura viene definida en el diagrama de la Figura 6.1.

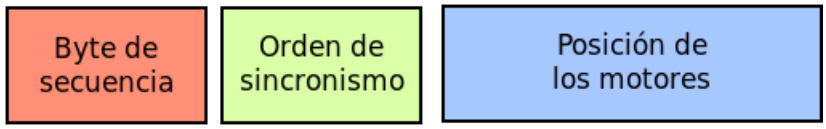


Figura 6.1: Mensaje de sincronismo

Por un lado, la **orden de sincronismo** es una secuencia de bytes constante que indica que el robot no está realizando ninguna acción. Por otro lado, el segmento de mensaje correspondiente a la **posición de los motores** se corresponde con el grupo 3 de los mensajes de escritura, definido en el capítulo 5, y que es reflejo de la lectura de datos de los encoders del robot. Lo verdaderamente importante de este tipo de mensajes viene recogido en el **byte de secuencia**. El byte de secuencia es una palabra de dos bytes que actúa como contador cíclico en el rango de 1 a 255 en base decimal y que se corresponde con el grupo 1 de los mensajes de escritura definido en el Capítulo 5. Mientras el controlador USB reciba de forma periódica la concatenación de valores que le corresponde a este contador, la conexión entre programa y dispositivo se mantendrá activa.

6.3. Hilo de ejecución

La interacción entre usuario y programa se considera puntual, si tenemos en cuenta el volumen de mensajes que manda el programa a la controladora USB en los periodos de tiempo en los que el usuario no indica ninguna acción a llevar acabo. Es por esto que se crea un hilo independiente al de sincronismo con el fin de atender las peticiones del usuario.

El hilo de ejecución está orientado a gestionar la tarea que debe llevar acabo el programa ante las distintas posibles órdenes que puede dar el usuario. El listado de órdenes disponibles se recogen en la Tabla E.1 del Anexo E y se resumen como todo aquel mensaje creado para generar movimiento en alguna de las articulaciones del robot o para llevar acabo una

secuencia distinta a la de sincronismo. Mientras el hilo de ejecución esté realizando un envío de mensajes, el hilo de sincronismo se mantiene a la espera y es el propio hilo de ejecución el encargado de mantener la sincronización durante el espacio de tiempo que le lleve terminar su función. Para ello, los mensajes que envíe deben cumplir la estructura del diagrama de la Figura 6.2.

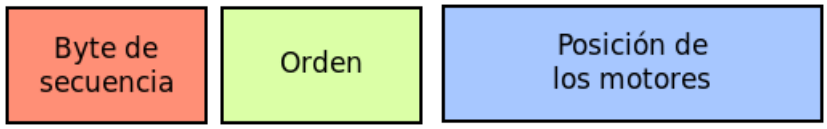


Figura 6.2: Mensaje de ejecución

A diferencia de los mensajes de sincronismo, el segmento del mensaje correspondiente a la **posición de los motores** no siempre es estrictamente reflejo de la posición actual de los motores; dependerá de si el programa debe ejecutar una acción que implique el movimiento del robot o ejecutar una orden de control. En cualquier caso, el segmento correspondiente a **orden** dependerá de la petición realizada al programa por el usuario. Por último, aparece nuevamente el **byte de secuencia** mencionado en la sección 6.2 cuya importancia sigue siendo la misma y debe cumplir las mismas reglas que las expuestas en la sección anterior.

Capítulo 7

Movimientos simples

El uso seguro del brazo robótico es uno de los objetivos principales a la hora de desarrollar un software que lo controle. No solo debe ser seguro para el usuario, sino también brindar cierto nivel de certeza de que el equipo no sufrirá daños durante su uso. Es por ello que se debe prestar atención a cumplir estos requisitos desde el nivel más bajo del programa, en el que se establece la conexión con la controladora USB, e ir extendiendo esa seguridad al resto de capas, que se añaden en forma de herramientas adicionales.

Con la implementación del hilo de ejecución, vista en el Capítulo 6, el programa dispone de una base sobre la que poder ir añadiendo funcionalidades que permita al usuario tener un control sobre el manipulador. La primera herramienta de todas es la que permite realizar movimientos controlados de cada una de las articulaciones. La ventaja que presenta desarrollar esta herramienta en primer lugar es que todas las articulaciones siguen la misma lógica a la hora de implementarla y establecerá las bases para futuras funcionalidades.

7.1. Desarrollo del movimiento

La idea consiste en que, tal como se refleja en el diagrama de la Figura 7.1, una vez conocido el ángulo que debe moverse la articulación y a qué velocidad debe realizar el movimiento, el programa deberá ser capaz de generar los mensajes necesarios para completar la orden. Dicha orden se capta mediante el hilo de ejecución, y será esta quien indique qué movimiento se realizará y por consecuencia, qué sección concreta de grupo 3 de mensaje de escritura, descrito en el Capítulo 5, debe ser modificada.

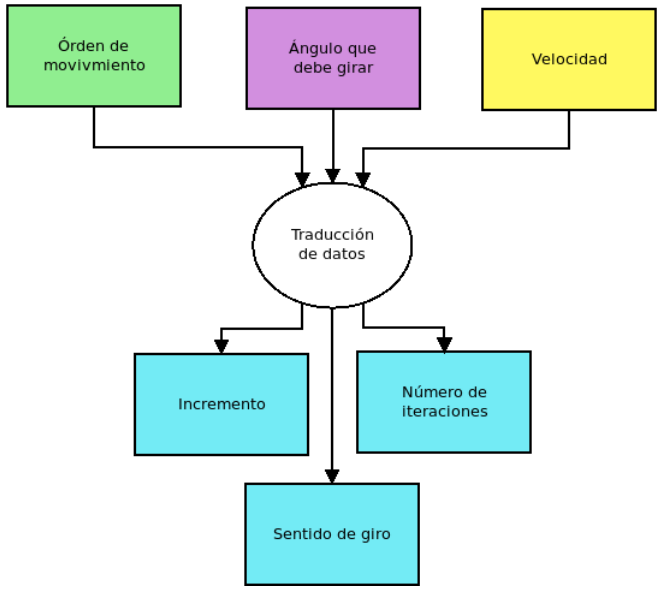


Figura 7.1: Traducción de los parámetros de un movimiento simple

Ahora es necesario ir indicando con cada mensaje asociado al movimiento del motor, cuánto a de incrementarse la posición del mismo. Para ello debemos tener presente que los datos asociados a cada motor dentro del mensaje de escritura están ordenados como indica la imagen de la Fi-

gura 7.2. Los mensajes que definen el movimiento que debe llevar acabo la articulación irán modificando estos cuatro dobles bytes, tal como se indica en la sección 7.2, teniendo como resultado una evolución de la posición tal como se refleja en la imagen de la Figura 7.3 donde se toma como ejemplo parte del segmento de **posición de los motores** del mensaje de escritura del diagrama de la Figura 6.2 para aumentar la posición de la articulación de la base de 12337 a 12357.

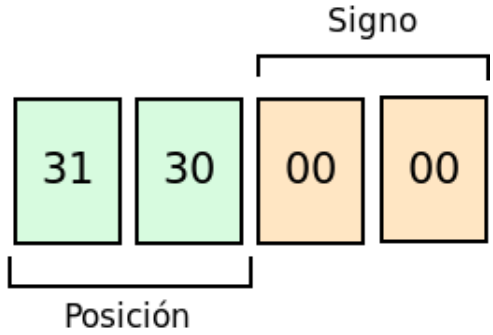


Figura 7.2: Bytes de escritura asociados a un motor

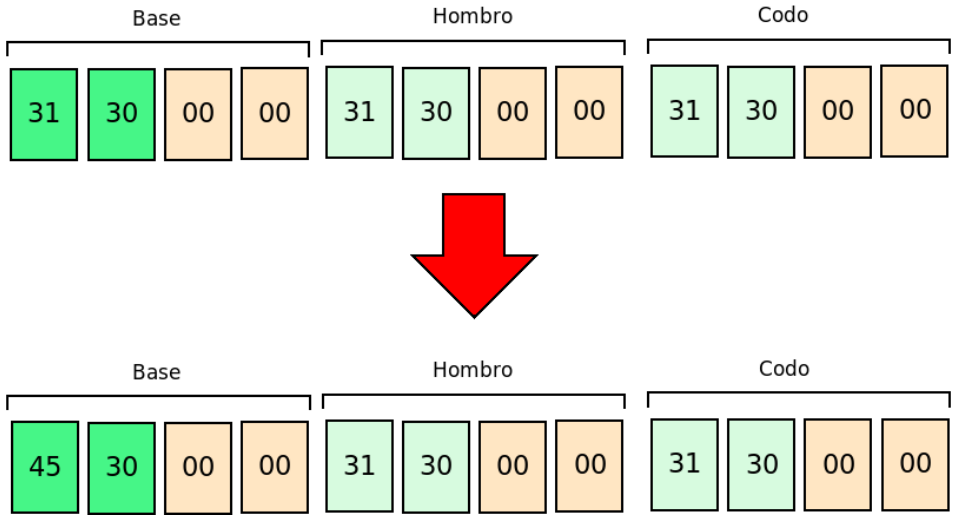


Figura 7.3: Incremento simple de la posición

Una vez finalizado el envío de los mensajes asociados al movimiento y estabilizada la lectura de los encoders, se considera que la orden ha finalizado y puede cerrarse el proceso de ejecución de movimiento. Una visión global de la lógica que sigue el proceso de ejecución de una orden de movimiento simple de cualquier articulación se puede resumir con el diagrama de la Figura 7.4.

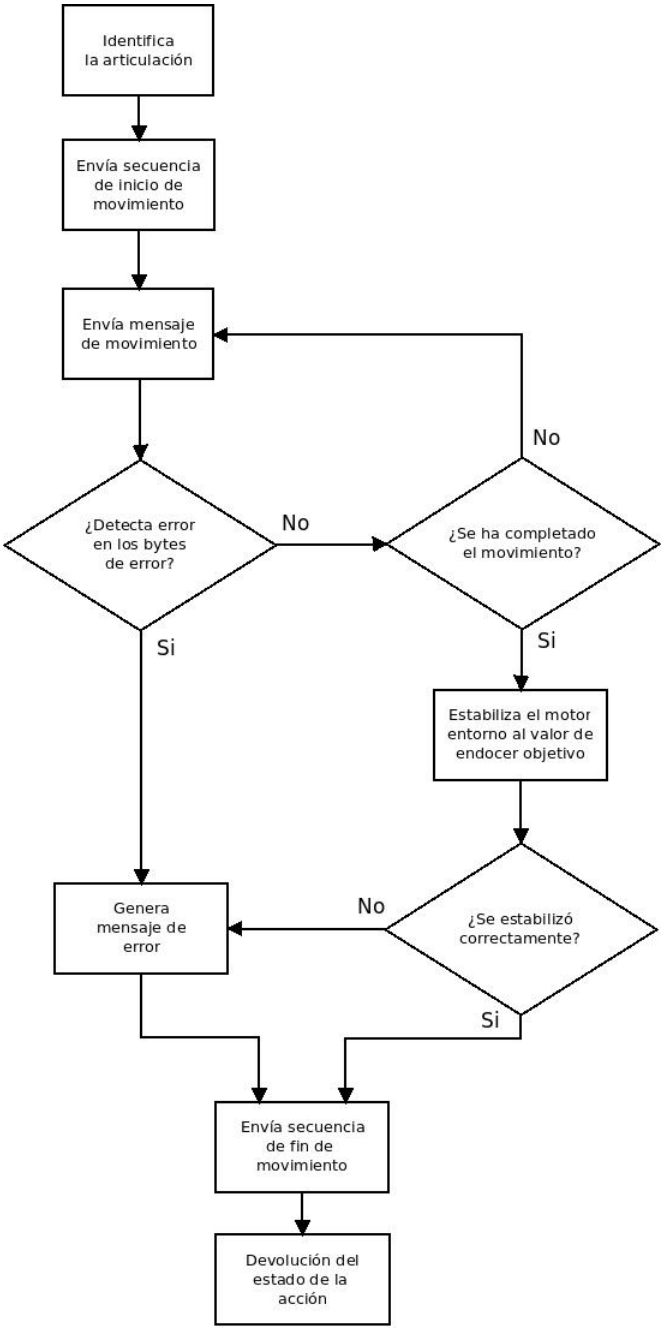


Figura 7.4: Lógica de un movimiento simple

7.2. Datos de los encoders

Independientemente del tipo de mensaje, descritos en el Capítulo 5, los dobles bytes con la información de la posición de los motores siguen la misma lógica. La diferencia reside en que en los mensajes de lectura se tiene información sobre la posición actual del *encoder* y en los mensajes de escritura se tiene la posición que debe alcanzar.

Analizando los dos dobles bytes asociados a la posición, se observa que siguen una progresión lineal directamente proporcional a la velocidad y al tiempo que dura el movimiento. Se observa que forman dos contadores con rango 0 a 255 en el que cada vez que el primero llega a su máximo, se incrementa en uno el valor del segundo. Al llegar ambos a su valor máximo, se reinicia a 0. Importante destacar que estos contadores son bidireccionales por lo que el aumento o la disminución de sus valores dependerá del sentido de giro del propio *encoder*. Para facilitar el manejo de estos contadores a lo largo del programa, se ha formado una palabra de cuatro bytes en el que el primer contador ocupa la mitad inferior de la palabra y el segundo la mitad superior, generando así un contador único de rango 0 a 65535 en base decimal(0000 a FFFF en base hexadecimal). Esto se ve reflejado en la Figura 7.5.

