UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

# Diseño, implementación y verificación de un controlador para robots educativos

Proyecto de Fin de Carrera

para la obtención del título de
Ingeniero en Informática

Marzo de 2014

Autor: Jonán Cruz Martín

Tutor: Pedro Medina Rodríguez

## Resumen

En el presente proyecto se analiza brevemente el alcance y las aplicaciones de la Robótica Industrial y la importancia que esta disciplina de la técnica tiene en la actualidad, y se propone una plataforma moderna para asistir en la docencia de dicha disciplina en centros universitarios.

Esta nueva plataforma educativa para la enseñanza práctica de la Robótica Industrial se basa en los sistemas robóticos de la empresa Rhino Robotics Ltd., aprovechando los brazos robóticos existentes, y reemplaza la electrónica de control por un nuevo controlador más moderno y, a la vez, retrocompatible. Además del controlador, esta plataforma también aporta nuevas utilidades software más modernas e intuitivas que las del sistema original.

El trabajo consiste fundamentalmente en la recepción de órdenes desde un ordenador personal que el controlador deberá interpretar para controlar los motores del brazo robótico. El controlador en sí mismo estará implementado como un sistema embebido basado en microcontroladores. Ello implica la implementación de un protocolo de comunicación entre el ordenador personal y el microcontrolador, el diseño de un intérprete de órdenes, el diseño de la electrónica de control de motores usando PWM y puentes en H, y la implementación de técnicas de control (en concreto, control PID).

Por tanto, en este proyecto se combinan técnicas de diseño e integración de software y hardware para el desarrollo de sistemas embebidos, con técnicas de control de motores y métodos de control realimentado extraídos de la Ingeniería de Control, junto con el estudio cinemático del brazo robótico Rhino XR-4.



## Abstract

This project briefly analyzes the scope and applications of Industrial Robotics, as well as the importance that this technical discipline has gained in the past decades. In addition, it proposes a modern platform to assist in teaching this discipline in colleges and universities.

This new educational platform for the teaching of Industrial Robotics is based on the robotic systems from Rhino Robotics Ltd., using the existing robotic arms and replacing the control electronics by a newer, modern and yet backwards-compatible controller. In addition to the controller, this platform also provides new, up-to-date software utilities that are more intuitive than those provided with the old system.

The work to be done consists essentially in receiving commands from a personal computer which the controller must interpret in order to control the motors of the robotic arm. The controller itself will be implemented as an embedded system based on microcontrollers. This requires the implementation of a communication protocol between the personal computer and the microcontroller, the design of a command interpreter, the design of the electronics for motor control using PWM and H-bridges, and the implementation of control techniques (more precisely, PID control).

Hence, this project combines software and hardware design and integration techniques with motor control techniques and feedback control methods from Control Engineering, along with the kinematic analysis of the Rhino XR-4 robotic arm.

# Índice general

# Índice de figuras

# Índice de tablas

# Índice de listados de código

# 1 Introducción

## 1.1 Historia de la robótica

Desde muy antiguo el ser humano ha empleado su ingenio para la fabricación de máquinas que le asistan en el desarrollo de tareas cotidianas. De igual manera, ha sentido fascinación por el desarrollo de máquinas capaces de imitar los movimientos humanos y animales. A este tipo de máquinas los griegos dieron el nombre *automatos*, palabra de la cual deriva la española *autómata*.

Entre los siglos VIII y XV, la cultura árabe continuó con los desarrollos griegos y no sólo realizó mecanismos dedicados a imitar a los seres vivos para el mero entretenimiento, sino que usaron esos conocimientos para fabricar máquinas capaces de facilitar o automatizar tareas de la vida cotidiana (particularmente para la realeza), como son diversos dispensadores de agua para beber o lavarse. Asimismo, durante esta época se crearon otros muchos autómatas en Europa con fines lúdicos o artísticos (por ejemplo, el Gallo de Estrasburgo, de 1352).

Los siglos XV y XVI trajeron consigo el Renacimiento y, con él, diversos ingenios mecánicos construidos principalmente para aristócratas y reyes. Aquí destacan el *León mecánico* de Leonardo Da Vinci o el *Hombre de palo* de Juanelo Turriano.

A partir del siglo XVII surgen más y más autómatas, creados principalmente por artesanos de la relojería, y que representaban figuras humanas y animales, ya fuera aisladas o en grupos, llegando a formar en algunos casos aldeas mecánicas enteras. Los ingenios de esta época ya empiezan a desarrollar diversas acciones bastante complejas que imitan el comportamiento de los seres animados que representan, que es lo que cabría esperar de un robot según el concepto que nos ha dado la literatura de ciencia ficción. Así, por ejemplo, el pato que Jacques Vaucanson fabricó en 1738 era capaz de graznar, beber, comer e incluso digerir y evacuar la comida.

A finales del siglo XVIII y principios del XIX, coincidiendo con la Revolución Industrial, los diseñadores de automatismos relegan el fin lúdico a un segundo plano y se enfocan primordialmente en la creación de máquinas para la automatización de tareas de fabricación, principalmente para la industria textil. Así, se desarrollaron diversas hiladoras mecánicas (Hargraves

en 1770, Crompton en 1779) y telares mecánicos (Cartwright en 1785, Jacquard en 1801). En particular, el telar de Jacquard es especialmente llamativo, puesto que utilizaba una cinta de papel perforada a modo de programa con instrucciones que indicaban a la máquina qué patrón debía tejer. Es aquí cuando empezamos a hablar de **automatización industrial**.

En el siglo XX es cuando empieza a darse forma al concepto de la robótica que tenemos hoy en día, que ha sido mitificada gracias a las diversas obras literarias que surgieron en este siglo. En particular, el drama de Karel Čapek *R.U.R. Robots Universales Rossum* (1921) es el que introduce la palabra *robot*[1] para designar a una máquina androide[2] creada con el fin de desarrollar los trabajos físicos que hasta entonces realizaban los humanos. A partir de ahí, otros muchos autores empuñaron el término, evitando que cayera en desuso, como es el caso de la novela *Metrópolis* de Thea von Harbou (1926), posteriormente llevada al cine por su marido Fritz Lang. Otro caso, que mencionamos aparte por su relevancia, es el del escritor Isaac Asimov, autor de diferentes relatos protagonizados por robots, y que en octubre de 1945 publicó por primera vez sus archiconocidas *tres leyes de la robótica*, más tarde ampliadas a cuatro.

Sin embargo, la robótica que hemos descrito hasta ahora y la robótica que tenemos en la actualidad se corresponden más bien en poco. Hasta ahora hemos mencionado robots principalmente con forma y comportamiento humanos o animales, tal como los primeros automatismos de la historia o los robots que protagonizan las obras de ciencia ficción del siglo XX. En cambio, la robótica como área de la técnica parte más bien de la creación de máquinas para la automatización industrial, tal como sucedió en el siglo XIX.
Concretamente, el antecesor directo de los primeros robots modernos fueron los **telemanipuladores**, mecanismos principalmente en forma de brazo articulado que se utilizaban para manipular objetos a distancia, siempre dirigidos por un operador humano. Éste es el tipo de máquina que se podía ver en 1948 en el Argonne National Laboratory manipulando elementos radioactivos, en los años 60 en la industria submarina, o que se usa desde los años 70 en la industria espacial. Asimismo entran dentro de esta categoría las máquinas quirúrgicas que pueden ser controladas por un cirujano que se encuentra a miles de kilómetros.
Lo que llevó del telemanipulador surgido en los años 40 al concepto actual de robot fue la introducción de la idea de sustituir al operario humano por la programación de la máquina, para que ésta funcionase de manera autónoma, ejecutando las instrucciones de un programa predefinido, muy similar a lo que Jacquard propuso con su telar en 1801.

De esta manera se pasó de los telemanipuladores —máquinas controladas por un operario humano— a los **manipuladores robóticos** —máquinas programables capaces de funcionar de manera autónoma, con poca o ninguna supervisión, si bien pueden ser supervisadas por un operario para asegurar su correcto funcionamiento y la seguridad del entorno de trabajo y del personal que opera en los alrededores—.
Los manipuladores robóticos, al igual que los telemanipuladores, suelen ser mecanismos en forma de brazo articulado que permiten la manipulación de objetos o materiales, como su mismo nombre indica. Por este motivo, los telemanipuladores y los manipuladores robóticos se han

[1] El término *robot* procede de la raíz eslava *robota* que en diversas lenguas eslavas da lugar a las palabras *trabajo*, *trabajar*, etc.

[2] Dícese de un robot inspirado en la morfología del ser humano, pero sin imitar su aspecto físico. Si, además, imita el aspecto físico de un ser humano, se denomina humanoide.

implantado casi exclusivamente en las cadenas de producción de las fábricas[3]. Si bien la forma de los manipuladores robóticos suele ser la de un brazo articulado, existen diversas morfologías de robot utilizadas en la automatización industrial, de manera que, en general, a estas máquinas se las denomina **robots industriales** y dan lugar al área de la **robótica industrial**.
Sin embargo, los manipuladores robóticos y, en general, los robots industriales, no son el único tipo de robot que existe en la actualidad. También existen otros muchos tipos de máquinas programables capaces de realizar tareas muy diversas, no solamente la manipulación de objetos y materiales. En general, se suelen clasificar los robots en dos categorías: **robots industriales** y **robots de servicio**. Los robots industriales son aquellos que se utilizan en la industria para las tareas de fabricación y manipulación de mercancías. Por su parte, los robots de servicio engloban muy diversos tipos de robots, la mayoría robots móviles, que se suelen utilizar para la asistencia a humanos (p.ej. en el transporte de mercancías o la asistencia a personas mayores), la exploración de zonas peligrosas o inaccesibles (desactivación de bombas, exploración submarina, rescate en zonas de montaña de difícil acceso, etc.), entre otras muchas áreas de aplicación.

## 1.2 Definición de la robótica

Ya en la introducción histórica que hemos hecho previamente se empieza a ver una definición más o menos precisa de qué es la robótica y qué es un robot. Si bien hay muchas definiciones de diferentes autores, cada una con matices particulares[4], todas ellas tienen en común los aspectos recogidos en el siguiente párrafo [2]:

Un robot es un dispositivo mecánico programable[5] que utiliza sensores para guiar uno o más efectores finales[6] mediante movimientos planificados a través del espacio de trabajo con el fin de manipular objetos físicos.

## 1.3 Antecedentes y motivación para el desarrollo de la robótica industrial: automatización dura vs automatización blanda

La creación de máquinas programables, o robots, es muy interesante desde el punto de vista técnico con el mero fin de la investigación y el desarrollo. Sin embargo, ¿tiene esto alguna

[3] Si bien los manipuladores se usan sobre todo en fábricas, también existen otros lugares donde se utilizan estas tecnologías, por ejemplo, para la manipulación de mercancías peligrosas o radiactivas en centrales nucleares, o en los transbordadores espaciales para las reparaciones y el transporte de mercancías a las estaciones espaciales.

[4] Para ver una discusión de las definiciones de diferentes autores y organizaciones, véase [1].

[5] Es decir, se puede crear un programa cuyas instrucciones controlan los movimientos del robot.

[6] El efector final es la herramienta que se acopla al extremo del manipulador robótico y que permite al robot realizar la acción para la que está programado: agarrar objetos con una pinza, soldar con un soldador, cortar mediante un chorro de agua o un láser, etc.

aplicación práctica? ¿Por qué se utilizan robots en la automatización industrial y no otro tipo de máquinas?