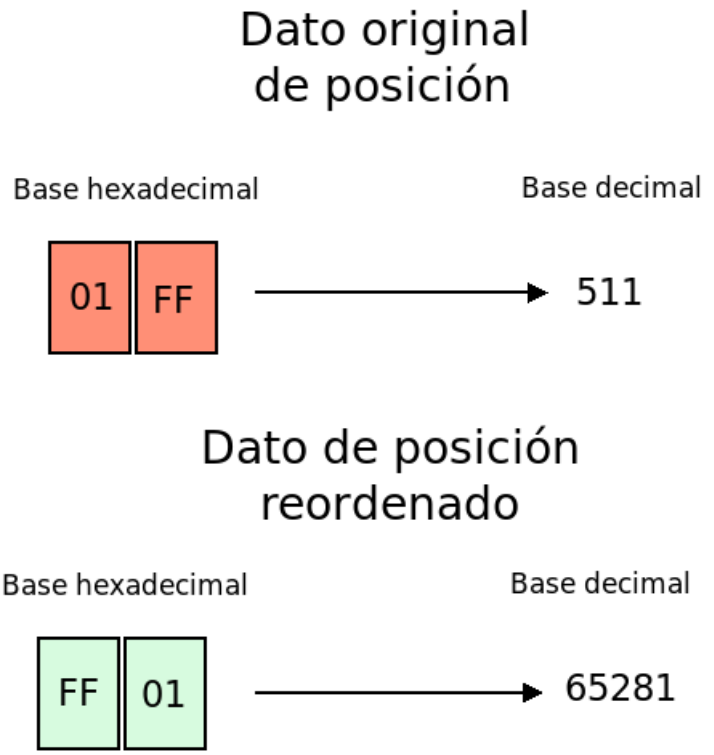


Figura 7.5: Reordenación e interpretación de los valores de encoder

Este rango se asocia al número de vueltas de encoder que provoca el motor al girar y que se traduce en posiciones de las articulaciones. No obstante, se debe tener en cuenta que la capacidad de giro de un motor debe estar limitado por los límites mecánicos de cada articulación. Es por ello que este rango de valores proporcionados por los encoders se implementa dividiéndolo en tres zonas diferenciadas:

– **0 - límite inferior**: Zona inferior cuyo límite superior local se corresponde con el límite inferior global. Tiene un valor por defecto de 20000.

– **límite inferior - límite superior**: Rango de valores que el *encoder* nunca alcanza por corresponderse con los límites del área de trabajo. A esta zona se le hace referencia con el nombre de zona muerta.

– **límite superior - 65535**: Zona superior cuyo límite inferior local se corresponde con el límite superior global. Tiene un valor por defecto de 50000.

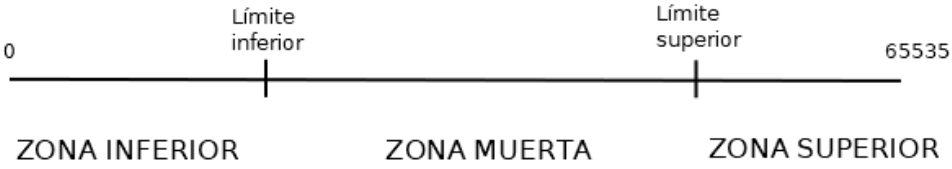


Figura 7.6: Distribución de valores de encoder por zonas

El siguiente dato relevante de los bytes asociados a la información de los motores, es el doble byte/bytes de signo. Dada la naturaleza de giro continuo del encoder, se hace necesario destinar un espacio de memoria a almacenar si el valor del contador ha pasado por el máximo o el mínimo en algún momento durante el movimiento. En los mensajes de escritura, la transición del valor máximo al mínimo se refleja cambiando los dos segundos dobles bytes de *0000* a *FFFF*, mientras que para la transición del valor mínimo al máximo se refleja con el cambio opuesto. Por otra parte, para la lectura se usa un doble byte en el que la transición del valor máximo al mínimo se refleja cambiando el tercer doble byte de *80* a *7F* y viceversa para la transición opuesta.

7.2.1. Transformación ángulo/encoder

El incremento de la posición que debe llevar acabo un motor viene determinado por un parámetro dado en grados a través de la interfaz gráfica

de usuario. Para poder ejecutar el movimiento, es necesario realizar una traducción del dato facilitado a un valor entendible para el programa.

Se realiza una conversión de ángulos a valores de encoder mediante factores de conversión extraídos del análisis de datos, véase la Tabla 9.3, y confirmados mediante la herramientas implementadas en el Scorbase.

	Ángulo(grados)	Encoder
Base	20	2837
Hombro	20	2300
Codo	20	2252
Pitch	20	676
Roll	20	558

Tabla 7.1: Relación ángulo-encoder

Para llevar a cabo la transformación, se hace uso del Algoritmo 7.1 presente en el script *libdef.py*, el cual permite realizar la conversión en ambos sentidos,

```
def conversorAngEnc(arti, conv, value):
  if arti == 1:
    if conv == 0:
      x = (20*value)/2837
    elif conv == 1:
      x = (value*2837)/20
      if x < 0:
        x = 65535 - abs(x)
  elif arti == 2:
    if  conv == 0:
      x = (20*value)/2300
    elif conv == 1:
      x = (2300*value)/20
      if x < 0:
        x = 65535 - abs(x)
  elif arti == 3:
```

```
17      if conv == 0:
18        x = (20*value)/2252
19      elif conv == 1:
20        x = (2252*value)/20
21        if x < 0:
22          x = 65535 - abs(x)
23    elif arti == 4:
24      if conv == 0:
25        x = (20*value)/676
26      elif conv == 1:
27        x = (676*value)/20
28        if x < 0:
29          x = 65535 - abs(x)
30    elif arti == 5:
31      if conv == 0:
32        x = (20*value)/558
33      elif conv == 1:
34        x = (558*value)/20
35        if x < 0:
36          x = 65535 - abs(x)
37    return round(x,2)
38
```

Algoritmo 7.1: Transformación de valor a angular a número decimal y viceversa

donde *arti* es la variable que indica a que articulación pertenece el dato a transformar, ver Tabla 7.2, *conv* indica a que unidades hay que transformar(grados o valor de encoder) y *value* es el dato a transformar.

Valor	Articulación
1	Base
2	Hombro
3	Codo
4	Pitch
5	Roll

Tabla 7.2: Variable *arti* de la función *conversorAngEnc*

7.2.2. Evolución de la posición de un encoder

El cambio de valor de un encoder depende proporcionalmente de la velocidad a la que el usuario determine que debe moverse un motor. Es decir, el parámetro que el usuario introduce como velocidad es igual a la cantidad de posiciones de encoder que incrementará o decrementará el motor con cada mensaje que se envía para realizar un movimiento por unidad de tiempo.

$$posSiguiente = posActual \pm \Delta pos \tag{7.1}$$

$$\Delta pos = velocidad * \Delta tiempo \tag{7.2}$$

donde el *tiempo* para el programa es el número de veces que se ha mandado el mensaje para aumentar la posición.

No obstante, empezar y acabar un movimiento a la velocidad de incremento máximo causa problemas de respuesta del motor provocando, generalmente, el bloqueo del mismo. La forma de evitarlo es implementar una evolución progresiva del incremento y decremento del movimiento describiendo una función trapezoidal que viene definida por la siguiente función a trozos:

$$posSiguiente = posActual \pm \Delta pos \tag{7.3}$$

$$\Delta pos = \begin{cases} \frac{velocidad}{12} t & si \quad t < 12 \\ velocidad * \Delta t & si \quad 12 \leq t < (ite - 12) \\ \frac{velocidad}{12} (ite - t) & si \quad ite \geq (ite - 12) \end{cases} \tag{7.4}$$

donde t representa el tiempo, que para el programa es el número de veces que se ha mandado el mensaje para aumentar la posición e *ite* es el total de mensajes que se mandarán para completar el movimiento. En la imagen de la Figura 7.7 se puede observar el resultado de esta función trapezoidal donde se puede apreciar la aceleración previa al alcance de la velocidad máxima y la frenada controlada que se produce para finalizar el movimiento. En este caso, el signo se asocia al sentido de giro del motor, por lo que esta gráfica representa el giro de un motor en un sentido y en otro a una velocidad de 20 unidades.

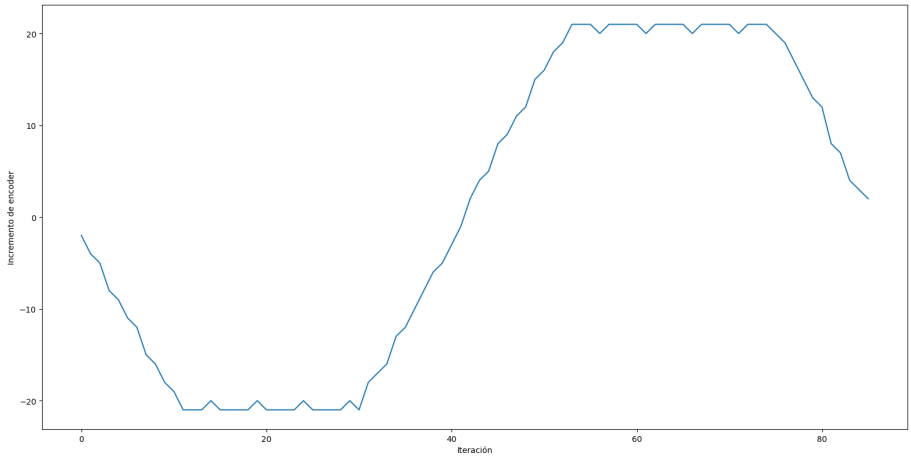


Figura 7.7: Evolución de la posición en función de la velocidad y el número de iteraciones

7.2.3. Control de la posición

El uso de motores de corriente continua con encoders acoplados provoca oscilaciones en las lecturas de las posiciones de las articulaciones. Esto es debido a la propia naturaleza del motor que, tras alcanzar una posición, debe esperar a que la señal de lectura se procese para saber si está en la posición correcta, tiempo durante el cual el motor sigue recibiendo corriente y, por ende, sigue girando. En procesos en los que se debe alcanzar un valor de encoder específico, esta variación provoca que el programa oscile entorno a dicho punto hasta que se estabilice.

Para solucionar o disminuir el efecto de esta situación, se ha implementado un lazo de control cerrado con el que se estabiliza entorno al punto objetivo la variación del valor del encoder de forma más eficiente.

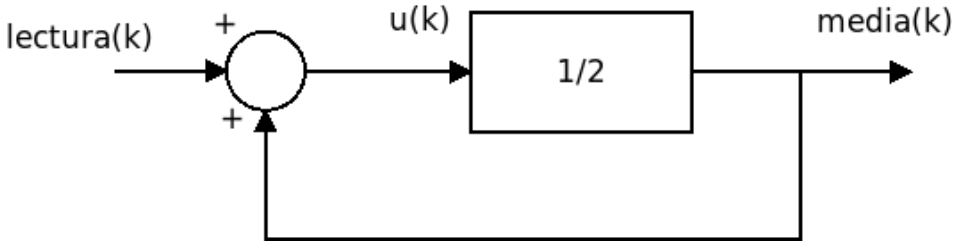


Figura 7.8: Lazo de control de la posición

donde k hace referencia al número de iteraciones.

Por si solo, este método de estabilización provoca situaciones de inestabilidad infinita que provocan el bloqueo de los motores en los casos en los que se pasa de un valor máximo de lectura a uno mínimo. Por ello, se ha implementado mediante algoritmo unas condiciones de control que los eviten.

```
def get_media(buffer, media):
  vec_pos = conf.readData("general","VEC_POS")
  for i in range(len(vec_pos)):
    dato = transform([buffer[vec_pos[i]], buffer[vec_pos[i
    ]+1]])
    if abs(dato - media[i] >= 1000):
      media[i] = dato
    else:
      dato_media = (media[i] + dato)/2
      media[i] = round(dato_media)

  return media

```

Algoritmo 7.2: Lazo de control

Tal y como se observa en el Algoritmo 7.2, la función *get_media*, que representa el lazo de control, se observa que en caso de que la diferencia entre la media y la nueva lectura supere un umbral determinado, el valor de la media pasa a ser el de la última lectura. Esto se hace para evitar que el resultado del cálculo de la media en una situación de este tipo de como resultado un incremento de posición desmesurado que provocaría el bloqueo de la articulación. De esta forma se evita que el motor intente alcanzar una posición mecánicamente inalcanzable y estabilice el valor de la media entorno a una posición más acorde a la realidad. En la imagen de la Figura 7.9 se puede ver como el paso en los valores de lectura de un máximo a un mínimo provocan que en la media aparezca un punto que diverge de la continuidad previa de la gráfica debido a ese cálculo de la media entre la lectura previa y la siguiente.

En la gráfica superior de la Figura 7.9 se puede apreciar como el programa ha indicado a el motor que alcance una posición inalcanzable, lo que ha provocado el bloqueo del mismo y que se puede comprobar observando la gráfica inferior de la imagen de la misma Figura donde las lecturas dejan de acompañar a la escritura.

A lo largo del programa se hace uso de la misma variable para almacenar el valor medio de las posiciones de los motores: el array *media*.

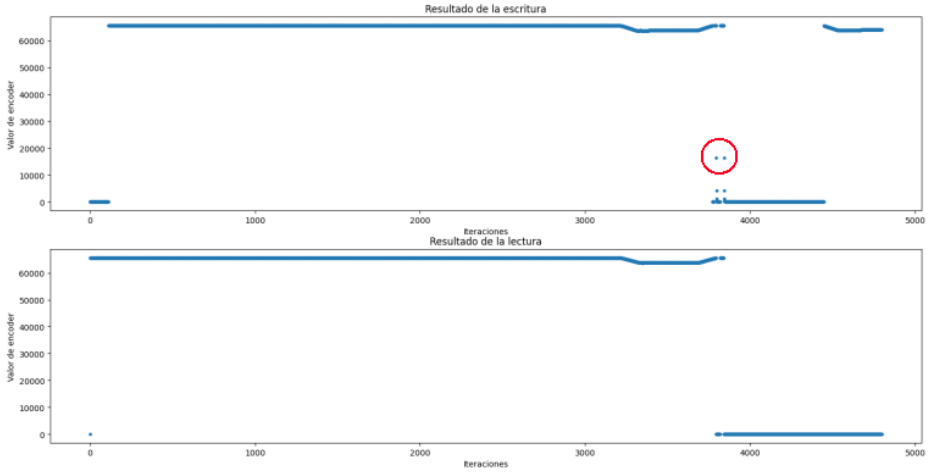


Figura 7.9: Bloqueo de la articulación por error en la media

Capítulo 8

Implementación del *home*

La ejecución de un movimiento complejo supone conocer en todo momento la posición de todas las articulaciones y el efector final del robot manipulador. En otras palabras, requiere de un sistema de referencia propio desde el que iniciar y concatenar sus movimientos.

8.1. Posición del *home*

La posición base o inicial del robot dentro de este sistema de referencia viene determinado por la posición del *home* y supone el punto de partida para la realización de movimientos complejos en la robótica de manipuladores. Se procede, por tanto, al desarrollo de una herramienta incluida dentro de la estructura del hilo de ejecución, el cual se introduce en el Capítulo 6, y que permita al programa dar las indicaciones necesarias a la controladora USB para que el robot manipulador alcance su posición de referencia.

Para el hilo de ejecución, esta acción se empieza a llevar acabo al detectar la orden de *home* [1] como petición del usuario. Una vez comienza el

[1] Puede verse la lista completa de órdenes en el anexo E

proceso, el programa va comprobando uno a uno si los micro-interruptores de cada articulación se encuentran activos o no siguiendo la lógica expuesta en la sección 8.2. En caso de que el micro-interruptor observado no se encuentre activo, se realizan envíos de mensajes hacia la controladora USB equivalentes al movimiento simple de la articulación en cuestión hasta que el programa detecte que el micro-interruptor se ha activado. En cambio, si el programa detecta que la articulación ya está en su posición *home*, pasa a evaluar la articulación siguiente siguiendo el orden especificado en el diagrama de la Figura 8.1.

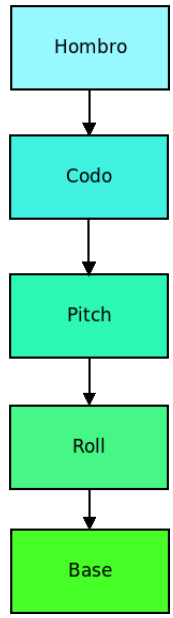


Figura 8.1: Orden para buscar el *home*

La similitud de funcionamiento entre la búsqueda del *home* y la realización de un movimiento simple, que se encuentra en el Capítulo 7, facilitan la construcción de esta herramienta, ya que , en resumidas cuentas, la bús-

queda del *home* es un conjunto de movimientos simples pre-establecidos a velocidad predeterminada que se ejecutan en caso de no detectarse el micro-interruptor de interés o hasta que éste se active. En el diagrama de la Figura 8.2 se resume el proceso descrito en esta sección.

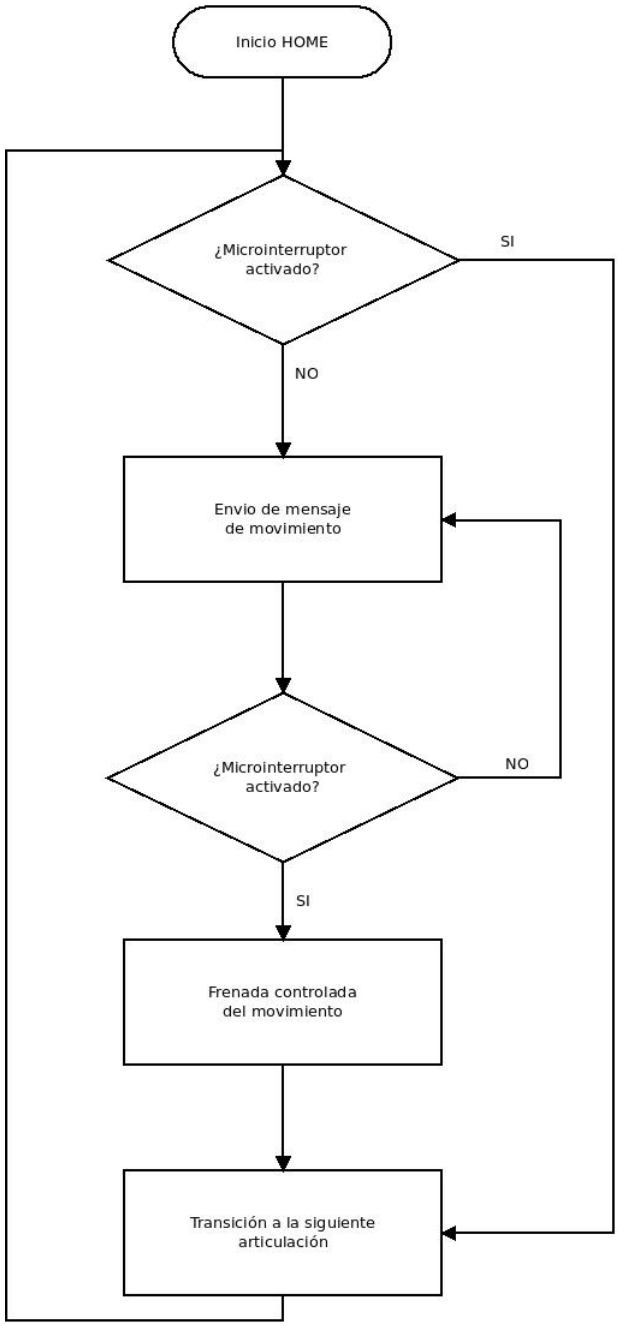


Figura 8.2: Diagrama funcional del *home*

8.2. Lectura de micro-interruptores

Dentro de la estructura de los mensajes de lectura expuesto en el capítulo 5, la imagen de la Figura 5.2 divide en grupos la información que estos mensajes contienen. En esta sección se quiere destacar el **grupo 3** referente al estado de los micro-interruptores presentes en cada articulación del robot. Este **grupo 3** se define como posibles valores que pueden aparecer en el sexto byte de los datos de lectura. La activación de más de un micro-interruptor a la vez queda reflejado en este segmento como un único dígito, de forma que si, por ejemplo, se activa el micro-interruptor de la base y el del codo el valor del byte sería de *05* en base hexadecimal [8].

Para identificar qué micro-interruptor está activo en cada momento, se hace uso del Algoritmo 8.1 dentro del script *libdef.py*, que diferencia cuáles están activos aprovechando la relación matemática existente entre los códigos: la suma de los códigos anteriores es igual al siguiente código menos uno. Se puede visualizar esta relación en la Tabla 8.1.

Hexadecimal	**Descripción**
01	Micro-interruptor de la cadera
02	Micro-interruptor del hombro
04	Micro-interruptor del codo
08	Micro-interruptor del movimiento *pitch*
10	Micro-interruptor del movimiento *roll*

Tabla 8.1: Valores de los micro-interruptores

```
def get_switch(art, lectura_sw):
state = False
if art == 1:
  if lectura_sw %2 != 0:
    state = True
  else:
    state = False
elif art != 1 and lectura_sw != 0:
  vect_sw = []
  if lectura_sw %2 != 0:
    lectura_sw -= 1
  if lectura_sw >= 16:
    lectura_sw -= 16
    vect_sw.append(16)
  if lectura_sw >= 8:
    lectura_sw -= 8
    vect_sw.append(8)
  if lectura_sw >= 4:
    lectura_sw -= 4
    vect_sw.append(4)
  if lectura_sw == 2:
    vect_sw.append(2)

  for i in range(len(vect_sw)):
    if vect_sw[i] == art:
      state = True
return state

```

Algoritmo 8.1: Algoritmo de detección de micro-interruptores activos

donde *art* es el código del micro-interruptor que se quiere saber si está activo y *lectura_sw* es el sexto doble byte de la última lectura realizada.

Capítulo 9

Modelo cinemático

La estructura modular con la que se ha dotado al programa permite añadirle todo tipo de funcionalidades que mejoren y aumenten su utilidad, como por ejemplo un generador de trayectorias que dé al robot posiciones objetivo que alcanzar dentro de un área de trabajo. En los anteriores capítulos se describe la construcción de la base del programa con el que poder controlar el brazo robótico y sobre la que poder implementar nuevas funcionalidades.

9.1. Generador de trayectorias

En este caso se ha implementado un generador de trayectorias punto a punto que realiza el movimiento simultáneo de los ejes, con el propósito de alcanzar el punto objetivo determinado por el usuario. La cinemática inversa se ha realizado siguiendo el modelo de Denavit-Hatenberg [9] y el código de cálculo se ha desarrollado y adaptado a Python haciendo uso del código extraído de las prácticas de la asignatura de Ampliación Sistemas Robotizados mediante la autorización de Rafael Arnay del Arco. En el

diagrama de la Figura 9.5 se refleja el proceso que se lleva acabo para generar la trayectoria.

La orden a la que se asocia un movimiento de este tipo es la *19*, tal como se indica en la Tabla E.1 del Anexo E, y lleva asociado un parámetro que define el punto objetivo en coordenadas cartesianas y otro que determina la velocidad del movimiento. Una particularidad de esta herramienta es que, a diferencia de otras, debe realizar un procesamiento previo de los parámetros facilitados por el usuario, diagrama de la Figura 9.1.

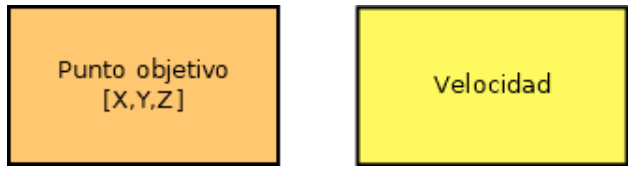


Figura 9.1: Parámetros iniciales

En primer lugar se aplica cinemática inversa al punto objetivo facilitado por el usuario. El generador de trayectorias implementado está pensado para realizar movimientos de 3 grados de libertad (GDL), por lo que se obtendrán los ángulos objetivo de la base, la articulación del hombro y del codo. De aquí se obtiene el ángulo que debe adoptar cada articulación respecto a la posición actual para alcanzar la posición objetivo definida inicialmente, tal como representa en el diagrama de la Figura 9.2. Una vez se comprueba que estos ángulos están dentro del área de trabajo del robot, se puede pasar a la siguiente etapa. La zona de trabajo del robot viene complementada en la Tabla 9.1 donde se resume el rango de acción de cada articulación dada en grados. Todos los ángulos recogidos en esta tabla son con respecto a la posición *home* de cada articulación.

Articulación	Rango de acción
Base	De -90º a 90º (180º)
Hombro	De 15º a 100º(85º)
Codo	De -120º a -30º (90º)

Tabla 9.1: Área de trabajo implementada

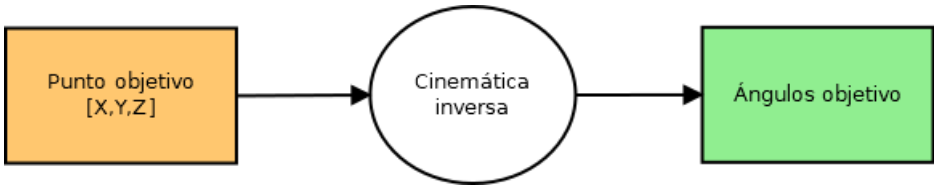


Figura 9.2: Aplicación de la cinemática inversa

En este momento solo faltaría aplicar la transformación de datos definida en la sección 9.2 para obtener las instrucciones de posición que debe mandarse a cada articulación, diagrama de la Figura 9.3. Teniendo esta información, el siguiente paso es empezar el envío de mensajes de movimiento a la controladora USB. La estructura de la información asociada a cada motor seguirá la misma lógica que la expresada en el diagrama de la Figura 6.2. No obstante a diferencia de los movimientos simples, dentro del segmento de **posición de los motores** se incrementará de forma simultanea, según corresponda, la posición de cada motor cuyo movimiento intervenga en alcanzar la posición objetivo. En el diagrama de la Figura 9.4 se ejemplifica este proceso donde se muestra parte del segmento **posición de los motores** donde se hace variar de forma simultánea la posición de la base, el hombro y el codo para alcanzar la posición objetivo. El incremento de la posición sigue la lógica expuesta en la sección 7.2.

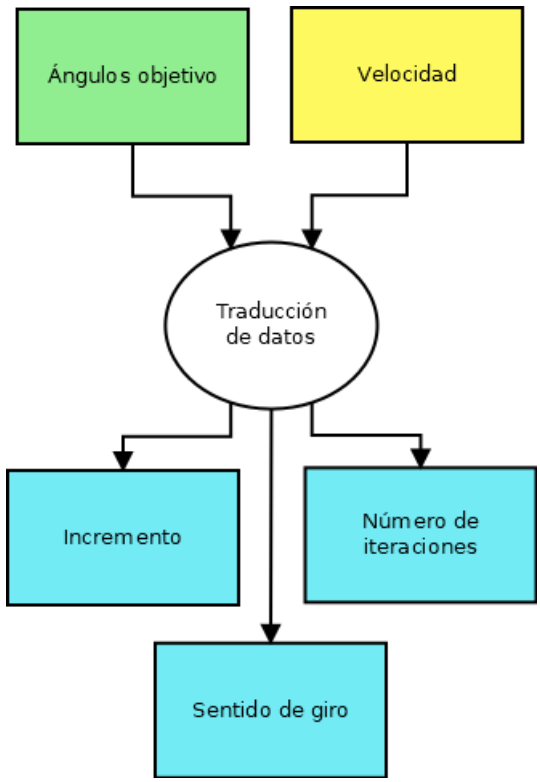


Figura 9.3: Traducción de ángulo a valores de encoder

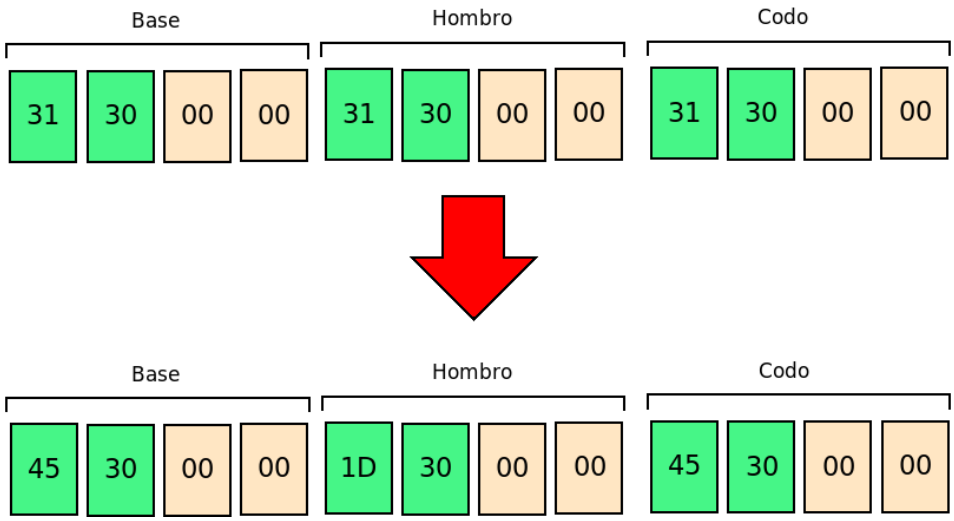


Figura 9.4: Incremento simultáneo de posiciones

Para completar la explicación, se adjunta el diagrama de la Figura 9.5 que contempla la parte del proceso llevado a cabo por el programa para ejecutar la herramienta de generación de trayectorias.