Más allá del interés teórico de la robótica y de lo que ésta puede contribuir al desarrollo de la tecnología, estas y otras preguntas acerca de su utilidad práctica surgen de manera natural y, si se busca respuesta, se puede llegar a muchas aplicaciones útiles. Pero, antes de ver el porqué de diseñar y construir robots, debemos observar primero de qué forma se automatizan las tareas en la industria.

Desde la Revolución Industrial hasta el momento previo al surgimiento de la robótica industrial, la industria ha desarrollado máquinas especializadas en desarrollar una tarea concreta de manera automatizada. Éste es el caso de los telares mencionados anteriormente y, más concretamente, del telar de Jacquard, que, si bien era "programable" en el sentido de que el patrón a tejer se indicaba mediante instrucciones perforadas en una tarjeta de papel, esta máquina sólo servía para tejer.
Asimismo, con la aparición de las cadenas de ensamblado para la producción en masa, introducidas por la Compañía de Motores Ford en 1905, cada etapa y tarea de la producción en cadena se ha conseguido automatizar y agilizar mediante máquinas especializadas en realizar una tarea concreta, y diseñadas para alcanzar un alto volumen de producción de partes mecánicas y eléctricas.

Sin embargo, la utilización de máquinas especializadas, comúnmente denominada **automatización dura** (del inglés *hard automation*), presenta ciertas desventajas.
En primer lugar, estas máquinas están especializadas en la fabricación de piezas muy concretas, de manera que, cada vez que cambia la pieza a fabricar, es necesario apagar las máquinas y hacer modificaciones en ellas. En el mejor de los casos, sólo habrá que hacer algunas modificaciones para adaptar la máquina a la nueva generación de piezas, pero, si las piezas han cambiado mucho, será necesario descartar las máquinas y diseñar nuevas máquinas para la fabricación de las nuevas piezas.
En segundo lugar, la utilización de máquinas especializadas tiene un coste elevado, principalmente por dos motivos. El primero de ellos que, en general, habrá que diseñar y fabricar máquinas específicas para una actividad industrial concreta. Si esa actividad industrial es desarrollada por pocas empresas, el coste del desarrollo será asumido por una sola empresa o por unas pocas empresas, con lo que el coste para cada una será mayor que si muchas empresas fuesen a fabricar o adquirir dichas máquinas. El segundo motivo por el que la automatización dura es cara es que requiere modificar o incluso rediseñar por completo las máquinas y sustituirlas por nuevas máquinas, con lo que hay que hacer un nuevo desembolso cada vez que se necesita reemplazar una máquina.
En resumen, la automatización dura es costosa y poco flexible, ya que las máquinas no son capaces de realizar diversas tareas, sino que están especializadas. Por contra, puesto que las máquinas están especializadas, suelen ser más eficientes y realizan un mayor volumen de trabajo, de manera que producen más.

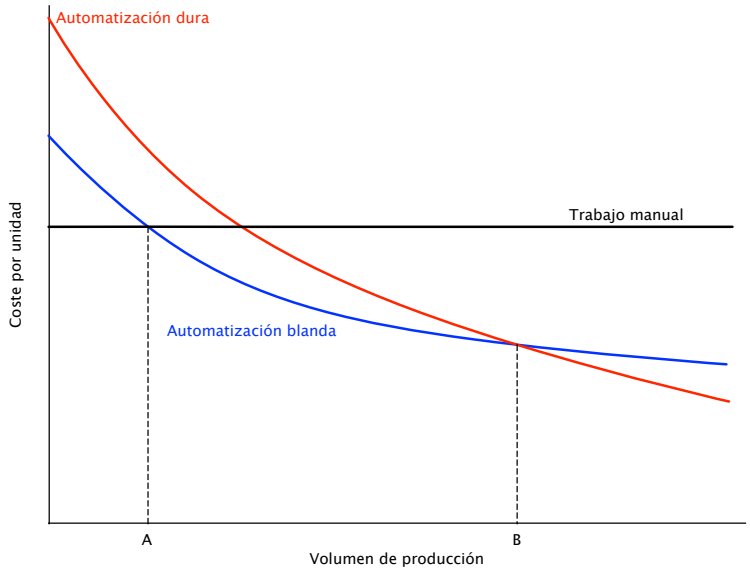
En oposición a la automatización dura surge la **automatización blanda** (del inglés *soft automation*), que introduce formas más flexibles de automatización para el ciclo de fabricación de nuevos productos. Dentro de la automatización blanda es donde se encuentran los robots industriales, puesto que son máquinas más flexibles que admiten ser utilizadas para propósitos

diferentes, para lo cual basta con adaptar el efector final, el programa que controla el robot y el entorno de trabajo del robot.
De esta manera, un mismo robot se podría utilizar tanto para pintar las aspas de un aerogenerador como para soldar piezas a su estructura. Sólo sería necesario cambiar el programa, la herramienta (efector final) y la montura (el entorno de trabajo). Así, para pintar el aspa de un aerogenerador, se puede dotar al robot de un spray de pintura como efector final, montar el robot en un carril deslizante y cambiar el programa[7]. Por su parte, se podría utilizar el mismo modelo de robot para realizar tareas de soldadura con tan sólo acoplarle un soldador como efector final, utilizar otro montaje para el entorno de trabajo del robot y cambiarle la programación[8].
De esta forma se reducen costes, puesto que un mismo robot se puede usar para diferentes tareas (siempre y cuando las características del robot sean adecuadas para todas ellas), por lo que no es necesario reemplazar los robots con tanta frecuencia como las máquinas de automatización dura. Además, puesto que un mismo robot puede ser utilizado para diversas tareas, hay un mayor mercado y los costes de producción de los robots no serán tan elevados, por lo que tendrán un coste más reducido que una máquina especializada.
Aparte de la flexibilidad que presentan los robots, también tienen como ventaja su alta precisión y repetibilidad, algo que también proporcionan las máquinas especializadas, pero no los operarios humanos. Adicionalmente, la utilización de robots evita ciertos riesgos para los operarios humanos (siguiendo con el ejemplo anterior de la pintura del aspa de un aerogenerador, el operario no estaría expuesto a la toxicidad de las pinturas utilizadas, que suelen ser de elevada toxicidad).

La elección entre un tipo u otro de automatización depende principalmente de las necesidades de la aplicación, especialmente en cuanto a flexibilidad y costes. Si, por ejemplo, se necesita gran adaptabilidad ante nuevos diseños de piezas, podría ser más conveniente la automatización blanda, ya que ésta evitaría reemplazar la maquinaria con cada nuevo rediseño de las piezas. Por el contrario, si hace falta una gran rapidez y se permite sacrificar flexibilidad, quizá convenga más la automatización dura.
Sin embargo, aquí también entra en juego otro factor, que es el volumen de producción requerido [2]. Si el volumen de producción es muy elevado, la automatización dura se vuelve eficiente, tanto en términos de producción como de coste. En cambio, si el volumen de producción es elevado, pero no demasiado, la automatización blanda es más eficiente. Por último, si el volumen de producción es bajo, el trabajo manual resultaría más eficiente en la relación producción-coste que cualquiera de los dos tipos de automatización.
Esto mismo se puede ver en la Figura 1.1 (pág. 6). Según datos citados en [2], el trabajo manual es de menor coste para volúmenes de producción pequeños (en el intervalo $[0,\ A]$). A medida que el volumen de producción aumenta, los robots presentan una mayor eficiencia producción-coste frente al trabajo manual y la automatización dura (cuando el volumen de producción se encuentra en el intervalo $[A,\ B]$). Por último, cuando el volumen de producción excede del umbral $B$, los robots dejan de ser la mejor solución según la relación producción-coste y la automatización dura se vuelve la mejor opción (intervalo $[B,\ \infty)$).

[7] Véase el siguiente vídeo sobre el uso de robots para pintar el aspa de un aerogenerador: http://youtu.be/xLyfegzK6AU.
[8] Véase el siguiente vídeo sobre el uso de robots para la soldadura: http://youtu.be/HUU3HdxOqZs.

Figura 1.1: Coste de los diferentes tipos de automatización frente al volumen de producción




## 1.4 Robótica industrial: áreas de aplicación, importancia y disciplinas

### 1.4.1 Áreas de aplicación e importancia de la robótica industrial

Como ya se ha comentado previamente, la robótica industrial es el área de la robótica que se encarga del diseño de robots para la automatización de tareas industriales y la manipulación de materiales peligrosos en entornos industriales. Tareas muy comunes de estos robots son:

- **El paletizado.** Dada la gran capacidad de carga de algunos robots, se puede ver cómo dichos robots son usados para apilar y paletizar mercancías pesadas.
- **El ensamblado.** Es muy común utilizar robots para tareas de colocación de piezas, por ejemplo, la colocación de componentes electrónicos en una placa de circuito impreso o la instalación de un parabrisas en un coche.
- **La soldadura.** En la fabricación de maquinaria, con mucha frecuencia se emplean robots para la soldadura de piezas con gran precisión.
- **El corte de materiales.** También es una aplicación bastante común utilizar robots para realizar corte de materiales por chorro de agua o por láser.
- **La colocación de productos para el empaquetado.** Normalmente se usan cintas transportadoras, guías y otros mecanismos para organizar los productos para el empaquetado, pero hay situaciones en las que esto no se puede realizar mediante estos mecanismos, sino que hacen falta robots especialmente diseñados para ello[9].

[9]Consíderese, por ejemplo, el empaquetado de tortitas, donde cada paquete debe contener un número determi-

Entre las ventajas de la implantación de robots se incluyen las siguientes:

- Los robots industriales permiten ganar en eficiencia, puesto que se aumenta el volumen de producción frente al trabajo manual. Esto conduce a la reducción de los costes laborales.
- Los robots proporcionan una mayor precisión en sus movimientos que los trabajadores humanos, lo cual es especialmente importante en la fabricación de ciertos objetos, en tareas de corte, soldadura, etc.
- El uso de robots reduce los riesgos para los operarios humanos en tareas peligrosas, como la manipulación de materiales tóxicos, explosivos o radiactivos.
- Los robots proporcionan mejores condiciones de limpieza e higiene, algo importante en la industria alimentaria o en la industria de los semiconductores[10].

Todas estas aplicaciones y ventajas del uso de robots han llevado a que su implantación en la industria haya crecido a pasos agigantados desde los inicios de la robótica hasta la actualidad y, con la extensión de la robótica, se requieren profesionales capaces de continuar los desarrollos en este sector y de llevar a cabo la implantación de estas tecnologías en entornos industriales.

### 1.4.2 Disciplinas de la robótica industrial

Como casi cualquier área de la ciencia y la técnica hoy en día, la robótica es un área multidisciplinar que se nutre de los conocimientos sobre mecánica, electricidad y electrónica, e informática, y de los nuevos desarrollos en estas áreas. Un sistema robótico (de robótica industrial) se compone de un manipulador robótico, de un armario de control y de una parte software, que es la que controla los movimientos del manipulador. En estos tres elementos vemos presentes los conocimientos de las áreas mencionadas.