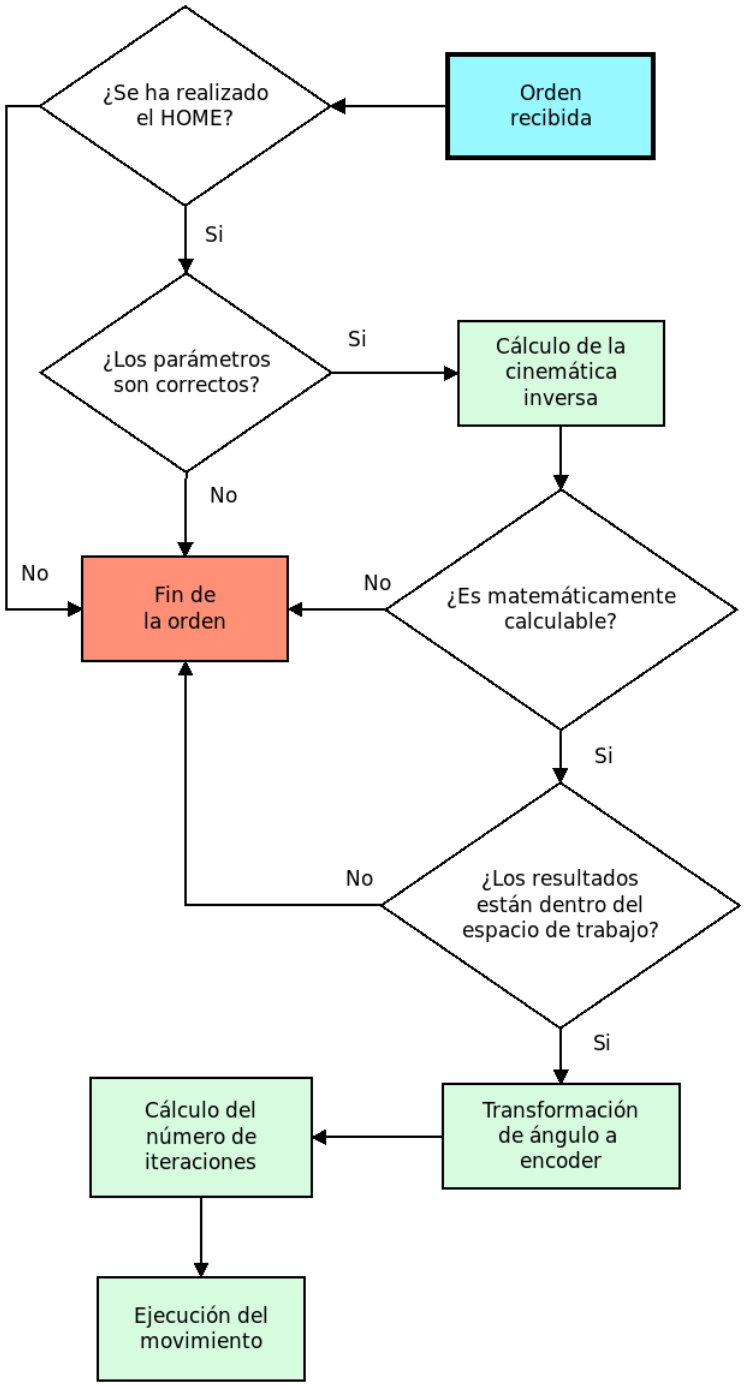


Figura 9.5: Proceso de generación de trayectorias

9.2. Transformación XYZ/encoder

Para realizar un movimiento, el robot necesita saber el sentido de giro, el número de mensajes que se van a enviar y a que velocidad se quiere que se mueva. El sentido de giro viene determinado por la relación entre la posición de referencia, que se corresponde con la posición objetivo alcanzada con la orden previa o con la posición referencia inicial del *home*, y la posición objetivo. En función de la zona en la que se sitúen una y otra dentro del rango de valores de encoder, expresado en la imagen de la Figura 7.6, y según la articulación que se esté evaluando se obtiene el sentido de giro deseado. Las siguientes tablas resumen las distintas posibilidades.

– En el caso en que la posición objetivo y la posición de referencia tengan unos valores que los sitúen en la misma zona, el sentido de giro viene dado por el signo resultante de la diferencia entre la posición objetivo y la de referencia. Tabla 9.2.

Signo / Articulación	Positivo	Negativo
Base	Derecha	Izquierda
Hombro	Bajar	Subir
Codo	Subir	Bajar

Tabla 9.2: Posición objetivo y de referencia en la misma zona

– En el caso en que la posición objetivo esté en la zona inferior y la posición de referencia esté en la zona superior. Tabla 9.3.

Articulación	Sentido
Base	Derecha
Hombro	Bajar
Codo	Subir

Tabla 9.3: Posición objetivo y de referencia en distintas zonas 1

– En el caso en que la posición objetivo esté en la zona superior y la posición de referencia esté en la zona inferior. Tabla 9.4.

Articulación	Sentido
Base	Izquierda
Hombro	Subir
Codo	Bajar

Tabla 9.4: Posición objetivo y de referencia en distintas zonas 2

Independientemente del caso en que se encuentre, para el programa lo relevante es saber si debe realizar una operación de suma o de resta. La siguiente tabla complementa a las anteriores proporcionando la relación entre el sentido de giro y la operación asociada según la articulación. Tabla 9.5.

Operación / Articulación	Suma	Resta
Base	Derecha	Izquierda
Hombro	Subir	Bajar
Codo	Bajar	Subir

Tabla 9.5: Relación entre sentido de giro y aritmética

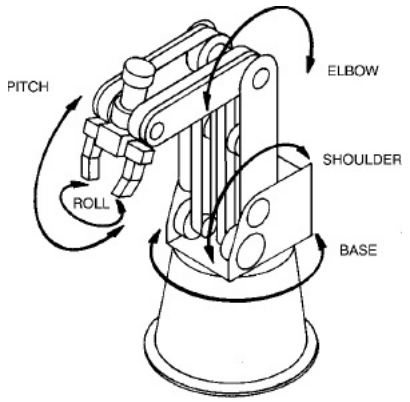


Figura 9.6: Movimientos de las articulaciones

El siguiente paso es determinar el número de mensajes necesarios para comunicar el conjunto del movimiento. Para ello, se calcula en primer lugar el incremento total del valor encoder respecto a la posición actual que cada motor tiene que realizar. La forma de calcularlo dependerá nuevamente de la zona en la que se encuentre la posición objetivo y la posición de referencia la una respecto a la otra.

– En el caso en que la posición objetivo y la posición de referencia tengan unos valores que los sitúen en la misma zona:

$$incremento = |posObj - posRef| \tag{9.1}$$

– En el caso en que la posición objetivo esté en la zona inferior y la posición de referencia esté en la zona superior:

$$incremento = 65535 - posRef + posObj \tag{9.2}$$

– En el caso en que la posición objetivo esté en la zona superior y la posición de referencia esté en la zona inferior:

$$incremento = 65535 - posObj + posRef \tag{9.3}$$

Posteriormente, se calcula el número de mensajes que son necesarios para llevar acabo el movimiento a partir de la ecuación trapezoidal expuesta en la sub-sección 7.2.2.

Capítulo 10

Desarrollo de la interfaz gráfica

Todo programa orientado a interactuar con un usuario está compuesto por un componente funcional y por un componente gráfico. Si se consigue que un programa, por muy complejas que sean las funcionalidades que este implementa, se pueda aprender a usar de forma intuitiva a través de una interfaz gráfica, se considera que la transición entre el *frontend* [1] y el *backend* [2] del programa se ha realizado con éxito.

En este caso, la interfaz gráfica de usuario ha sido diseñada y creada con la herramienta Qt Designer de Qt. Está basada en uso de Widgets que cohesionan automáticamente los elementos gráficos con el código funcional o de programación. Además se trata de una plataforma que cuenta con licencia de código abierto, con ciertas limitaciones respecto a la versión de licencia comercial, pero cuyas prestaciones de la versión *open source* cubren todas las necesidades para el desarrollo de este proyecto.

[1] Elementos del programa destinados a interactuar con el usuario.

[2] Elementos del programa invisibles para el usuario y que procesan las tareas indicados por éste.

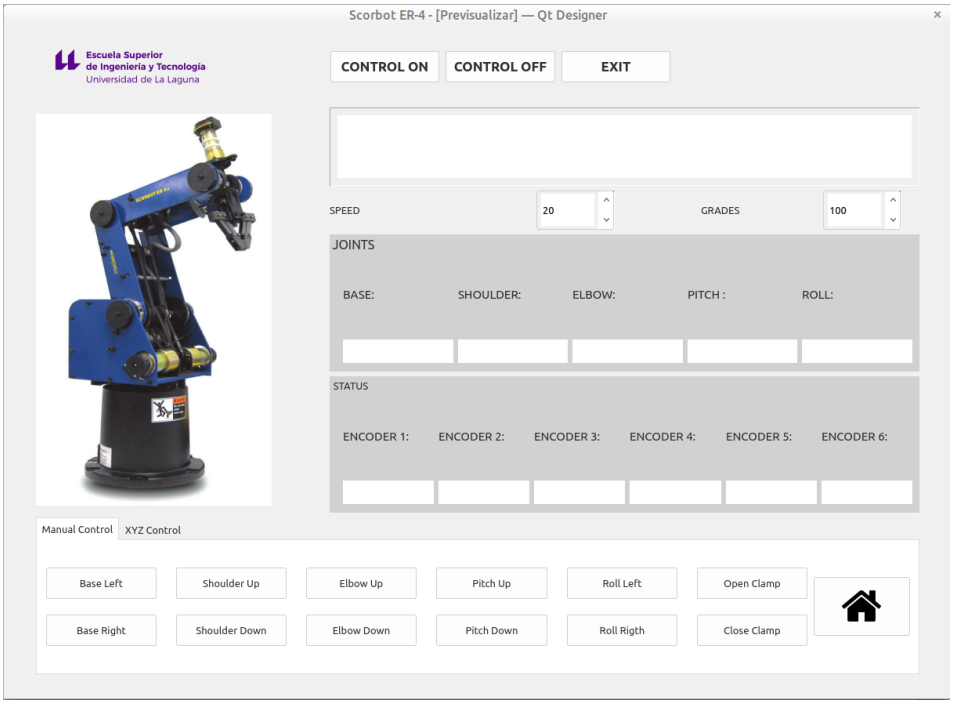


Figura 10.1: Ventana principal de la GUI

El objetivo de esta herramienta es crear una experiencia cómoda e intuitiva al usuario que haga uso de ella. En la vista general que observamos en la imagen de la Figura 10.1 podemos diferenciar cinco secciones principales:

– **Botones de control:** Encargados del control general del programa. Habilitando o deshabilitando el control sobre los motores, o ejecutando el protocolo de cierre. En la imagen de la Figura 10.2 se diferencia en el modo *Control Off* como se deshabilitan los botones de movimiento y se muestra un mensaje de estado.

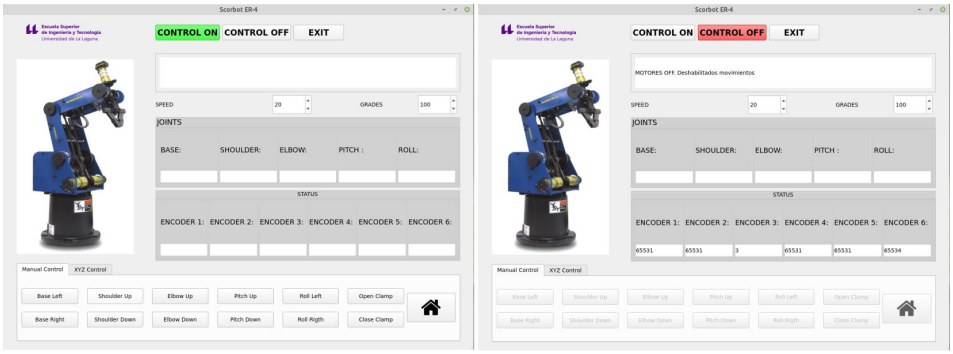


Figura 10.2: Vista de la interfaz con CONTROL ON y CONTROL OFF.

– **Control de movimientos:** Dispuesto en una ventana con dos pestañas, tal y como se observa en la imagen de la Figura 10.3. La primera da lugar a los movimientos simples descritos en el Capítulo 7 y al *home* del Capítulo 8. La segunda pestaña presta control sobre el robot en torno a un punto XYZ, haciendo uso del programa desarrollado en el Capítulo 9.

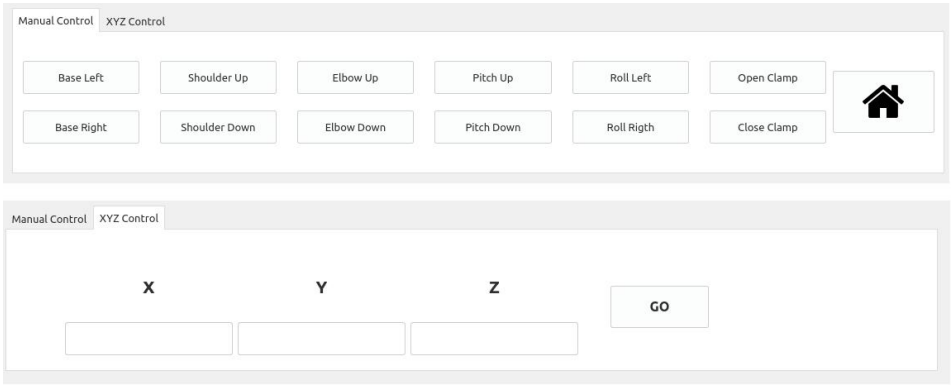


Figura 10.3: Pestañas de control manual y XYZ.