El manipulador robótico, que suele ser un brazo articulado, es la parte mecánica, compuesta por mecanismos y engranajes, y que contiene los motores que mueven las articulaciones del robot. En general hay tres tipos de tecnologías empleadas para los actuadores de los robots industriales, tanto para los motores de las articulaciones como para el efector final (principalmente pinzas): hidráulica, neumática y eléctrica. En la actualidad, la mayor parte de los motores empleados para las articulaciones de robots suelen ser eléctricos, debido a su gran precisión, rapidez de movimiento y facilidad de control, y a que su capacidad de carga suele ser suficiente para la mayor parte de las aplicaciones. En caso de necesitar una gran capacidad de carga (en

nado de tortitas. Esta tarea no se puede realizar mediante los mecanismos tradicionales mencionados; o lo hace un operario humano, o lo hace un robot especialmente diseñado para ello (robots dotados con una bomba de succión para recoger las tortitas): http://youtu.be/wg8YYuLLoM0. El mismo principio se puede aplicar a otros productos: http://youtu.be/8unkXTtfEBQ.

[10]En la producción de semiconductores la limpieza es vital, por lo que el proceso se desarrolla en "salas limpias", donde gran parte del trabajo está automatizado y los operarios humanos deben llevar trajes especiales para evitar contaminar las salas a las que tienen acceso, como se puede ver en el siguiente vídeo de presentación de una fábrica de Intel: http://youtu.be/PecKlm6VutU.

robots muy grandes o que deben ser capaces de levantar mucho peso), se utilizan actuadores hidráulicos, que, si bien son más lentos que los eléctricos, tienen una mayor capacidad de carga. Los actuadores neumáticos, por su parte, son cada vez menos usados (como mucho se usan en las pinzas), puesto que tienen una precisión baja en comparación con las otras tecnologías y los actuadores eléctricos se pueden usar para los mismos fines y presentan mayores ventajas.

El armario de control es donde se encuentran los dispositivos eléctricos y electrónicos encargados de proporcionar la alimentación eléctrica al robot y de controlar sus motores. Suele estar formado por una instalación eléctrica que alimenta los actuadores (principalmente motores eléctricos) y por un conjunto de dispositivos electrónicos con microcontroladores o PCs industriales y electrónica específica para el control de los motores[11] y la planificación de movimientos de acuerdo a la programación del robot.

Por último, el robot necesita un software que se encargue de leer la información sensorial e indicar los movimientos que debe realizar el robot para ejecutar sus tareas satisfactoriamente. Esta parte software es la que se ejecuta en los microcontroladores o PCs industriales instalados en el armario de control.

Por tanto, la robótica combina los conocimientos de la Ingeniería Mecánica (diseño de máquinas, hidráulica, neumática y máquinas eléctricas), de la Ingeniería Eléctrica y Electrónica (máquinas eléctricas, instalaciones eléctricas para la alimentación, electrónica de potencia, microcontroladores e ingeniería de control) y de la Ingeniería Informática (programación, ingeniería de control, arquitecturas para computadores industriales, planificación de movimientos e inteligencia artificial[12]).

## 1.5 Robótica educativa

Por **robótica educativa** entendemos el conjunto de métodos y prácticas empleados para la enseñanza de la robótica y, en particular, de la robótica industrial, con el fin de formar profesionales capaces de trabajar de forma competitiva en el desarrollo de la robótica y en la implantación de las tecnologías robóticas en la industria. Por lo comentado en la Sección 1.4.1 (pág. 6), es necesario formar profesionales de la robótica, para que con cada generación se renueve el equipo profesional del sector y se satisfaga la demanda de dichos profesionales.

Puesto que la robótica es un área eminentemente práctica, si bien requiere unos sólidos fundamentos teóricos, no es suficiente con conocer la teoría. La robótica educativa contempla, por tanto, la enseñanza práctica de la robótica industrial. Para ello, de forma paralela a la exposición de la teoría, se suelen desarrollar tareas prácticas de programación de robots y diseño de células robotizadas[13]. Esto se suele llevar a cabo mediante **robots educativos**, es decir, robots

[11] Por *electrónica específica* nos referimos a la microelectrónica y electrónica de potencia necesaria para el control de motores, con dispositivos como puentes en H, generadores de PWM, etc.

[12] La inteligencia artificial es especialmente en las aplicaciones más complejas donde se requiere que el robot sea capaz de moverse en un entorno con obstáculos, a veces móviles, o donde se necesitan técnicas de visión artificial para el reconocimiento de objetos, por ejemplo.

[13] Se denomina célula robotizada a la instalación de un robot como parte de una cadena de producción para desarrollar un trabajo conjunto con otros robots u otra maquinaria de dicha cadena de producción, donde el trabajo

diseñados específicamente para la enseñanza de la robótica.

Los robots educativos para la enseñanza de la robótica industrial suelen ser manipuladores robóticos de dimensiones reducidas, con un pequeño controlador del tamaño de una caja y que ejerce las funciones del armario de control, y que puede ser programado mediante un conjunto de instrucciones especializado para el control del robot. Además, en algunos casos este controlador puede llevar consigo una paleta de programación, una especie de mando de control pensado para manejar el robot sin necesidad de programarlo y para enseñarle movimientos o puntos del espacio que deba guardar en su memoria, de forma que dichos puntos puedan ser usados posteriormente en la programación.

## 1.6 Enseñanza de la robótica industrial en el Departamento de Informática y Sistemas

En la antigua Facultad de Informática de la Universidad de Las Palmas de Gran Canaria se contemplaba la enseñanza de la robótica industrial en la asignatura optativa *Robótica*. En ella se exponían los conocimientos teóricos fundamentales de la robótica industrial y se proporcionaba enseñanza práctica sobre la programación y el control de robots industriales mediante un programa de prácticas con robots educativos de las empresas *Rhino Robotics, Ltd.*[14] e *Intelitek*[15].

Entre el equipamiento disponible para la enseñanza se encuentran un robot Rhino XR-4, und robot Rhino SCARA y un robot SCORBOT-ER 4u de Intelitek, cada uno con su respectivo controlador (en el caso de los robots Rhino, dos controladores Rhino Mark IV). Sin embargo, el equipamiento disponible de la empresa Rhino Robotics data de los años 80 y, si bien los brazos robóticos se mantienen en correcto funcionamiento, el uso y el paso de los años ha causado el deterioro de los controladores. A pesar de que uno de ellos aún sigue operativo, el otro presenta un mal funcionamiento que impide su utilización.

Aunque el robot SCROBOT-ER 4u es bastante más moderno que los robots de Rhino Robotics, los robots de Rhino Robotics siguen siendo los principales protagonistas de la enseñanza práctica de la asignatura *Robótica*. Estos robots, en particular los modelos XR-3 y XR-4 de Rhino Robotics, han gozado de gran aceptación en los diferentes centros de enseñanza superior de varios países, hasta el punto de ser considerados como caso de estudio en algunos libros de texto (véase, por ejemplo, [2]). Algunos motivos para que esto sea así son sus especificaciones abiertas[16], su gran cantidad de accesorios[17] y su programabilidad (tanto por su facilidad de pro-

---

de cada componente de la célula suele requerir estar sincronizado con el resto de componentes para el correcto funcionamiento del conjunto.

[14] http://www.rhinorobotics.com/

[15] http://www.intelitek.com/

[16] Los robots de Rhino Robotics van acompañados de una buena documentación que, entre otras cosas, detalla las dimensiones del brazo robótico, de manera que se pueda analizar su cinemática, algo que es de gran ayuda para la asimilación de los conceptos teóricos de la robótica.

[17] La empresa Rhino Robotics proporcionaba también otros accesorios, como varios efectores finales (aparte de la pinza), una cinta transportadora y una mesa giratoria, además de diferentes sensores. Todo esto permitía montar

gramación como por su versatilidad).
Aparte de todo esto, disponer de varios robots utilizables posibilita el montaje de una célula robotizada en la que dos o más robots deban colaborar para realizar una tarea. De hecho, uno de los ejercicios prácticos de la asignatura *Robótica* consiste en montar una célula robotizada donde el robot SCORBOT-ER 4u deposita unas piezas en una cinta transportadora y la cinta transportadora conduce las piezas al otro extremo, donde el robot XR-4 toma las piezas y realiza una tarea de paletizado con una mesa giratoria. De esta manera, se crea un currículo de prácticas más completo donde los alumnos pueden entrenarse en la programación de una célula robotizada al completo (no sólo de un robot individualmente).
En definitiva, los sistemas robóticos de Rhino Robotics presentan unas características únicas que otros sistemas de robótica educativa no han igualado hasta el momento y, a pesar de su edad, siguen siendo una parte clave de la docencia de la robótica. Por ello, alargar la vida útil de estos sistemas es altamente deseable, tanto para poder seguir aprovechando esas características que ningún otro sistema proporciona, como para no tener que realizar una inversión innecesaria en adquirir sistemas que, aunque más modernos, quizá no se adapten plenamente a los requisitos de la docencia.

El sistema robótico básico necesario para comenzar la realización de las prácticas de *Robótica* consta de un brazo robótico (Rhino XR-4 o Rhino SCARA), un controlador Rhino Mark IV y una paleta de programación (que acompaña al controlador). La interacción del usuario con el controlador para la programación del robot se realiza o a través de la paleta de programación o a través de un programa de utilidad en línea de comandos para el sistema MS-DOS que permite enviar instrucciones individuales al controlador, así como programas completos. Puesto que el único componente que presenta un mal funcionamiento es el controlador, se plantea el diseño y desarrollo de un nuevo controlador que sustituya al antiguo controlador Mark IV de Rhino Robotics, para así poder seguir empleando los brazos robóticos de Rhino Robotics en la enseñanza de la robótica. Además, dado que la interfaz entre el usuario y el controlador sólo está disponible bajo emulación del sistema MS-DOS y es incómoda y compleja de utilizar, se plantea también el desarrollo de una nueva aplicación de escritorio con interfaz gráfica que permita al usuario interactuar con el controlador de manera más cómoda y usando un sistema operativo moderno.

La renovación del sistema robótico será, por tanto, el propósito general del presente trabajo. Los objetivos concretos se detallan en la sección que viene a continuación.

## 1.7 Objetivos

La tarea principal del presente trabajo es el diseño y desarrollo de un nuevo controlador robótico para sustituir al antiguo controlador Rhino Mark IV. Este nuevo controlador estará implementado con tecnologías más modernas y será, idealmente, más compacto que el controlador actualmente en uso. Además, se desea sustituir el antiguo controlador por el nuevo en el control de los mismos brazos robóticos y manteniendo los mismos contenidos de las prácticas

una célula robotizada bastante completa con la que mejorar la variedad y profundidad de los ejercicios prácticos a resolver por los alumnos.

de *Robótica* y la misma forma de programar los robots, por lo que se requiere que el nuevo controlador sea compatible con el antiguo. Esto significa que deberá soportar todo el conjunto de instrucciones del controlador Mark IV, si bien se permitirá que algunas instrucciones de uso poco frecuente o que no afectan al control del movimiento del robot no sean implementadas[18]. Sin embargo, todas las instrucciones de movimiento del robot deberán estar soportadas y comportarse de la misma forma que en el antiguo controlador.

Adicionalmente, puesto que la programación del robot con el sistema actual se realiza enviando comandos individuales o programas enteros al controlador desde un PC a través del puerto serie (mediante el protocolo RS-232C), pero sólo hay disponible para ello una utilidad en línea de comandos bajo el sistema MS-DOS, también deberá desarrollarse una aplicación de escritorio con interfaz gráfica que permita una interacción más cómoda con el controlador. Esta aplicación deberá poder ejecutarse de manera nativa (sin emulación) bajo el sistema operativo Windows. Eventualmente, si fuera posible, una aplicación multiplataforma capaz de ejecutarse en otros sistemas operativos sería de gran conveniencia.

Estos son los objetivos del presente trabajo. A continuación se expondrá la metodología a utilizar (Capítulo 2, pág. 13) y luego se procederá a hacer un análisis más detallado de los requisitos, los recursos necesarios y el plan de trabajo (Capítulo 3, pág. 15).

# Referencias

[1] Antonio Barrientos, Luis Felipe Peñín, Carlos Balaguer, and Rafael Aracil. *Fundamentos de Robótica*. McGraw-Hill, 1997.

[2] Robert Joseph Schilling. *Fundamentals of Robotics: Analysis and Control*. Prentice Hall, 1990.

[3] International Federation of Robotics. *History of Industrial Robotics*. http://www.ifr.org/uploads/media/History_of_Industrial_Robots_online_brochure_by_IFR_2012.pdf.

[18] Al menos no en los primeros desarrollos de este nuevo controlador, para así centrarse en las funcionalidades realmente importantes.

# 2 Metodología

El ámbito del presente proyecto se centra en el desarrollo de una plataforma hardware/-software basada en microcontroladores, así como de una aplicación de escritorio con interfaz gráfica para facilitar la interacción del usuario con dicha plataforma.

Por la propia naturaleza del proyecto de la plataforma basada en microcontroladores, que requiere la integración de hardware con software, en algunos casos diseñando hardware específico, será necesario realizar una gran tarea de investigación y experimentación, estudiando la forma de uso de los componentes hardware, de la arquitectura de los microcontroladores, y de las tecnologías que estos proporcionan y que puedan ser aprovechadas para alcanzar los objetivos del proyecto. Por este motivo, las metodologías de desarrollo de software surgidas en el ámbito de la Ingeniería del Software no se adecúan totalmente a las características de este proyecto. En su lugar, un enfoque más apropiado consiste en dividir el proyecto en diversos experimentos, donde se estudiará y probará los diferentes componentes, tecnologías y enfoques de diseño posibles, para así encontrar el enfoque más adecuado.
Si bien los enfoques de desarrollo de la Ingeniería del Software no se adecúan a este trabajo, el enfoque de desarrollo "basado en experimentos" que se propone aquí se asimila bastante a un desarrollo ágil iterativo, en el que se realizan sucesivas iteraciones de corta duración sobre el producto, donde en cada iteración se puede llevar a cabo tanto tareas de análisis, como de diseño o de implementación. La diferencia entre estos dos enfoques radica en que, en el enfoque "basado en experimentos" que se seguirá en este proyecto, un experimento no se corresponde unívocamente con una iteración, sino que realmente un experimento puede consistir de varias iteraciones (o de una única iteración, si el experimento es muy sencillo).

La ventaja de un desarrollo basado en experimentos (o de un desarrollo ágil, en el caso del software) es que permite el desarrollo de un prototipo que se irá ampliando (añadiendo funcionalidades) y refinando (corrigiendo errores) de manera sucesiva a lo largo de muchas iteraciones de corta duración, permitiendo, por tanto, (1) que el desarrollo del proyecto se adapte mucho más fácilmente a los posibles cambios en los requisitos o dificultades que surjan durante el desarrollo, y que puedan conducir a cambios en el diseño o en las herramientas y tecnologías que deba utilizarse, y (2) que el prototipo en desarrollo presente avances visibles de manera más rápida, dando una mayor sensación de progreso y permitiendo utilizar el prototipo para la

captura de nuevos requisitos (o la modificación de los ya obtenidos) y para la comprobación de la aceptación por parte de los usuarios.

Debido a los beneficios que la metodología ágil presenta, también se plantea utilizar una metodología ágil para el desarrollo de la aplicación de escritorio[1]. De esta forma, se podrá desarrollar un prototipo de la aplicación de escritorio que presente funcionalidades desde un estadio temprano del desarrollo, pudiendo empezar a ser utilizado en el laboratorio de Robótica casi desde el principio, y permitiendo al mismo tiempo seguir con la captura de requisitos y las pruebas, para la continuación del desarrollo.

Por tanto, para el desarrollo de ambas aplicaciones se plantea el uso de una metodología de desarrollo ágil donde un prototipo se irá ampliando (añadiendo funcionalidades) y refinando (corrigiendo errores) a lo largo de muchas iteraciones sucesivas y de corta duración. Al comienzo de cada iteración se realizará un diseño únicamente de la parte a implementar en la iteración y de aquellas partes del software (o hardware) directamente relacionadas o dependientes. Todas aquellas partes que no intervengan ni estén relacionadas con las partes a diseñar e implementar, no se contemplarán en el diseño de esa iteración. Esto permitirá centrarse en una pequeña porción del sistema y en diseñar e implementar esa porción de manera correcta y eficaz. Lo importante al final de cada iteración será tener una porción del sistema bien diseñada, implementada y verificada (es decir, que se haya comprobado que su funcionalidad es realizada correctamente). Una vez finalizada la iteración, se continuará con la siguiente porción del sistema a diseñar e implementar. Ante todo, se procurará que estas partes diferentes del sistema no sean interdependientes, para que el diseño de una iteración previa no se vea obligado a cambiar por cambios en los requisitos en la siguiente iteración o requisitos no tenidos en cuenta. Sin embargo, en caso de suceder esto, simplemente se modificará el diseño debidamente y se refactorizará la implementación para adaptarse a los nuevos requisitos.

Finalmente es necesario apuntar que, durante todo el proceso de desarrollo, se hará mucho énfasis en la verificación continua de los productos (hardware o software). De esta manera, realizando la tarea de verificación juntamente con el diseño y la implementación, se conseguirá mejorar la robustez del sistema final y encontrar cualquier posible fallo con mayor prontitud, de forma que el menor número posible de errores permanezca en la implementación durante muchas iteraciones, lo cual dificultaría encontrar y corregir dichos errores.
Además, durante el proceso de desarrollo también se dará importancia a la producción de documentación acerca del diseño y la implementación. De esta manera, la tarea de documentación no necesitará ser realizada a posteriori y esto también permitirá que cualquier persona pueda incorporarse al proceso de diseño e implementación una vez éste haya comenzado.

# Referencias

[1] Roger S. Pressman. *Ingeniería del software: un enfoque práctico*. McGraw-Hill, 6 edition, 2006.

[1] En este caso, al tratarse de un sistema puramente software, las metodologías de desarrollo ágil de la Ingeniería del Software se pueden aplicar sin ningún problema.

# 3 Análisis y planificación

## 3.1 Definición de los subsistemas a analizar

Dentro del ámbito de este proyecto se puede distinguir dos sistemas a diseñar:

1. Un controlador para robots educativos basado en microcontroladores, que se compondrá de una parte hardware (que llamaremos *RhinoChip*) y una parte software (que llamaremos *RhinoChip OS*).

2. Una aplicación de escritorio que servirá de interfaz para la comunicación entre el usuario y el controlador (a la que denominaremos *MarkCommander*).

Por tanto, vemos que hay tres subsistemas a analizar, diseñar e implementar. En primer lugar, será necesario analizar los requisitos en cuanto al hardware del controlador RhinoChip que reemplazará al controlador Mark IV de Rhino Robotics. En segundo lugar, habrá que analizar los requisitos del software (firmware) que se ejecutará en el controlador y que será el que implemente las funcionalidades de dicho controlador. Por último, se deberá analizar los requisitos del software de aplicación que utilizarán los usuarios para comunicarse con el controlador RhinoChip desde un ordenador personal.

## 3.2 Estructura y funcionamiento del controlador Mark IV

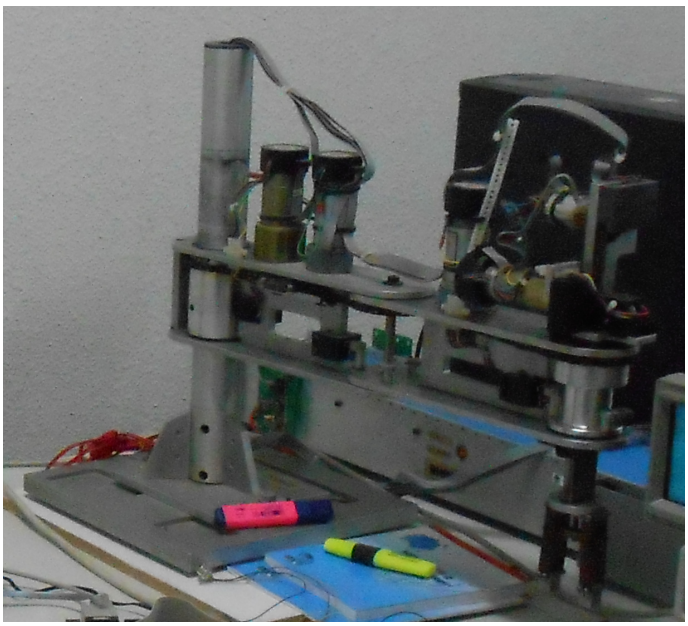
El controlador Mark IV de Rhino Robotics (Figura 3.1, pág. 16) es un controlador de 8 ejes[1] que proporciona un conjunto de instrucciones de bajo nivel para su manejo. Los 8 puertos de control de motores admiten motores con codificador de posición incremental, a los cuales se puede conectar los robots XR-4 (5 ejes más efector final) o SCARA (4 ejes más efector final), mostrados en la Figura 3.2 (pág. 16). Además, el controlador Mark IV dispone de 2

[1] Es decir, con capacidad para controlar 8 motores en bucle cerrado.

Figura 3.1: Controlador Mark IV de Rhino Robotics


Figura 3.2: Brazos robóticos de Rhino Robotics


(a) Robot Rhino XR-4



(b) Robot Rhino SCARA

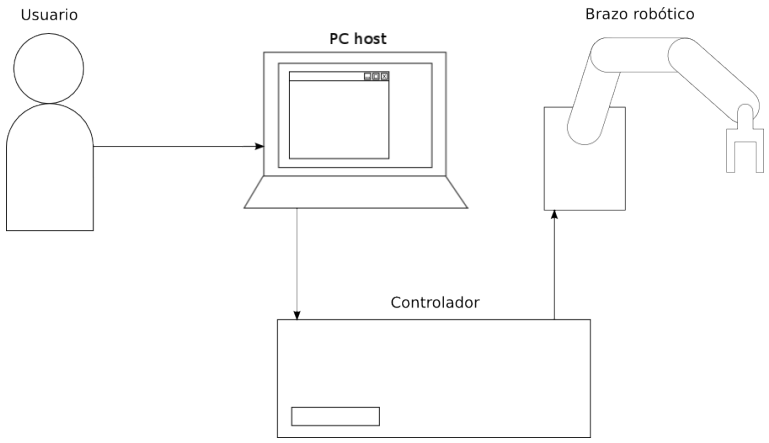
puertos auxiliares para control de motores sin codificador de posición, 8 puertos de entrada digital y 8 puertos de salida digital independientes, y 8 líneas de entrada digital adicionales cuyo valor se establece manualmente mediante un interruptor. Aparte de esto, el controlador dispone de un puerto serie para la comunicación con un PC host y una conexión para una paleta de programación. Finalmente, hay que mencionar que el controlador dispone de una memoria EEPROM para el almacenamiento de un programa de usuario. En la Tabla 3.1 se puede ver un resumen de todos estos datos técnicos del controlador Mark IV.