– **Ventana de información:** En la ventana superior se muestra el estado del programa en cada instante.

– **Encoders y joints:** Muestra el valor de los encorders en base decimal y los ángulos de cada articulación una vez se ha realizado el *home*. En la imagen de la Figura 10.4 se atiende al resultado de la interfaz gráfica de usuario tras finalizar el *home*.

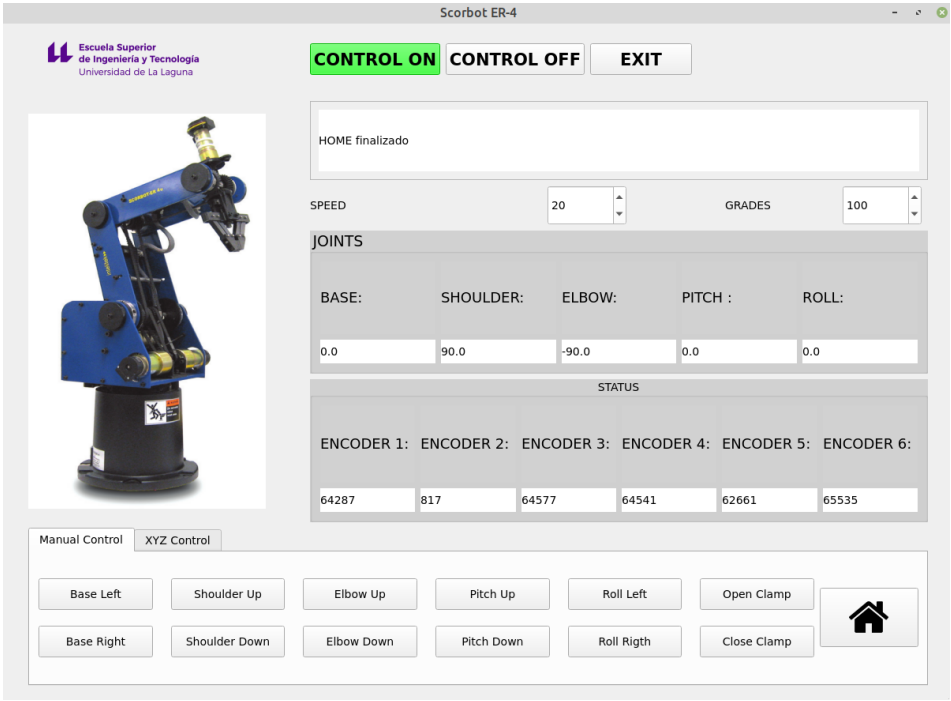


Figura 10.4: Vista de la interfaz con los joints inclusive.

– **Velocidad y grados:** Cuadros de texto donde se determina la velocidad y los grados del movimiento simple a realizar.

10.1. Conexión y lanzamiento de la GUI

La primera etapa tras la ejecución del programa puede tener dos posibles resultados del que depende enteramente el lanzamiento inicial de la interfaz gráfica.

Por un lado, se puede dar la situación en la que el programa detecta a la controladora USB y consigue establecer comunicación con ella. En este caso, la interfaz gráfica se lanza con el modo *motores habilitados* o *Control On* y el programa estaría listo para ejecutar órdenes.

Por otro lado, la otra situación posible es que el programa no consiga establecer la conexión inicial con la controladora USB. En este caso, la interfaz gráfica se lanza directamente en su versión *motores deshabilitados* o *Control Off* y un mensaje informativo en la ventana de información. El proceso a seguir en esta situaciones es revisar que todo esté debidamente conectado y entonces volver a ejecutar el programa.

10.2. Respuesta ante detección de eventos

La interacción entre el usuario y la interfaz gráfica queda reflejada en la detección de los eventos que se generan al interactuar entre ellos.

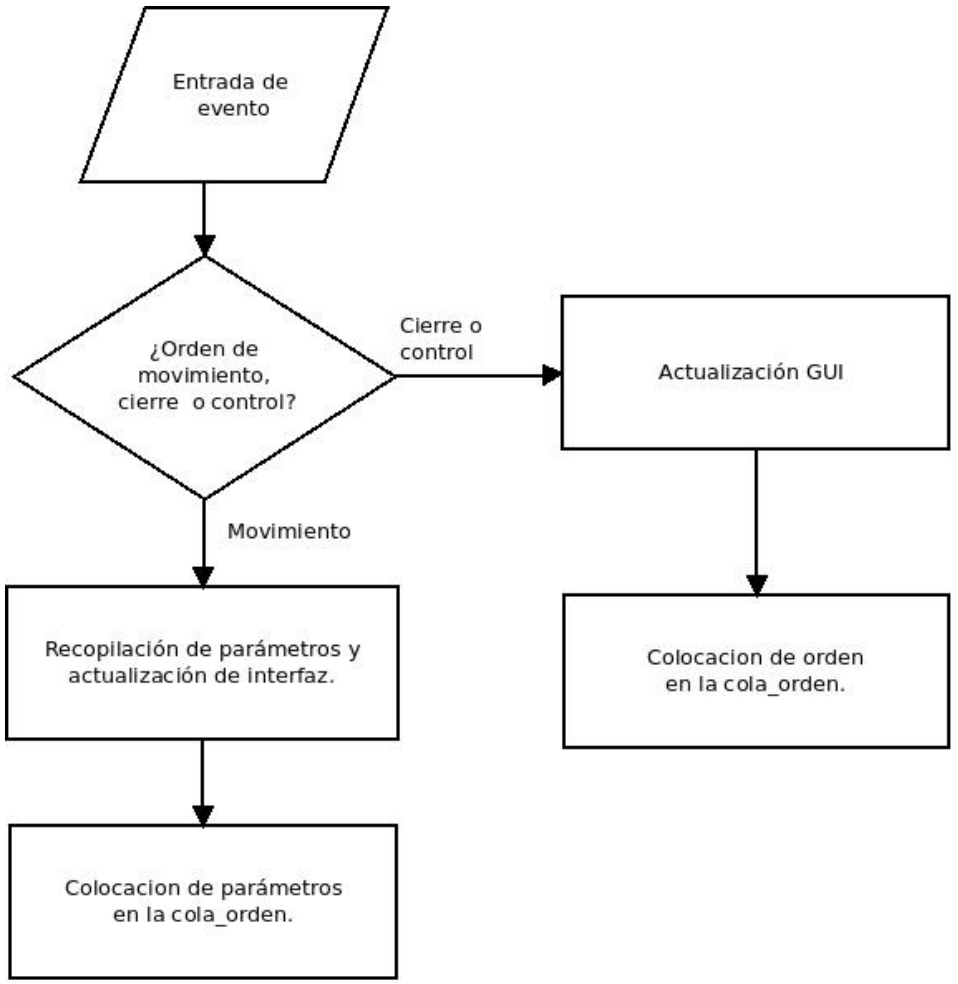


Figura 10.5: Diagrama funcional de la recogida de eventos de la GUI

Al detectarse un evento tras pulsar un botón, la interfaz gráfica actualiza la ventana de información, quedando esta en estado de espera mientras el software mantiene la comunicación necesaria con la controladora USB asociada a la petición del usuario. A partir de este punto, las instrucciones

dadas por el usuario pasan a ser recopiladas junto con los parámetros relacionados al evento como pueden ser los parámetros de velocidad y grados en caso de movimiento simple, o los parámetros X, Y y Z y velocidad en caso de movimiento por control XYZ. En función de la orden, se muestra por interfaz un mensaje que describe la ejecución que se va a realizar, además de dejar patente en el registro la acción que se va a llevar acabo. El resto del proceso pasa a ser responsabilidad del hilo de ejecución que, tras finalizar la tarea, devolverá un código de proceso que indicará cuál ha sido el resultado de la ejecución.

10.3. Estado de encoders y joints

Sección de la interfaz gráfica destinada a dar un información al usuario sobre la evolución de la posición de cada motor.

Por un lado, la sección de encoders muestra en todo momento en base decimal la posición que adopta cada motor después de un movimiento, tomando como referencia el valor de encoder leído en el momento en el que se lanzó el programa.

Por otra parte, la sección joints muestra valores angulares de las posiciones de los motores, pero siempre respecto a la posición de *home*. Es por ello que este apartado no muestra datos hasta que no se ha realizado la acción del *home* sin producir ningún error de ejecución.

Capítulo 11

Resultados

Un programa funcional y de uso intuitivo son características muy apreciadas en una aplicación, sobre todo en una de carácter educativo. El conjunto de los capítulos anteriores conforman la primera versión del programa Open Scorbot, software de código abierto diseñado para el control multiplataforma de brazos robóticos antropomóficos, como es el caso del Scorbot ER-4U.

El programa ha resultado en un conjunto de diez scripts de Python y un archivo con formato *user interface*, que contiene la interfaz gráfica. La relación entre ellos se refleja en el diagrama de la Figura 11.1 donde se han enmarcado las distintas partes del programa dentro de la estructura inicial planteada en el diagrama de la Figura 2.3.

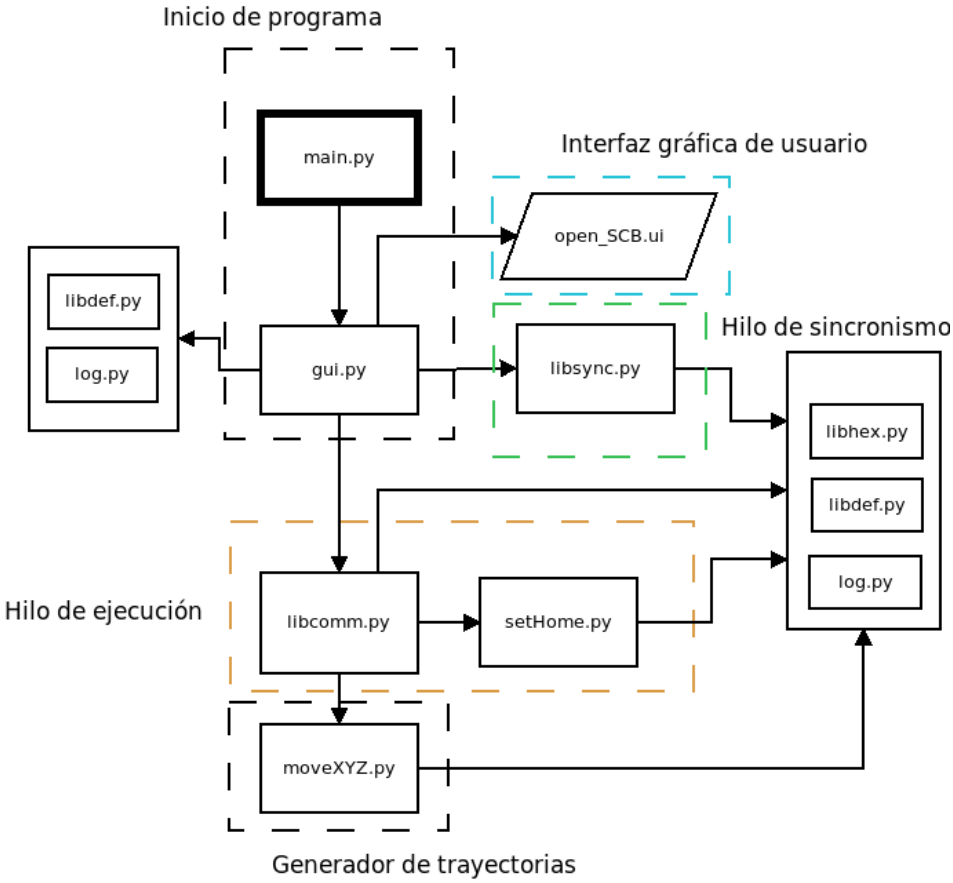


Figura 11.1: Estructuración de los scripts del programa

A continuación se da una breve definición de la funcionalidad asociada a cada uno de los archivos:

– **main.py**: Script principal del programa. Se encarga de iniciar el proceso de ejecución haciendo una llamada al script *gui.py*. Además se encarga de cerrar los hilos una vez el programa termine de ejecutarse.

- **gui.py**: Script encargado en primer lugar de crear la via de comunicación con la controladora. También es el responsable de la creación de los dos hilos de ejecución que componen el programa, además de ser el lanzador de la interfaz gráfica.

- **libsync.py**: Realiza la secuencia de conexión inicial con la controladora, además de ejecutar el hilo encargado de matener la sincronización con la misma.

- **libcomm.py**: Realiza el hilo de ejecución encargado de evaluar las órdenes especificadas desde la interfaz. También realiza la ejecución de los movimientos simples de las articulaciones.

- **setHome.py**: Realiza la secuencia de movimientos encargada de buscar la posición de *home* de todas las articulaciones.

- **moveXYZ.py**: Realiza el proceso de generación de una trayectoria a partir de una posición inicial y una posición objetivo. Se encarga tanto de generar los datos para llevar acabo el movimiento como del propio movimiento.

- **libdef.py**: Libería de funciones de carácter general presentes en todo el programa.

- **libhex.py**: Librería con secuencias de datos estándar en base hexadecimal que son usadas en distintos puntos del programa.

- **log.py**: Realiza la configuración de los mensajes de registros generados en el programa.

- **open_SCB.ui**: Fichero correspondiente a la interfaz gráfica desarrollada en el Qt 5 Designer.

Haciendo un análisis en conjunto del proyecto, la ejecución del programa termina pudiéndose englobar dentro del diagrama definido en el diagrama de la Figura 2.4 donde además se contempla una funcionalidad añadida, el control de errores(ver Anexo D).