Tabla 3.1: Especificaciones técnicas del controlador Mark IV

| | |
|---|---|
| Puertos de control de motores | • 8 puertos para motores con codificador de posición incremental<br>• 2 puertos auxiliares de 20 V (DC) para control de motores sin codificador |
| Entrada/salida | • 8 puertos de entrada optoaislados con limitador de corriente<br>• 8 puertos de salida optoaislados con limitador de corriente<br>• 8 líneas de entrada con interruptor |
| Conectividad | • Puerto RS232-C para la comunicación con el PC host<br>• Puerto RS232-C para la conexión de una paleta de programación |
| Almacenamiento y programabilidad | • Repertorio de instrucciones con más de 80 instrucciones<br>• Memoria EEPROM para el almacenamiento de la configuración del controlador y de un programa de usuario |

El modo de uso más frecuente del controlador Mark IV se ilustra en la Figura 3.3 (pág. 18). En dicha figura puede verse que el controlador Mark IV actúa como interfaz entre el usuario y el brazo robótico. El usuario, desde un PC host, envía órdenes al controlador, que le indican

Figura 3.3: Esquema de funcionamiento del controlador Mark IV




a éste qué acciones debe realizar para controlar el brazo robótico, o que le solicitan información acerca del estado del sistema. Estas órdenes pertenecen al repertorio de instrucciones del controlador, mencionado en la Tabla 3.1 (pág. 17), y se envían desde un PC host mediante una conexión RS232-C al controlador. Para llevar a cabo esta comunicación, el usuario puede emplear cualquier software de terminal, como HyperTerminal o Termite, pero Rhino Robotics proporciona, junto con el hardware del sistema, una pareja de utilidades software para la interacción con el controlador: MK4UTIL y MK4CONFIG. Ambas son utilidades en línea de comandos para el entorno MS-DOS[2]. La primera de ellas, MK4CONFIG, se conecta al controlador Mark IV y permite configurarlo de manera que pueda empezar a utilizarse. La segunda, MK4UTIL, es la utilidad software que permite al usuario conectarse al controlador desde un PC host para enviarle órdenes mediante una shell interactiva, además de posibilitar la escritura y ejecución de programas en un lenguaje de más alto nivel, RoboTalk, con una sintaxis inspirada en el lenguaje BASIC[3].

Otro modo de uso del controlador consiste en la escritura de programas —ya sea mediante el repertorio de instrucciones de bajo nivel o mediante el lenguaje RoboTalk— y la ejecución de dichos programas desde el PC host. Este modo de uso es lo que se conoce como programación off-line, y de nuevo seguiría el esquema de la Figura 3.3, sólo que ahora el usuario no emplea una shell interactiva, sino que escribe un programa sin conectarse al robot y controlarlo en tiempo real (es decir, off-line), y luego ordena al controlador que ejecute dicho programa.

Además, existe un último modo de utilizar el controlador Mark IV: mediante una paleta de programación, al igual que cualquier robot industrial disponible en el mercado. La paleta de programación del controlador Mark IV (Figura 3.4, pág. 19) es un mando de control que se conecta al controlador y que permite realizar una amplia variedad de acciones con el robot, como el movimiento del brazo en tiempo real o el almacenamiento de puntos del espacio para ser usados posteriormente en un programa. La paleta también permite la grabación de programas sencillos, que son almacenados para poder ser ejecutados posteriormente. Este modo de

[2]El sistema MS-DOS está desfasado y actualmente se usan versiones de Windows iguales o superiores a Windows XP. Debido a que estas utilidades no son compatibles con el entorno actual, es necesario emplear el emulador DOSBox para poder utilizar este software.

[3]El repertorio de instrucciones del controlador Mark IV podría equipararse a un lenguaje ensamblador (con ciertas limitaciones), mientras que el lenguaje RoboTalk tiene construcciones de más alto nivel que pretenden facilitar la programación del robot.

Figura 3.4: Paleta de programación del controlador Rhino Mark IV


uso se conoce como programación on-line, puesto que mediante un mando de control (la paleta de programación) se realizan movimientos del brazo robótico con una conexión al robot en tiempo real (on-line) y se va almacenando las instrucciones necesarias para repetir dichos movimientos. Si bien la programación on-line mediante la paleta no permite toda la versatilidad de la programación off-line (particularmente en cuanto al repertorio de instrucciones que puede usarse), la paleta sigue siendo una herramienta de gran utilidad para la realización de programas sencillos o el aprendizaje de puntos (almacenamiento de puntos del espacio, que luego podrán ser usados en un programa, ya sea creado mediante la paleta o mediante el PC).

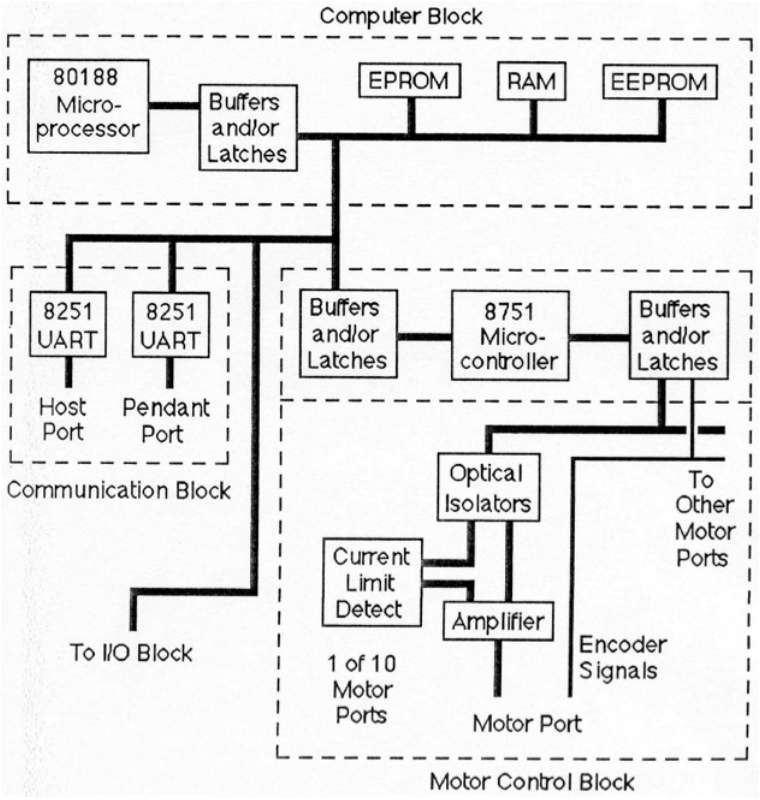
El controlador Mark IV utiliza dos procesadores: un microprocesador 80188 para el cómputo general y un microcontrolador 8751 para el control de los motores.
El microprocesador Intel 80188 se encarga del procesamiento que requieren la comunicación serie con el PC host y la paleta de programación, así como el manejo de los puertos de entrada/salida. Esto incluye la comunicación mediante el protocolo RS232-C (tanto con el PC host como con la paleta de programación), la interpretación de los comandos enviados desde el PC host, la gestión de los puertos de entrada/salida y la gestión de temporizadores definidos por el usuario. Además, el microprocesador 80188 también se encarga de la ejecución del bucle de control PID en el que se recalculan los niveles de potencia que hay que entregar a los motores (mediante PWM[4]) y de la comprobación de errores en los motores, como el bloqueo de los motores (debido, por ejemplo, al choque con un obstáculo).

[4]La técnica de modulación por ancho de pulso (del inglés *Pulse Width Modulation*) permite controlar la potencia entregada a una carga sin variar la amplitud de la señal. Así, por ejemplo, se puede utilizar para controlar la velocidad de un motor DC sin variar el voltaje de la señal de alimentación, que será normalmente el máximo voltaje admitido por el motor.

El microcontrolador 8751, por su parte, se ocupa de la lectura de los codificadores de posición de los motores y de la actualización de los registros donde se almacena el contador de pasos de los motores (es decir, la posición de los motores), así como de la generación de las señales PWM y de dirección que controlan la velocidad y sentido de giro de los motores. Aquí conviene resaltar que el microcontrolador 8751 no realiza por sí solo todo el control de los motores, puesto que no ejecuta el bucle de control PID. Como se ha indicado, el microcontrolador 8751 sólo actualiza los contadores de posición de los motores, que luego usará el microprocesador 80188 para ejecutar el bucle PID, tras lo cual volverá a comunicar al microcontrolador los nuevos niveles de PWM para el movimiento de los motores, para que el microcontrolador genere las señales de PWM de acuerdo a los últimos cálculos del bucle PID.

La organización de las partes principales del controlador Mark IV se pueden ver en la Figura 3.5. Aquí se observan los tres bloques principales: (1) el bloque de cómputo (*computer*

Figura 3.5: Organización general del controlador Mark IV




*block*), (2) el bloque de control de motores (*motor control block*), y (3) el bloque de comunicación (*communication block*).
El bloque de cómputo es la unidad que realiza el cómputo general mediante el microprocesador 80188, como ya se ha comentado. Este bloque contiene una memoria RAM para la ejecución del firmware del controlador, una memoria EPROM[5] en la que se almacena el firmware del controlador, y una memoria EEPROM[6], que es donde se almacena la configuración del contro-

[5]Erasable Programmable Read-Only Memory, memoria programable que puede ser borrada mediante luz ultravioleta para luego ser reprogramada, lo cual requiere la extracción del chip.

[6]Electrically Erasable Programmable Read-Only Memory, que puede ser borrada eléctricamente, para poder ser programada de nuevo desde el mismo firmware (sin necesidad de extraer el chip).

lador.
El bloque de control de motores es donde se encuentra el microcontrolador 8751 con el resto de la circuitería específica para el control de motores. Como ya se ha indicado, este microcontrolador ejecuta un programa minimalista que se encarga de leer los codificadores de posición de los motores y de actualizar los registros contadores de acuerdo a las lecturas, además de generar las señales PWM y de sentido de giro de los motores de acuerdo con los cálculos del bucle PID del microprocesador 80188. En la Figura 3.5 (pág. 20) también puede verse que este bloque cuenta con un módulo de opotoaisladores y un módulo de protección de sobreintensidad para proteger la electrónica del controlador ante posibles fallos eléctricos. Puesto que tanto el microprocesador 80188 como el microcontrolador 8751 necesitan acceder a ciertos registros en común, se puede ver que existen vías de comunicación entre los registros de ambos bloques.
En último lugar se ve en la figura el bloque de comunicación, que contiene dos chips UART[7] 8251, uno para la comunicación RS232-C entre el controlador y el PC host, y otro para la comunicación RS232-C entre el controlador y la paleta de programación.

El controlador Mark IV es capaz de operar en dos modos: (1) modo host (*host mode*), y (2) modo paleta de programación (*teach pendant mode*).
En el modo paleta de programación, la paleta tiene la capacidad de controlar todos los movimientos del robot y de ejecutar otras muchas acciones mediante el controlador. Puesto que la paleta de programación tiene control total sobre el robot, el PC host ve limitado el conjunto de comandos que puede enviar al controlador. De esta manera, el PC host puede seguir ejecutando un subconjunto del repertorio de instrucciones del controlador, pero no puede emplear ciertas instrucciones, puesto que interferirían con el control llevado a cabo por la paleta de programación. Por su parte, en el modo host, el PC host tiene a su disposición el repertorio de instrucciones del controlador al completo y posee control total sobre el robot. En este caso, la paleta de programación puede seguir usándose para llevar a cabo ciertas acciones, pero no puede controlar plenamente el robot, y las órdenes del PC host son prioritarias en este momento.

### 3.2.1 Tipos de robot soportados

El controlador Mark IV es capaz de controlar los motores de los manipuladores robóticos XR-4 y SCARA de Rhino Robotics, así como cualquier otro motor o brazo robótico genérico. Con el fin de dar soporte a esto, el controlador dispone de unos registros de configuración donde se puede ajustar el **tipo de robot**, a elegir entre

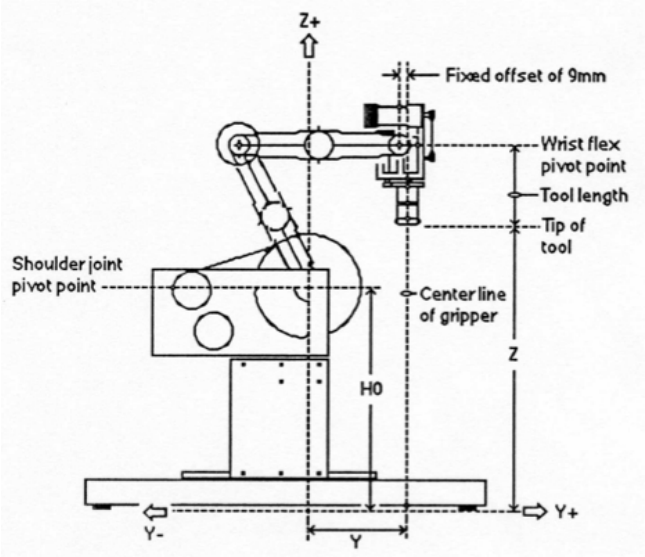
- XR3 (el antecesor del robot Rhino XR-4, totalmente compatibles entre sí),
- SCARA (el robot Rhino SCARA),
- GENERIC (robot genérico).

[7] Un receptor/transmisor universal asíncrono (del inglés *Universal Asynchronous Receiver/Transmitter*) es un circuito integrado usado para enviar y recibir bytes mediante un protocolo de comunicación serie.

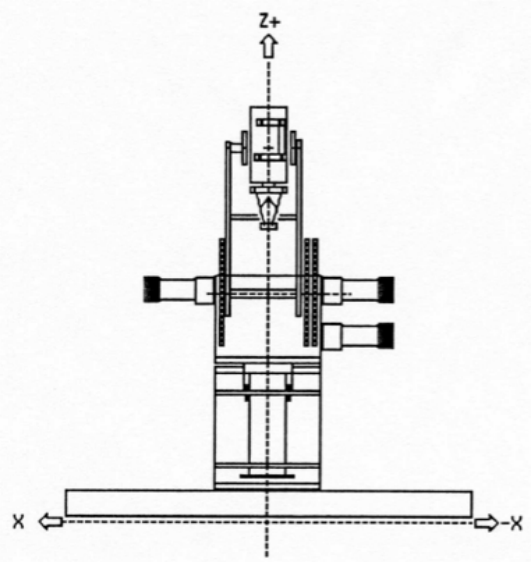
### 3.2.2 Sistemas de coordenadas del robot

El controlador Mark IV permite que la especificación de la configuración del robot (es decir, la posición donde debe encontrarse cada uno de los motores) se realice en el **espacio de las articulaciones** indicando la posición de los motores expresada en pasos de motor o, para los robots XR-4 y SCARA, también en el **espacio cartesiano**, tomando como referencia un sistema de coordenadas situado en la base del robot.
La Figura 3.6 muestra el sistema de coordenadas cartesianas del robot Rhino XR-4 tomado como referencia para los movimientos especificados en el espacio cartesiano, y la Figura 3.7 (pág. 23) muestra el sistema de coordenadas usado para el robot Rhino SCARA.

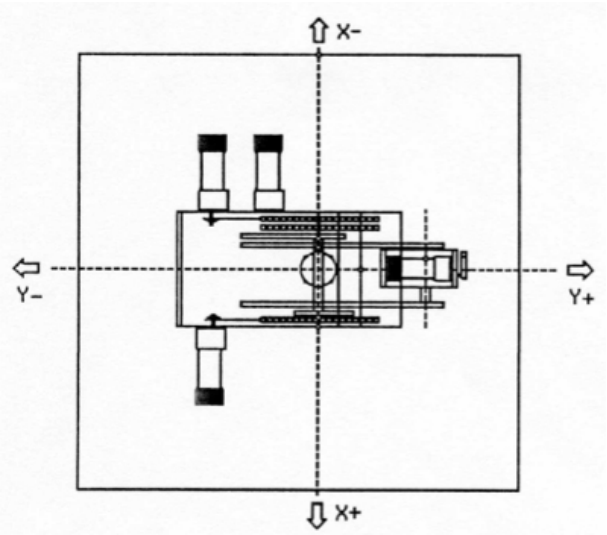
Figura 3.6: Sistema de coordenadas cartesiano del Rhino XR-4




(a) Vista lateral

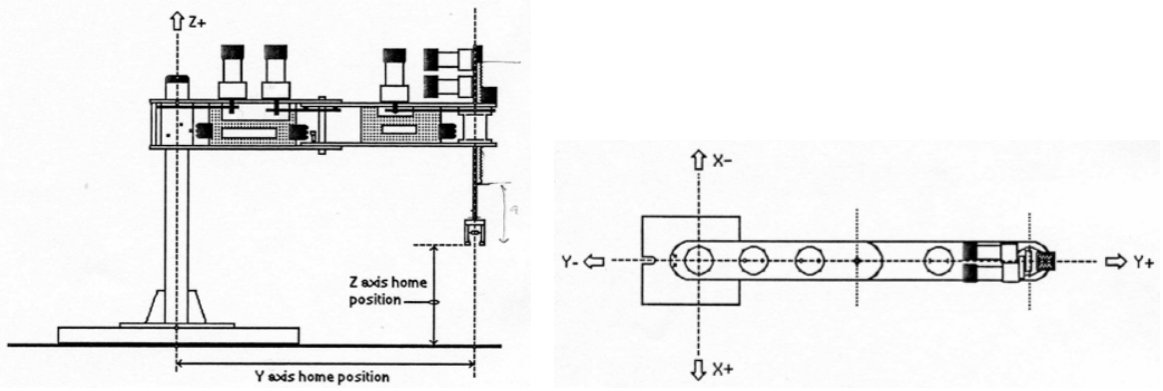



(b) Vista frontal




(c) Vista superior

Figura 3.7: Sistema de coordenadas cartesiano del Rhino SCARA




(a) Vista lateral (b) Vista superior

La especificación de la configuración en el espacio de las articulaciones es bastante simple: básicamente consiste en indicar el desplazamiento o posición de cada articulación en términos de pasos del motor. En el caso del robot Rhino XR-4 (cuyas articulaciones son todas de revolución) y el caso de las articulaciones de revolución del Rhino SCARA, esto equivale a especificar un ángulo de rotación para el motor. En el caso de la articulación prismática del robot SCARA, equivale a especificar el ángulo de rotación del motor que produce un cierto desplazamiento lineal de la articulación prismática.
Por su parte, la especificación de la configuración en el espacio cartesiano es más compleja, y representa un enfoque fundamentalmente diferente al anterior. La primera diferencia de este método con el anterior es que en el espacio de las articulaciones hay que especificar tantas variables como articulaciones tenga el robot (4 para el SCARA ó 5 para el XR-4), mientras que en el espacio cartesiano sólo se especifican las tres coordenadas (X, Y, Z) junto con un ángulo (que indica la orientación del efector final). La segunda diferencia es que las coordenadas proporcionadas no indican de forma directa la posición de los motores (variables de articulación) que producen una determinada configuración del brazo, sino que especifican la posición y orientación del efector final, a partir de lo cual debe calcularse qué valores de las variables de articulación producen dicha configuración del brazo[8].

### 3.2.3 Modos de control de motores

El controlador Mark IV permite que cada uno de los motores se encuentre en uno de cuatro modos posibles, que determinan el tipo de control que se realizará sobre el motor:

1. Motor inactivo (*idle*): no se realiza ningún tipo de control, el motor no se mueve (éste es el modo en el que deben estar los puertos de motores que no tengan un motor enchufado).

2. Motor en modo trapezoidal (*trapezoidal*): el motor se mueve siguiendo un perfil de velocidad trapezoidal con realimentación de posición, para la corrección de errores en bucle

[8]Esta información se consigue resolviendo la cinemática inversa del manipulador. Para más información, véase [1] o [2].

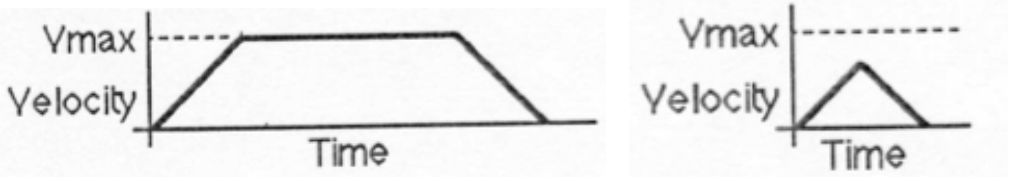
cerrado.

3. Motor en modo velocidad (*velocity*): el motor es controlado especificando la velocidad deseada, en lugar de la posición.

4. Motor en bucle abierto (*open-loop*): el motor puede moverse especificando los parámentro de PWM y sentido de giro, pero no se realiza realimentación de posición y, por tanto, el controlador no lleva a cabo corrección de posición.

De todos estos modos de funcionamiento de los motores, el más interesante de ellos (y el más utilizado en la docencia) es el modo trapezoidal. En el modo trapezoidal, todos los movimientos comandados desde el terminal o desde un programa siguen un perfil de velocidad trapezoidal. Esto quiere decir que la trayectoria a ser descrita por el motor en el espacio de la articulación se dividirá en tres fases (1) una fase de despegue, en la que el motor incrementa su velocidad desde cero hasta la velocidad máxima deseada, (2) una fase de movimiento a velocidad constante, y (3) una fase de asentamiento, en la que el motor va reduciendo su velocidad hasta llegar al punto de destino, momento en el cual la velocidad del motor llegará a cero. Esto se ilustra en la Figura 3.8(a).

Figura 3.8: Posibles perfiles de velocidad de un motor en modo trapezoidal




(a) Perfil de velocidad trapezoidal (b) Perfil de velocidad triangular

Si bien el controlador implementa un perfil de velocidad trapezoidal tal como se ha descrito, hay ocasiones en las que no es posible obtener dicho perfil de velocidad. Concretamente, en desplazamientos muy cortos, suele ocurrir que la fase de asentamiento comienza antes de que se haya concluido la fase de despegue del movimiento, puesto que el punto medio de la trayectoria es alcanzado antes de concluir la fase de despegue. En tal caso, el perfil de velocidad trapezoidal se degenera, conduciendo a un perfil de velocidad triangular, tal como se ve en la Figura 3.8(b).

### 3.2.4 Tipos de movimiento

Como ya se indicó en la Sección 3.2.2 (pág. 22), el controlador Mark IV permite especificar la configuración del brazo robótico en el espacio de las articulaciones o en el espacio cartesiano. Además, da dos opciones para la especificación de un movimiento: el movimiento se puede especificar de forma (1) relativa, o (2) absoluta.
Para especificar movimientos relativos en el espacio de las articulaciones se utiliza la orden `PR` del controlador. Si el motor se encuentra en la posición 2000, la especificación de un movimiento relativo de 1000 pasos mediante el comando `PR` supondrá la adición de 1000 pasos a la posición actual, es decir, 2000, resultando en una posición final de 3000 pasos.