Capítulo 12

Conclusiones

Desde el primer momento se ha trabajado bajo la premisa de que el programa resultante fuera de código abierto, de modo que en el futuro pueda ser modificado, corregido y mejorado. La principal ventaja de este tipo de desarrollo es el potencial crecimiento y mejora del programa que se puede generar en caso de crearse una comunidad de trabajo en torno al programa.

El programa *Openscorbot* desarrollado en este proyecto cumple los siguientes objetivos:

1. El programa es multiplataforma, lo que significa que puede ser ejecutado en distintos sistemas operativos siempre que se tenga instalada la versión de Python correcta, y se ha desarrollado como código abierto y libre distribución haciendo uso de herramientas con la misma característica.

2. Control sobre las principales funcionalidades del Scorbot ER-4U.

3. Desarrollo de la funcionalidad que permite al robot llevar acabo la búsqueda de la posición referencia *home*.

4. Implementación de un generador de trayectorias que permite realizar movimientos complejos sobre un plano en coordenadas cartesianas.

5. Desarrollo de una interfaz gráfica de usuario para el uso del programa.

El punto de inflexión a partir del cual se comenzó a realizar progresos fue en el momento en el que se comprendió el concepto de tiempo de procesamiento, es decir, tiempo que tarda la controladora USB en procesar la orden que se la indicado y en realizar una respuesta en consecuencia. Una vez establecido un tiempo medio de respuesta para la ejecución de movimientos de cada articulación, el desarrollo del resto de funciones resultó ser más sencillo por estar estas basadas en el movimiento simple de cada articulación.

12.1. Trabajo futuro

Una vez establecidas las bases con las que comunicarse y dar órdenes básicas al robot, todo esfuerzo debe dedicarse al desarrollo de nuevas herramientas que pongan a disposición del usuario un mayor abanico de opciones de trabajo, como otro tipo de generadores de trayectorias.

El siguiente paso para mejorar la funcionalidad del robot, sería implementar un sistema de control vía internet haciendo uso de una *Raspberry Pi* dedicada que se comunique con el usuario a través de internet y con la controladora USB de forma local. De estar forma sería posible, por ejemplo, telecomandar el robot.

Conclusions

Since the beginning it has been worked under the premise that the program turned out to be open source, so that in the future it can be modified, corrected and improved. The main advantage of this type of development is the potential growth and improvement of the program that can be generated if a working community is created around the program.

The multi-platform Open Scorbot program developed in this project meets the following objectives:

1. The program is multiplatform, which means that it can be run on different operating systems as long as the correct version of Python is installed, and it has been developed as open source and free distribution using tools with the same characteristic.

2. Control over the main functionalities of the Scorbot ER-4U.

3. Development of the functionality that allows the robot to carry out the search for the home reference position.

4. Implementation of a trajectory generator that allows complex movements on a plane in Cartesian coordinates.

5. Development of a graphical user interface for the use of the program.

The turning point from which progress began to be made was at the time when the concept of processing time was understood, which is the time

it takes for the USB controller to process the order indicated and to carry out a properly answer. Once an average response time was established for the execution of movements of each joint, the development of the rest of the functions turned out to be easier since they were based on the simple movement of each joint.

12.2. Future work

Once established the bases with which to communicate and give basic orders to the robot, all effort should be devoted to the development of new tools that make a greater range of work options available to the user, such as other types of trajectory generators.

The next step to improve the functionality of the robot would be to implement a control system via the internet using a dedicated raspberry that communicates with the user via the internet and with the USB controller locally. In this way it would be possible, for example, to remote control the robot.

Bibliografía

[1] Varios autores. «Librerias Python». https://www.docs.python.org.

[2] Varios autores. «Linux Mint». https://www.linuxmint.com.

[3] Varios autores. «Python». URL: https://www.python.org.

[4] Varios autores. «PyUSB». URL: https://github.com/pyusb/pyusb.

[5] Varios autores. «Qt Designer». https://doc.qt.io/qt-5/qtdesigner-manual.html.

[6] Jan Axelson. *«USB Complete»*. Third edition.

[7] Antonio Barrientos. *Fundamentos de la Robótica*. Segunda Edición. Mc Graw Hill, 2007.

[8] Yeray Miranda Betancor. *Estudio y análisis de las comunicaciones en el robot Scorbot ER- 4U. Desarrollo de un driver e implementación para su uso en interfaz web*. 2017.

[9] Praneel Chand Raul R Kumar. «Inverse kinematics solution for trajectory tracking using artificial neural networks for SCORBOT ER-4U». En: *Conference Paper* (2015).

[10] *Scorbot-ER 4u, Manual de usuario*. Intelitek. Sep. de 2001.

Universidad de La Laguna
Escuela Superior de Ingeniería y Tecnología
Departamento de Ingeniería Industrial

ESCUELA DE SUPERIOR DE
INGENIERÍA Y TECNOLOGÍA

Trabajo de Fin de Grado

Desarrollo e implementación de un controlador multiplataforma para el Scorbot-ER4U

TOMO II

Pliego de condiciones y presupuesto

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Estudiante: Yolanda Mercedes Gimeno Rodríguez
Estudiante: José Luis Pérez Pérez

Tutor: Fernando Luis Rosa González
Cotutor: Iván Rodríguez Méndez

9 de julio de 2020

Capítulo 13

Pliego de condiciones

La principal condición para la realización de este proyecto consiste en que el desarrollo del programa de control multi-plataforma de un brazo robótico antropomórfico debe usar *software* libre, cerciorando que sea una versión funcional y de libre acceso y distribución. Otras condiciones para el desarrollo del proyecto son las siguientes:

- Emplear un sistema operativo compatible con la versión de Python utilizada.
- Utilizar *Qt 5 Designer* para el desarrollo del diseño de la interfaz gráfica.
- Usar *Latex* para la redacción y documentación del proyecto.
- Emplear *Git* para la compartición de archivos y documentos del proyecto.
- Utilizar Python, versión 3.6.9, para el desarrollo del software.

Capítulo 14

Presupuesto del proyecto

El presupuesto del proyecto asciende a un total de $7490,0$ €. En la Tabla 14.1, se descompone dicho presupuesto, donde se indican los elementos implicados, su cantidad, su precio unitario (si procede) y el precio total.

Los gastos de este proyecto se componen exclusivamente de la mano de obra, puesto que no se ha tomado como gasto en materiales al robot manipulador, su controladora USB ni los interfaces entre estos. Son productos adquiridos por el departamento de Ingeniería Industrial de la Universidad de La Laguna con fecha anterior al desarrollo de este proyecto y con fines meramente educativos.

Descripción	Horas	Precio/hora	Coste total
Desarrollo de software	200	20 €	4000 €
Análisis de datos	100	20 €	2000 €
Redacción del proyecto	50	20 €	1000 €
		Sub Total	7000 €
		IGIC(7 %)	490 €
		Total	7490 €

Tabla 14.1: Descripción de los gastos

Universidad de La Laguna
Escuela Superior de Ingeniería y Tecnología
Departamento de Ingeniería Industrial

ESCUELA DE SUPERIOR DE
INGENIERÍA Y TECNOLOGÍA

Trabajo de Fin de Grado

Desarrollo e implementación de un controlador multiplataforma para el Scorbot-ER4U

TOMO III

Anexos

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Estudiante: Yolanda Mercedes Gimeno Rodríguez
Estudiante: José Luis Pérez Pérez

Tutor: Fernando Luis Rosa González
Cotutor: Iván Rodríguez Méndez

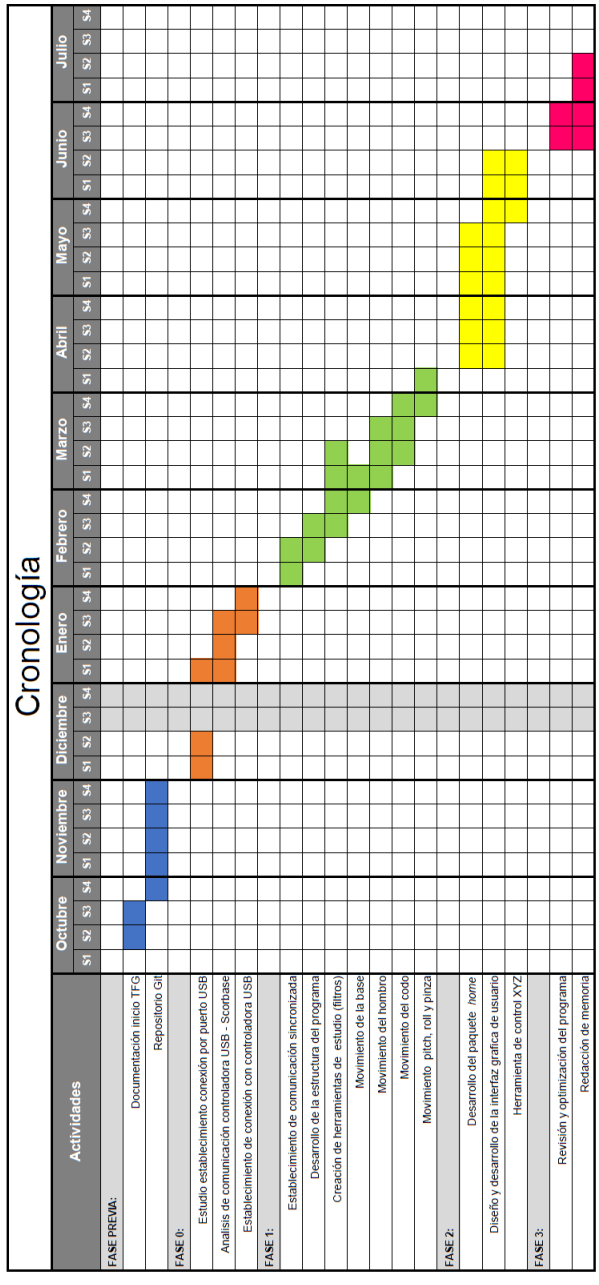
9 de julio de 2020

Anexo A

Planificación temporal

En este anexo se encuentra la cronología que se ha seguido para el desarrollo del presente proyecto. Se expone una cronología por fases, donde se detalla los diferentes hitos a seguir y el tiempo dedicado para ello, y un diagrama piramidal de la evolución del trabajo.

A.1. Cronología del proyecto



A.2. Distribución por etapas del trabajo

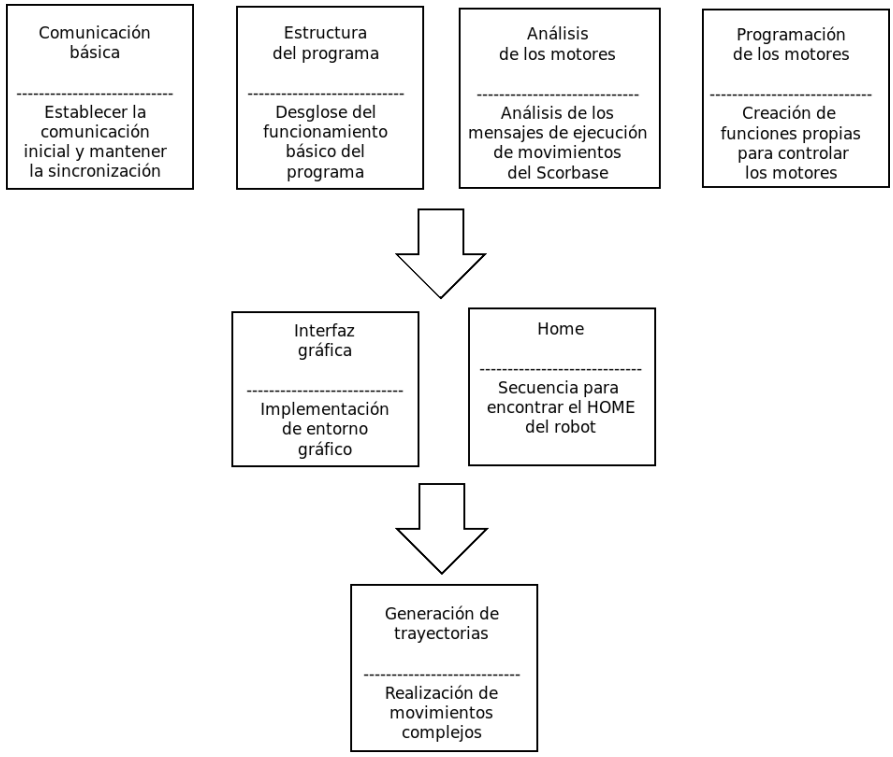


Figura A.1: Diagrama de evolución del trabajo

Anexo B

Diagrama de flujo del programa

En el presente anexo se muestra un diagrama de flujo que representa la funcionalidad completa del programa desarrollado, contemplando de forma genérica los posibles escenarios que pueden presentarse durante la ejecución del mismo.