En caso de especificar un movimiento absoluto, usando para ello la orden `PD`, si el motor se encuentra en la posición 2000 y se envía el comando `PD` indicando 1000 como parámetro, el controlador tomará 1000 como una posición absoluta, de manera que la posición final del motor será la posición 1000 y no la posición 3000, como sucedía en el caso anterior.

Finalmente, a la hora de ejecutar el movimiento, el controlador Mark IV da otras dos opciones para los motores en modo trapezoidal: (1) el movimiento independiente, y (2) el movimiento coordinado.
El movimiento independiente consiste en que cada articulación del robot (es decir, cada motor) ejecuta el movimiento siguiendo un perfil de velocidad trapezoidal donde la velocidad máxima del perfil es la velocidad máxima deseada para dicho motor. En general, este comportamiento provocará que algunos motores terminen su movimiento antes que otros, en caso de que el desplazamiento que deban realizar dichos motores sea más corto que el de los otros.
Por contra, en el movimiento coordinado, todas las articulaciones se mueven de forma coordinada, es decir, cada motor se mueve a una velocidad tal que todos los motores terminarán el movimiento al mismo tiempo. Para ello, el controlador reducirá la velocidad de los motores cuyo desplazamiento es más corto, mientras que mantiene la velocidad máxima deseada de los motores cuyo desplazamiento es más largo.

### 3.2.5 Registros del controlador

El controlador debe mantener en memoria cierta información sobre el estado del sistema y de los motores, además de la configuración del controlador. Mientras que la configuración permanente del sistema se almacena en un chip de memoria EEPROM, la configuración temporal y el estado del sistema se almacena en elementos de memoria abstractos denominados registros.

El controlador proporciona un conjunto de registros para ajustar los parámetros de los movimientos y para obtener información del estado del sistema y de los motores. De estos registros se puede obtener información (como, por ejemplo, la posición de un motor), o se puede escribir en ellos para modificar o comandar movimientos. Cada motor lleva asociados los siguientes registros:

- **Posición actual:** almacena la posición actual del motor medida en pasos respecto a la posición de hard home.
- **Posición de destino:** almacena la posición de destino del motor (a dónde se moverá tras la ejecución de un comando de movimiento) en el espacio de las articulaciones.
- **Posición de destino XYZ:** almacena la posición de destino del motor en el sistema de coordenadas del robot.
- **Velocidad deseada del motor:** almacena la velocidad máxima que puede alcanzar el motor durante la ejecución del perfil de velocidad trapezoidal, expresada como porcentaje de la velocidad del sistema (ver más abajo).

- **Nivel de PWM y dirección:** almacena el nivel de PWM (duración del ciclo de trabajo) y el sentido de giro del motor.

Además, existen dos registros globales[9] de gran relevancia, cuya modificación afecta al movimiento de todos los motores:

- **Velocidad del sistema:** indica la velocidad máxima que se permitirá que cualquier motor alcance, expresada en forma de porcentaje, y que se calcula sobre la velocidad máxima a la que el motor es capaz de moverse.
- **Aceleración del sistema:** indica la aceleración que deberá presentar un motor cuando se ejecuta la fase de despegue o asentamiento de un movimiento trapezoidal, expresada en porcentaje.

A estos registros, así como a los registros de configuración del sistema, no es posible acceder directamente, sino que se debe usar una instrucción del repertorio de instrucciones del controlador para leer del registro, y otra instrucción para escribir en el registro. Así, por ejemplo, para escribir en el *registro de posición deseada* de un motor, se utilizaría el comando `PD` (mencionado en la Sección 3.2.4, pág. 24), y, para leer de dicho registro, se utilizaría otro comando, en concreto, el comando `PW`.

### 3.2.6 Comandos de uso frecuente

El repertorio de instrucciones del controlador Rhino Mark IV comprende más de 80 instrucciones o comandos. Todos ellos se encuentran listados y explicados en [3], pero no todos ellos se usan con frecuencia. Por ello, en la Tabla 3.2 se listan sólo aquellos comandos que más se usan o que son fundamentales en el desarrollo de la docencia mediante el sistema robótico de Rhino Robotics.

Tabla 3.2: Comandos fundamentales para el uso del sistema

| Comando | Significado | Comentarios |
|---|---|---|
| SA | Lee el estado de los motores | Innecesario en una sesión interactiva, pero necesario para un programa de control del robot desde un PC |
| SC | Lee la configuración del sistema | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |
| SE | Lee la pila de error del sistema | |
| SM,m | Lee el modo del motor *m* | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |

[9] Es decir, hay un solo registro para todo el sistema, no un registro por cada motor.

| Comando | Significado | Comentarios |
|---|---|---|
| SS | Lee el estado del sistema | |
| CC,d | Establece el sistema de coordenadas (articulación o cartesiano) | Poco frecuente, pero necesario para la configuración inicial |
| CG,s | Habilita o deshabilita la pinza | Poco frecuente, pero necesario para la configuración inicial |
| CM,m,d | Establece el modo del motor *m* | Poco frecuente, pero necesario para la configuración inicial |
| CR,d | Establece el tipo de robot | Poco frecuente, pero necesario para la configuración inicial |
| AR | Lee la aceleración del sistema | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |
| DR,m | Lee el nivel de PWM y la dirección del motor *m* | Necesario sólo si se planea utilizar los motores en modo de bucle abierto |
| GS | Lee el estado de la pinza | Innecesario en una sesión interactiva, pero necesario para un programa de control del robot desde un PC |
| PA,m | Lee la posición actual del motor *m* | |
| PW,m | Lee la posición de destino del motor *m* | Prescindible, pero de gran utilidad para la inspección del sistema |
| VR,m | Lee la velocidad deseada del motor *m* | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |
| VX | Lee la velocidad del sistema | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |
| AC,m | Reinicia la posición actual del motor *m* | Poco frecuente, pero necesario para implementar el hard home o soft home por software (fuera del controlador) |
| AS,d | Establece la aceleración del sistema | |
| DS,m,d | Establece el nivel de PWM y la dirección del motor *m* | Necesario sólo si se planea utilizar los motores en modo de bucle abierto |
| GC | Cierra la pinza | |
| GO | Abre la pinza | |
| HH | Ejecuta un hard home para todos los motores | Aunque se podía implementar por software, es muy útil tener la rutina de hard home implementada en el controlador |
| MA | Para todos los motores y puertos auxiliares | Poco frecuente |

| Comando | Significado | Comentarios |
|---|---|---|
| MC | Inicia el movimiento coordinado de todos los motores | |
| MI | Inicia el movimiento independiente de todos los motores | |
| MM,m | Para el motor *m* | Poco frecuente |
| MS,m | Inicia el movimiento del motor *m* | Poco frecuente |
| PD,m,d | Establece la posición de destino absoluta del motor *m* | |
| PR,m,d | Establece la posición de destino relativa del motor *m* | Menos frecuente que el comando anterior, pero de gran utilidad |
| VG,d | Establece la velocidad del sistema | Poco frecuente, pero necesario para la configuración inicial |
| VS,m,d | Establece la velocidad del motor *m* | Poco frecuente, pero necesario para la configuración inicial |
| TH | Entrega el control al PC host (entra en modo host) | |
| TX | Entrega el control a la paleta de programación (entra en modo paleta de programación) | |
| KA,m,d | Establece la ganancia proporcional del motor *m* | Poco frecuente, pero necesario para la configuración inicial |
| KB,m,d | Establece la ganancia diferencial del motor *m* | Poco frecuente, pero necesario para la configuración inicial |
| KC,m,d | Establece la ganancia integral del motor *m* | Poco frecuente, pero necesario para la configuración inicial |
| RA,m | Lee la ganancia proporcional del motor *m* | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |
| RB,m | Lee la ganancia diferencial del motor *m* | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |
| RC,m | Lee la ganancia integral del motor *m* | Prescindible, pero de gran utilidad para inspeccionar la configuración del sistema |

En la Tabla 3.2 (pág. 26) se ha incluido algunos comandos cuyo uso es poco frecuente, pero que son esenciales para poder utilizar el sistema robótico, particularmente aquellos comandos que permiten configurar el sistema (la configuración se realiza una sola vez y, por tanto, el comando sólo se ejecuta una vez, pero, sin ese comando, no podría usarse el sistema).
Asimismo, se ha excluido los comandos de movimiento cartesiano. Como tal, estos comandos son fundamentales para el uso del robot, pero sólo en caso de que se quiera controlar el sistema robótico en el espacio cartesiano, lo cual es poco frecuente en la práctica. Además, el control en

el espacio cartesiano se puede realizar por software desde un PC host que resuelva la cinemática inversa y traduzca los movimientos en el espacio cartesiano a movimientos en el espacio de las articulaciones.
En esta lista tampoco se ha considerado los comandos para manejar la entrada/salida, a pesar de que estos comandos también son esenciales si se desea manejar la entrada/salida desde el controlador. Sin embargo, la gestión de la entrada/salida del controlador es un aspecto avanzado del controlador que sale del alcance del presente trabajo y que, además, se puede implementar desde el mismo PC. En otras palabras, no es necesario que la gestión de E/S la realice el propio controlador, lo cual, además, eliminaría aspectos secundarios del diseño, permitiendo concentrarse en los aspectos que sí son vitales y fundamentales para el controlador.

## 3.3 Análisis de RhinoChip

El controlador RhinoChip es el componente hardware de la plataforma RhinoChip. Consistirá en una placa de circuito impreso que da soporte a un sistema de cómputo embebido basado en microcontroladores. Este hardware representa la electrónica de control del manipulador robótico.

### 3.3.1 Requisitos de RhinoChip

1. La conexión al PC host se realizará mediante el puerto RS232-C, al igual que en el sistema antiguo.
2. Debe implementar el conjunto de instrucciones del Mark IV (las versiones iniciales de desarrollo pueden implementar sólo un subconjunto, siempre y cuando este subconjunto contenga las instrucciones más usadas, pero las siguientes versiones de desarrollo deberán implementar todo el conjunto).
3. Cualquier nuevo desarrollo debe proporcionar retrocompatibilidad (tanto a nivel del conjunto de instrucciones como en el formato de envío de instrucciones y transmisión de programas).
4. Debería permitir el control del mayor número de grados de libertad posible, teniendo en cuenta que los robots Rhino disponen de 4 (SCARA) ó 5 (XR-4) grados de libertad, más el efector final.
5. Debe controlar la velocidad y el sentido de giro de los motores.
6. Debe proporcionar al menos 6 puertos de entrada y 6 puertos de salida independientes.
7. Debe emplear dos microcontroladores, uno de propósito general (GPMCU[10]) y otro de control de motores (MCMCU[11]), para llevar a cabo las tareas que en el sistema heredado realizaban el bloque de cómputo y el bloque de control de motores, respectivamente.

[10] General Purpose Microcontroller Unit
[11] Motor Control Microcontroller Unit

8. Deberá usarse la placa de desarrollo Microchip dsPICDEM 2 con microcontroladores de la familia dsPIC30F.

### 3.3.2 Comparativa de hardware

Al surgir la idea de este proyecto, se pensó inicialmente en diseñar e implementar un controlador robótico para reemplazar el Rhino Mark IV usando para ello alguna plataforma de hardware libre basada en microcontroladores, como podría ser la plataforma Arduino[12].