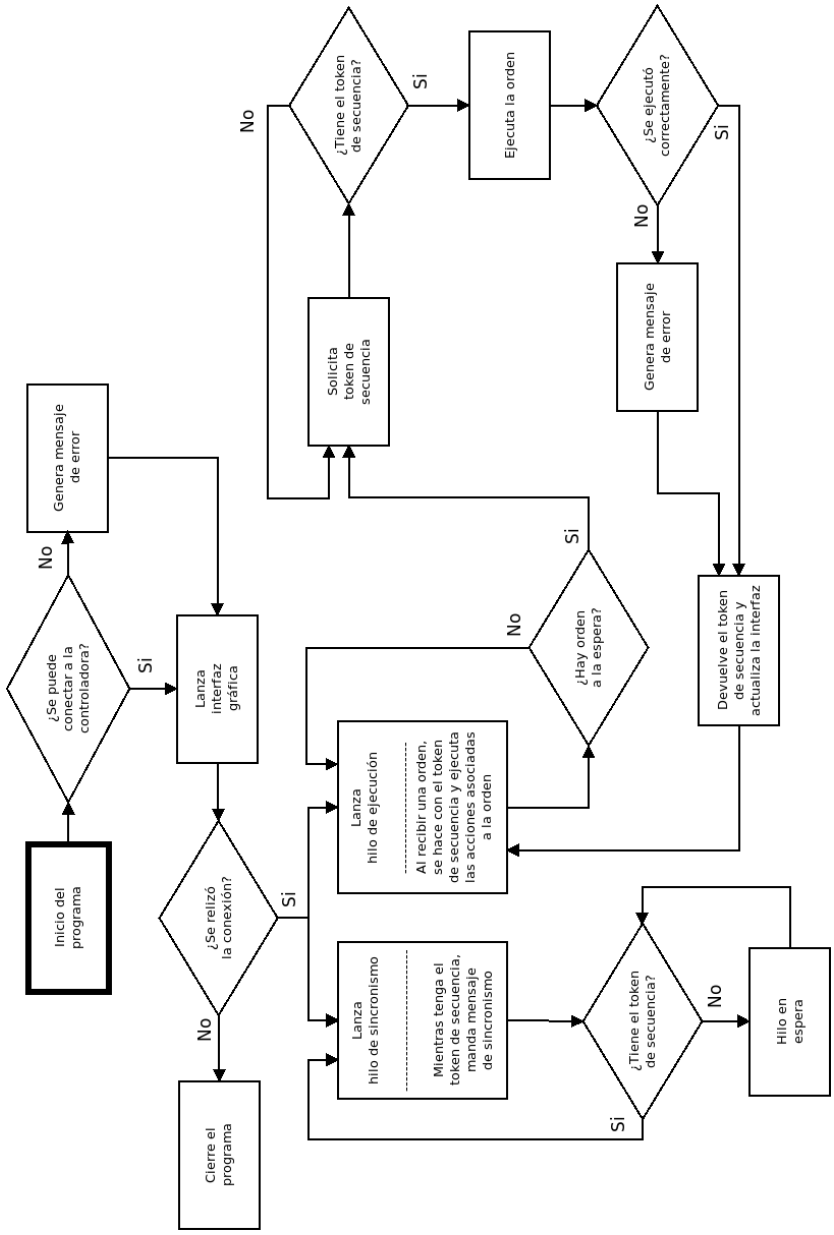


Figura B.1: Estructura de ejecución

Anexo C

Herramientas de análisis de datos

C.1. filter_SCB.py

Filtro que permite generar un fichero que contenga solo la columna de datos de los mensajes correspondientes al *endpoint IN, OUT* o ambos eliminando las repeticiones que genera la controladora por defecto que dificultan el análisis, además del resto de grupos de datos.

C.2. calculate_SCB.py

Representa gráficamente la diferencia en valores de *encoder* de la posición de un motor entre dos mensajes consecutivos. Se emplea para conocer la relación entre la velocidad asignada por el usuario y el número de mensajes necesarios para realizar un determinado movimiento.

C.3. calculateTime_SCB.py

Este script se centra en la columna *Time* del archivo de datos original. El objetivo es determinar el tiempo que existe entre cada dos mensajes de cada tipo, mensaje IN o OUT, a lo largo del proceso para finalmente calcular un tiempo medio de procesamiento, que es el parámetro a partir del cual se empieza a determinar la cantidad de tiempo que necesita la controladora para atender las peticiones de movimiento del host sin producirse pérdidas de datos.

C.4. graphic_SCB.py / multigraphic_SCB.py

Representa gráficamente las distintas posiciones por las que pasa un motor durante la toma de datos, permitiendo comprender el funcionamiento global de cada articulación y determinar los valores límite de cada uno. Representa tanto los datos asociados a la posición en los mensajes de escritura como de lectura, de forma que puede verse como los mensajes de lectura "siguen.ª los de escritura.

El script multigraphic_.py sigue la misma lógica que el anterior pero con la diferencia que estudia la posición de todos los motores a la vez. Con las gráficas generadas es posible ver con mayor facilidad la dependencia de movimientos entre articulaciones.

C.5. global_error_SCB.py

Genera una tabla en terminal desglosando los bytes de error asociados a cada motor en los mensajes de lectura. Permite determinar a partir de que valor de error el Scorbase determina que ha habido un fallo de funcionamiento.

Por otra parte, para comprobar los progresos en el programa, se realizaron scripts espejo de los anteriormente expuestos compatibles con la estructura de fichero generada por el programa con los que analizar los mensajes generados a lo largo de su funcionamiento. El volcado de datos a fichero .csv se realiza desde terminal; desde programa se hace uso de la función *filter* del script *libdef.py*.

Anexo D

Control de errores

Con el fin de evitar bloqueos durante la ejecución del programa, se ha implementado un sistema de control de errores en el que se usan identificadores asociados al error y un sistema de registros con el que crear un historial de ejecución del programa. Los puntos de la ejecución del programa donde ha sido necesario implementar el sistema se pueden ordenar por cronología de ejecución.

En primer lugar, se comprueba que el programa es capaz de conectarse a la controladora del robot y establecer una vía de comunicación. De no ser posible, se lanza la interfaz gráfica con el mensaje de error correspondiente y se bloquea cualquier envío de datos hacia la controladora que no sea la de cerrar el programa.

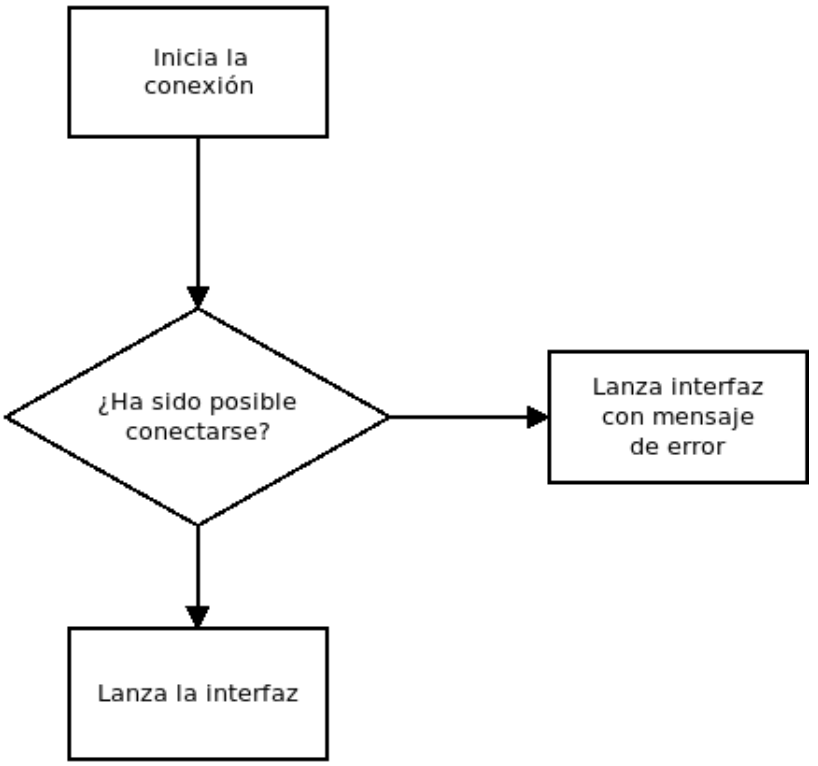


Figura D.1: Control de errores en el inicio del programa

Una vez lanzada satisfactoriamente la interfaz, el control de errores se ramifica de forma que todas las órdenes que implican movimiento del robot tienen contemplados los errores más usuales que pueden aparecer durante la normal ejecución del programa. En este punto, los posibles errores que pueden darse se agrupan en dos subgrupos:

– **Error durante movimiento**: El programa detecta un mal funcionamiento del brazo robótico mediante una comparación entre la posición que debe tener el motor y el valor que realmente tiene.

– **Error de procesamiento**: No ha sido posible calcular todos los parámetros necesarios para llevar a cabo el movimiento. Esta situación se puede deber a un valor mal introducido en la interfaz, por la imposibilidad matemática de calcular un ángulo mediante cinemática inversa o por tener como resultado de la misma unos ángulos que sobrepasan el área de trabajo. En cualquier caso, la interfaz gráfica sacará por pantalla un mensaje de *feed-back* con el que notificar el evento al usuario.

La lógica implementada para llevar a cabo mediante programa este control de errores se desglosa en el siguiente diagrama:

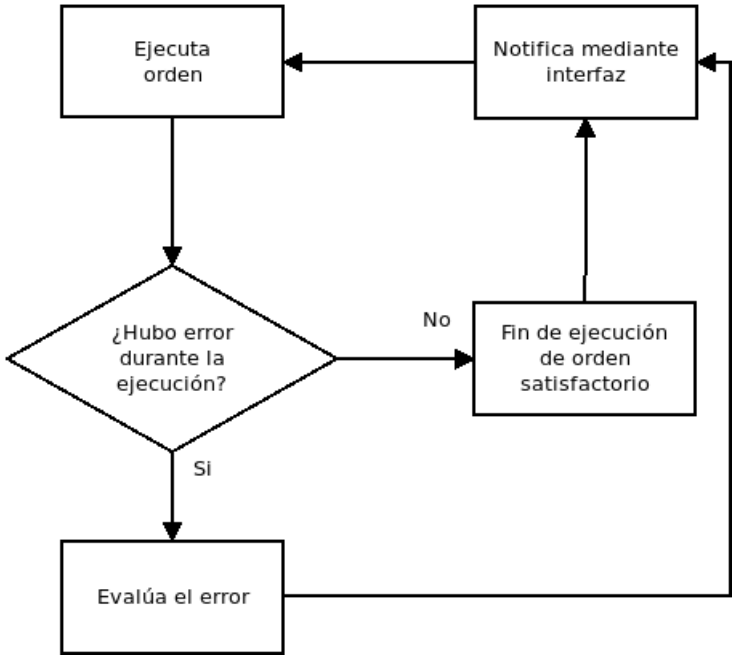


Figura D.2: Estructura de la función *control_error*

Anexo E

Órdenes y acciones

A continuación se adjunta una tabla que recoge todos los códigos de órdenes que pueden ser enviadas por el usuario al programa mediante la pulsación de los distintos botones presentes en la interfaz gráfica.

Orden	**Acción**
4	Mover base a la izquierda
5	Mover base a la derecha
6	Subir el hombro
7	Bajar el hombro
8	Subir el codo
9	Bajar el codo
10	Movimiento de *pitch* elevando la pinza
11	Movimiento de *pitch* bajando la pinza
12	Movimiento de *roll* en sentido horario
13	Movimiento de *roll* en sentido anti-horario
14	Abrir pinza
15	Cerrar pinza
16	Deshabilitar los motores
17	Habilitar los motores
18	Realizar el *home*
19	Alcanzar un punto [x,y,z] definido por el usuario
528	Cerrar el programa y cerrar conexiones

Tabla E.1: Órdenes y acciones

Anexo F

Nociones sobre la comunicación USB

La comunicación USB usada por el Scorbot ER-4U facilita al desarrollador de software la creación de nuevas herramientas con las que utilizar este equipo.

La comunicación se basa en la interconexión entre un *host* o manager de la conexión y un *device* o dispositivo. Desde el punto de vista del host, este tipo de conexión realiza de forma predeterminada los primeros pasos para establecer una vía de comunicación, de forma que solo hay que preocuparse de gestionar la información inicial y no de realizar las peticiones para su obtención. Esta primera parte está constituida por mensajes de control o *setup* que normalmente son invisibles para el usuario y se la denomina enumeración. De ahí en adelante, todo intercambio de información entre el *host* y el dispositivo se corresponderán con transferencias de datos asociadas a la propia funcionalidad de la aplicación.

La comunicación USB se puede dividir en dos categorías: las comunicaciones para realizar la enumeración y las comunicaciones que necesitan las aplicaciones para llevar acabo las funciones del dispositivo [6]. La enumeración es un proceso mediante el cual el host aprende sobre el dispositi-

vo mientras que las comunicaciones de las aplicaciones son todas aquellas transferencias de datos cuyo objetivo es llevar a cabo una acción.

F.1. Componentes de la comunicación

La finalidad de establecer una vía de comunicación es la de intercambiar paquetes de datos con uno o varios dispositivos. Para conseguirlo, es necesario establecer líneas de comunicación entre ellos haciendo uso de los datos recopilados en la enumeración. El resultado es la conexión mediante pipes de los *endpoints* de los dispositivos con el *host*.

Endpoint

Un endpoint es un buffer o espacio de memoria donde se almacena la información previo procesamiento. La información fluye en ambos sentidos en una conexión USB por lo que tanto en el destino como en el origen debe haber algún elemento que la almacene: los buffers. En los dispositivos estos elementos se conocen como endpoints, que son definidos por las especificaciones USB [6]. De esta definición se determina que un dispositivo dispone de unos *endpoints* cuyas características son únicas e invariables. Por otra parte, en el *host* también hay buffers, pero no son de tipo endpoint ya que disponen de espacios de memoria reservados exclusivamente para esta función y pueden emplearse los mismos para comunicarse alternativamente con cada dispositivo. El host actúa, por tanto, como inicio y final de los mensajes transmitidos en una conexión USB.

El endpoint se caracteriza por tener definido un sentido y una dirección o nombre característico. El sentido puede ser IN o OUT, según sea información que ha enviado el host al dispotivo o información que ha enviado el dispositivo al host, respectivamente. En cuanto a la dirección, se trata de dos bytes que identifican el punto de origen o destino de la información y deberá ser único dentro de cada conexión.

Sentido	Origen	Destino
IN	Dispositivo	Host
OUT	Host	Dispositivo
Setup	Host	Dispositivo

Tabla F.1: Endpoints

Pipe

Es la representación abstracta del vínculo de conexión entre host y dispositivo donde se enlazan los endpoints del dispositivo con el software del host.

El host crea estos pipes durante la enumeración, de forma que si el dispositivo es desconectado los pipes son eliminados. Por defecto, siempre existe un pipe denominado *Defualt Control Pipe* que enlaza con el endpoint 0 para realizar la conexión referente a los mensajes de SETUP.

Handshake

Elemento dentro de una transacción empleado para el control del flujo de datos. Consiste en un sistema de *status* y *control* con el que confirmar la recepción y el correcto envío de información que permite decidir al receptor que hacer a continuación.

Este sistema es usado en todos los tipos de transferencias salvo en las isócronas. Para indicar el éxito o el fallo de alguna transferencia, existe un conjunto de códigos que se asocian a cada una de las posibles respuestas que un código *handshake* puede albergar.

Los *status* de *handshake* existentes son los de ACK, NAK, STALL y NYET. En caso de ausencia de código de *status*, se entiende que se ha producido un error más grave.

– **ACK**:Indica que un host o un dispositivo ha recibido la información sin errores.

- **NAK**: Indica que el dispositivo está ocupado o que no tiene información que devolver. El host nunca envía este handshake.

- **STALL**: Puede indicar tres eventos distintos: solicitud de control no soportada, solicitud de control fallida o fallo en el endpoint. Este *handshake* se usa en transferencias de control con el fin de realizar *feed-backs* más explícitos en un tipo de transferencia tan crítico.

- **NYET**: Protocolo que permite al host saber antes de enviar información si el dispositivo está listo para recibirla. La respuesta del dispositivo ante este *handshake* será otro de tipo ACK o NAK según esté o no disponible.

Descriptores

Los descriptores USB son conjuntos de datos estructurados que contienen información sobre un dispositivo y son la respuesta ante las peticiones de datos como *Get_Descriptor* de un *host*. Ésta puede ser una descripción general del dispositivo o datos más específicos como los *endpoints* de cada interfaz, de forma que toda la información queda estructurada por niveles:

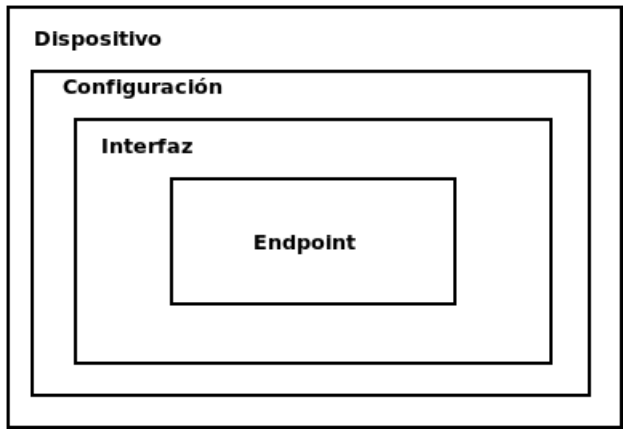


Figura F.1: Jerarquía de los descriptores

Los niveles superiores informan al *host* de la existencia de los inferiores.

Cada dispositivo tiene un único *device descriptor* del que salen los *configuration descriptors*, uno por cada configuración soportada por el dispositivo. En cada *configuration descriptor* viene definido, entre otras cosas, el número de interfaces que soporta la configuración. A su vez, el *interface descriptor* especifica el número de *endpoint descriptors* presentes en la interfaz o la ausencia de ellos, lo que obligaría a la interfaz a comunicarse mediante el *endpoint* de control. Por último, cada *endpoint descriptor* contiene toda la información necesaria para comunicarse con el *endpoint* que define.

Device descriptor

Contiene información básica sobre el dispositivo. Es el primer descriptor que recibe el *host* tras conectarse a un dispositivo. Este descriptor tiene catorce campos distintos.

Campo	Descripción
bLength	Tamaño en bytes del descriptor
bDescriptorType	-
bcdUSB	Versión USB soportada
bDeviceClass	-
bDeviceSubclass	-
bDeviceProtocol	-
bMaxPacketSize0	Tamaño de paquete máximo para en *endpoint 0*
idVender	Identificador de vendedor
idProduct	Identificador de producto
bcdDevice	-
iManufacturer	-
iProduct	-
iSerialNumber	-
bNumConfigurations	Número de posibles configuraciones

Tabla F.2: Campos de un *device descriptor*

Configuration descriptor

Cada dispositivo tiene como mínimo una configuración disponible. En este conjunto de información se encuentra lo relacionado con el consumo de corriente del equipo y el número de interfaces soportadas. En este caso, el descriptor está formado por ocho campos.

Campo	Descripción
bLength	Tamaño del descriptor en bytes
bDescriptor	-
wTotalLength	Número de bytes en el descriptor y los relacionados
bNumberInterfaces	Número de interfaces en la configuración
bConfigurationValue	Diferencia entre peticiones
iConfiguration	-
bmAttributes	-
bMaxPower	Potencia requerida por el bus

Tabla F.3: Campos de un *configuration descriptor*

Interface Descriptor

Contiene información sobre las funcionalidades y características del dispositivo. Además, contiene la información concerniente a los *endpoints*. Una configuración puede contener varias interfaces, que no tienen por qué tener funcionalidades relacionadas, pero que son gestionadas dentro de la misma configuración.

Un *interface descriptor* está formado por nueve campos:

Campo	Descripción
bLength	Tamaño del descriptor en bytes
bDescriptorType	-
bInterfaceNumber	Identificador de la interfaz
bAlternateSetting	-
bNumEndpoints	Número de *endpoints* soportados, sin contar el 0
bInterfaceClass	-
bInterfaceSubclass	-
bInterfaceProtocol	-
iInterface	*String* descriptivo de la interfaz

Tabla F.4: Campos de un *interface descriptor*

Endpoint descriptor

Cada *endpoint* tiene un conjunto de datos que lo define y caracteriza. Sin embargo, el *endpoint* 0 no tiene descriptor al deber ser soportado por defecto por todos los dispositivos. Un *endpoint descriptor* está formado por seis campos:

Campo	Descripción
bLength	Tamaño del descriptor en bytes
bDescriptorType	-
bEndpointAddress	Número y dirección del *endpoint*
bmAttributes	Tipo de transferencias soportadas
wMaxPackSize	Máximo tamaño de paquete soportado
bInterval	Máximo retardo soportado

Tabla F.5: Campos de un *endpoint descriptor*

F.2. Enumeración

Proceso previo a la comunicación entre la aplicación y el dispositivo mediante el cual el *host* realiza una serie de peticiones de información estándar al dispositivo para aprender todo lo básico del mismo. Detalles como el identificador del dispositivo dentro del *hub* USB, información sobre el fabricante o datos sobre el descriptor del dispositivo son algunos ejemplos de los datos que se obtienen en este primer proceso.

Normalmente la enumeración es invisible para el usuario, pasando directamente a la ejecución de la GUI. Para el desarrollador, sin embargo, es vital conocer el contenido de estos primeros intercambios de datos con los que podrá crear una vía de comunicación estable entre *host* y dispositivo a través de programa.

Las especificaciones USB describen seis *device states* por los que pasa el dispositivo durante la enumeración, de los cuales cuatro (Encendido, Por defecto, Dirección y Configurado) siempre tienen lugar y dos (*Attached* y Suspendido) dependen de la aplicación:

Encendido: El usuario conecta un dispositivo a un puerto USB. Éste debe estar conectado a la ruta del *hub* en el *host*. El *hub* alimentará el puerto y el dispositivo entrará en el estado de **Encendido**. El *hub* monitoriza los niveles de voltaje en las líneas de señal de cada puerto, detectando así cuando se conecta un dispositivo. El *hub* cuenta con una resistencia *pull-down* de entre 14 y 25 kilohmios en cada una de las dos líneas de datos que tiene. Un dispositivo tiene una resistencia *pull-up* de entre 900 a 1575 ohmios en cada línea de datos. Cuando un dispositivo es conectado en un puerto, la resistencia *pull-up* pasa la tensión de la línea de datos a *high*, permitiendo al *hub* detectar que el dispositivo ha sido conectado. En este punto, el *hub* sigue proveyendo de energía, pero aun no transmite tráfico a través del USB hacia el dispositivo. Para aprender sobre el dispositivo, el *host* hace uso del *endpoint* de interrupción para comunicar eventos. Para aprender sobre un evento, el *host* envía peticiones estándar como la orden

Get_Port_Status. La respuesta dirá al *host* lo necesario para establecer la comunicación. El *hub* detecta la velocidad de transferencia de datos a la que trabaja el dispositivo comprobando el voltaje en las dos líneas de datos del USB.

Por defecto: El *hub* resetea el dispositivo dando a ambas líneas de datos el valor lógico *low*, estado característico del reset. Cuando el *hub* termina el reseteo, el dispostivo se encuentra en el estado **Por defecto**. El dispositivo está listo para responder a transferencias de control en el *endpoint* 0. El *host* envía la petición *Get_Descriptor* a la dirección 0 del dispositivo, el *endpoint* 0. Hay que tener en cuenta que la enumeración debe hacerse dispositivo a dispositivo, por lo que si varios dispositivos están conectados a la misma dirección, se deberá realizar la petición desde el *host* tantas veces como dispositivos haya.

Dirección: El *host* asigna una dirección única al dispositivo mediante el envío de la petición *Set_Address*. A partir de este punto, todas las comunicaciones se harán a la nueva dirección, la cual será válida hasta que el dispositivo sea desconectado o deje de ser detectado por el *hub*. La reconexión del dispositivo no asegura que se le vuelva a asignar la misma dirección. El host manda una nueva petición *Get_Descriptor*, pero esta vez a la nueva dirección para acceder al descriptor del dispositivo. Un descriptor es un conjunto de información que sigue una determinada estructura donde se especifica el tamaño máximo del paquete que soporta el dispositivo, todas las configuraciones que puede adoptar el dispositivo y otras informaciones básicas. En caso de haber varios dispositivos en uno o varias configuraciones, el *host* continúa realizando peticiones de información hasta saber toda la información que define al dispositivo. A continuación, el *host* realiza una petición con la que obtener el descriptor de la configuración a lo que el dispositivo responderá devolviendo además el descriptor de la o las interfaces donde se encuentra la información sobre los *endpoints* que se usarán durante la comunicación con la aplicación. En el caso de

Windows, el sistema operativo buscaría el driver que mejor se ajustase al dispositivo para manejar las comunicaciones entre este último y la aplicación. En el caso de linux, el uso de un driver no es necesario al permitir trabajar directamente en la capa de comunicación dentro de la aplicación.

Configurado: Una vez finalizada las peticiones de descriptores, el driver o aplicación determina la configuración deseada para el dispositivo mediante la petición *Set_Configuration*. El dispositivo lee la petición y se configura de la forma que ha sido requerida. Ahora el dispositivo está en el estado **Configurado** y las interfaces del mismo están habilitadas.

Adicionalmente hay dos posibles estados más que puede adoptar el dispositivo. Por un lado, el estado ***attached*** representa la ausencia de corriente en el *hub* debido a un pico de corriente o tensión. La ausencia de energía en el hub imposibilita la comunicación entre el *host* y el dispositivo. Por otro lado, un dispositivo puede entrar en estado **suspendido** al no detectar actividad en el *bus* durante un determinado periodo de tiempo.

F.3. Comunicaciones de aplicación

En esta parte entra en juego el firmware que lleva a cabo la funcionalidad para la que fue diseñado el equipo. Normalmente, los dispositivos tienen un programa embebido con el que gestionan la información recibida y dan respuesta a la misma. Es por tanto objetivo del desarrollador, crear un método de comunicación en forma de programa que permita generar los mensajes de control necesarios para hacer funcionar el dispositivo, en base a las respuestas predefinidas del mismo.

Para ello, es necesario determinar qué características cumple la comunicación USB del dispositivo y es aquí donde haremos uso de la información recopilada en el proceso de enumeración.

F.4. Tipos de transferencias

Existen cuatro tipo de transferencias distintas dentro de la comunicación USB, las cuales siguen una misma estructura general: control, *bulk*, *interrupt* e isócrono.

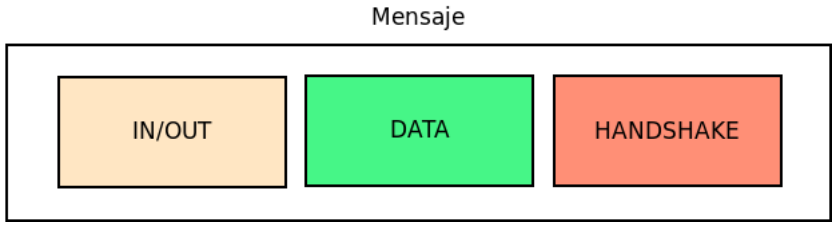


Figura F.2: Estructura de una transferencia

Control

Principalmente este tipo de transferencias son usadas para que el host adquiera la información del dispositivo necesaria para establecer una vía de comunicación. Adicionalmente puede ser empleada por el vendedor del dispositivo para comunicaciones adicionales con el host para el desarrollo de funcionalidades extra.

Todos los dispositivos deben soportar este tipo de transferencias y definirlas como tipo de transferencia por defecto. Puede darse el caso en el que un dispositivo tenga más de un *pipe* de control, aunque no es estrictamente necesario ya que el host es capaz de reservar un ancho de banda específico en función del volumen de datos a transferir.

Bulk

Tipo de transferencia empleada en comunicaciones en las que el tiempo no es un recurso crítico. Una característica importante de este tipo de comunicación es que pueden realizarse grandes transferencias de datos sin

riesgo a bloquear el bus ya que se espera a tener un espacio temporal lo suficientemente grande para realizar el envío.

Al contrario que las transferencias de control, no todos los dispositivos deben ser capaces de soportar este tipo de comunicación ya que requieren tener un endpoint que soporte transferencias *bulk* para cada sentido de la comunicación. Esto significa que es necesario tener un *pipe* dedicado exclusivamente a los mensajes tipo IN y otro a los tipo OUT.

Interrupt

Las ventajas de este tipo de transferencia aparece cuando la información a transmitir debe enviarse en una cantidad de tiempo específica. Todas aquellas aplicaciones que requieren simular transferencias en tiempo real hacen uso de este tipo de comunicación; un ejemplo claro es el teclado.

A nivel de transacciones, las transferencias *interrupt* funcionan igual que las transferencias *bulk* con la variación en el factor tiempo.

Isócrono

Comunicaciones en las que el intercambio de información es constante y en tiempo real con cierta tolerancia a errores. Ante la presencia de una conexión de este tipo en el bus, el host le da preferencia sobre otras conexiones para asegurar el intercambio de datos. También sirve para crear un sistema de preferencias dentro de un bus, permitiendo acelerar el envío de información crítica en cualquier momento.

Al igual que sus predecesores, este tipo de transferencia requiere de un *pipe* específico para cada sentido de la comunicación.