Sin embargo, debido a la escasa disponibilidad de placas Arduino en el departamento y a las bajas prestaciones de la versión de Arduino disponible, además de la disponibilidad de hardware de Microchip en desuso, se planteó como requisito para el proyecto el uso del hardware de Microchip ya presente. En concreto, se disponía de dos placas de desarrollo dsPICDEM 2 y varios microcontroladores de la familia dsPIC30F adquiridos para proyectos anteriores pero que no habían sido usados.

Si bien desde casi el principio se impuso la utilización de dicho hardware como requisito, no por ello hemos dejado de investigar y comparar las alternativas existentes. En este estudio comparativo se han tenido en cuenta diferentes versiones de Arduino (la mayoría basadas en microcontroladores ATmega de Atmel), la familia de microcontroladores dsPIC30F de Microchip[13], y la plataforma mbed[14], que emplea microcontroladores basados en ARM.
Los motivos para elegir la plataforma Arduino para esta comparativa son su especificación abierta (hardware y software libre) y su bajo coste. Por otro lado, la familia dsPIC30F merece ser tenida en cuenta porque es una de las opciones profesionales de facto en la industria, para así comparar sus características y prestaciones frente a las otras opciones, y analizar si conviene su adquisición y uso. Por último, se ha incluido la plataforma mbed en la comparativa porque, junto con la plataforma Arduino, es otra de las plataformas de desarrollo de bajo coste que gozan de gran popularidad entre los aficionados a la electrónica y los poyectos amateur.
Aparte de las tres plataformas ya mencionadas, existen otras muchas más plataformas de desarrollo basadas en microcontroladores, así como existen diversas familias de microcontroladores PIC, y otras muchas plataformas basadas en microcomputadores (Raspberry Pi, BeagleBone, Cubieboard, etc.). Sin embargo, no es factible realizar un análisis comparativo exhaustivo de todas ellas, por lo que nos ceñiremos a estas tres plataformas debido a su popularidad, carácter abierto y bajo coste (en el caso de Arduino o mbed), y debido a su amplia cuota de mercado en las aplicaciones industriales y profesionales (en el caso de los PIC de Microchip).

La plataforma Arduino ofrece una gran variedad de placas de diferentes tamaños, formas y especificaciones técnicas. Para el desarrollo de este proyecto se han descartado las placas más pequeñas con recursos más modestos, así como aquéllas que disponen de conectividad inalámbrica o de red cableada, puesto que no se necesitan semejantes características. Las versiones de Arduino más interesantes para el presente trabajo son, por tanto:

[12] http://www.arduino.cc/
[13] http://www.microchip.com/
[14] https://mbed.org/

- Arduino Uno
- Arduino Leonardo
- Arduino Due
- Arduino Robot
- Arduino ADK
- Arduino Mega

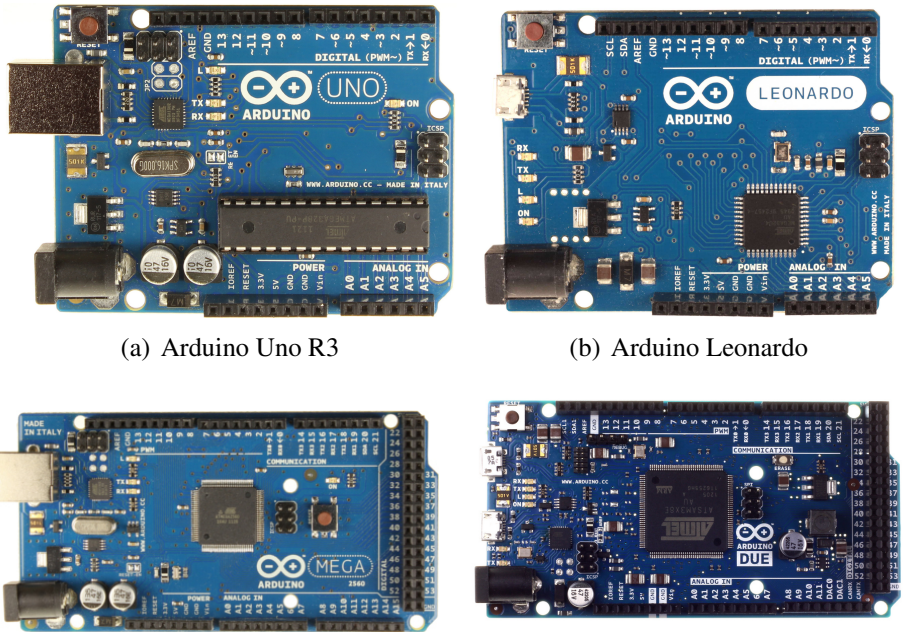
Las versiones Arduino Robot y Arduino ADK resultan interesantes, pero no se analizarán en la comparativa. En el caso del Arduino Robot, esto es así porque la plataforma está más enfocada a la robótica móvil, y la placa del Arduino Robot está dotada con diversos componentes que no se necesitan en este proyecto. Por su parte, la placa Arduino ADK resulta interesante porque está dedicada a la integración de Arduino con Android, lo cual podría ser de utilidad para desarrollar una aplicación en la que un móvil Android pudiera ser utilizado como paleta de programación. Sin embargo el Arduino ADK es muy similar al Arduino Mega (se basa en éste) y los mismos resultados (o similares) se podrían obtener con un Arduino Mega u otra versión de Arduino, por lo que no merece la pena considerar el Arduino ADK en la comparativa como una placa independiente.
Las características más interesantes de la plataforma Arduino son su popularidad —lo cual conlleva una gran comunidad y soporte—, su bajo coste, su carácter abierto —permitiendo personalizar los diseños y adaptarlos a necesidades específicas— y su facilidad de uso. Respecto a esto último, la plataforma Arduino proporciona un entorno de desarrollo basado en Processing que utiliza el lenguaje Wiring —un lenguaje de sintaxis similar a C— y que trae consigo bibliotecas de alto nivel para realizar funciones como la E/S y la generación de señales de PWM. La comparativa entre las diferentes versiones de Arduino se puede ver en la Tabla 3.3 (pág. 33).

Por su parte, la oferta de microcontroladores de Microchip Technology, Inc. es muy variada, siendo la familia PIC la más popular. Sin embargo, puesto que ya se dispone en el departamento de unos modelos muy concretos de los microcontroladores de la familia dsPIC, serán estos los que se analizarán. En concreto, se dispone ya de la placa de desarrollo dsPICDEM 2, junto con los siguientes microcontroladores:

- dsPIC30F3012 (*General Purpose and Sensor Family*)
- dsPIC30F4013 (*General Purpose and Sensor Family*)
- dsPIC30F4011 (*Motor Control and Power Conversion Family*)

Se dispone, por tanto, de microcontroladores de propósito general (*General Purpose and Sensor Family*) y microcontroladores para control de motores (*Motor Control and Power Conversion Family*). Entre ambas familias de microcontroladores dsPIC no existen grandes diferencias, excepto que los microcontroladores de la familia de control de motores están diseñados

Figura 3.9: Diferentes modelos de la *Arduino I/O Board* analizados en la comparativa




(a) Arduino Uno R3

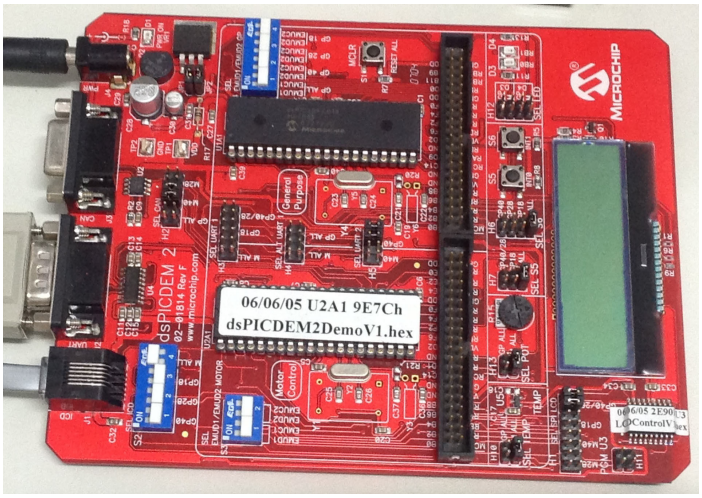
(b) Arduino Leonardo

(c) Arduino Mega 2650

(d) Arduino Due

específicamente para el control de motores y, por tanto, disponen de algunos periféricos adicionales, como un módulo PWM o un módulo QEI[15], sacrificando para ello puertos de entrada/salida general.
Las características más destacadas de los microcontroladores de Microchip son sus altas prestaciones y la amplia aceptación de dichos microcontroladores en el desarrollo de aplicaciones

Figura 3.10: Placa de desarrollo dsPICDEM 2 con un dsPIC30F4013 (arriba) y un dsPIC30F4011 (abajo)




[15]Quadrature Encoder Interface, un módulo para hacer de interfaz con los condificaciones de posición incrementales.

Tabla 3.3: Características de las diferentes versiones de Arduino seleccionadas

| | Arduino Uno | Arduino Leonardo | Arduino Mega | Arduino Due |
|---|---|---|---|---|
| Microcontrolador | Atmel ATmega 328 | Atmel ATmega 32u4 | Atmel ATmega 2560 | Atmel SAM3X8E ARM Cortex-M3 |
| Voltaje de operación | 5 V | 5 V | 5 V | 3.3 V |
| Arquitectura y longitud de palabra | AVR (RISC) 16–32 bits | AVR (RISC) 16–32 bits | AVR (RISC) 16–32 bits | ARM Cortex (RISC) 32 bits |
| Registros de propósito general | 32 registros de 8 bits | 32 registros de 8 bits | 32 registros de 8 bits | 13 registros de 32 bits |
| Velocidad de la CPU | 16 MHz (16 MIPS)[16] | 16 MHz (16 MIPS) | 16 MHz (16 MIPS) | 84 MHz |
| Memoria de programa (flash) | 32 KB (bootloader ocupa 0.5 KB) | 32 KB (bootloader ocupa 4 KB) | 256 KB (bootloader ocupa 8 KB) | 512 KB |
| Memoria de datos (SRAM) | 2 KB | 2.5 KB | 8 KB | 96 KB (2 bancos: 64 KB + 32 KB) |
| Memoria EEPROM | 1 KB | 1 KB | 4 KB | No |
| Pines de E/S digital | 14 (6 PWM) | 20 (7 PWM) | 54 (15 PWM) | 54 (12 PWM) |
| Número de UARTs | 1 | 1 | 4 | 4 |
| Módulos I2C | 1 | 1 | 1 | 2 |
| Módulos SPI | Sí (multiplexado con 2 pines PWM) | Sólo con conector ICSP | Sí | Sí |
| Quadrature Encoder Interface | No[17] | No[17] | No[17] | Disponible por medio de timers contadores (hasta 3 codificadores) |
| Temporizadores | 2 timers de 8 bits, 1 timer de 16 bits | 1 timer de 8 bits, 2 timers de 16 bits, 1 timer de 10 bits de alta velocidad | 2 timers de 8 bits, 1 timer de 16 bits | 9 timers de 32 bits de propósito general |

| | Arduino Uno | Arduino Leonardo | Arduino Mega | Arduino Due |
|---|---|---|---|---|
| Precio del microcontrolador | Incluido en la placa | Incluido en la placa | Incluido en la placa | Incluido en la placa |
| Precio de la placa de desarrollo | 20 €[18] | 18 €[18] | 39 €[18] | 39 €[18] |
| Precio de las herramientas de desarrollo | Gratuito | Gratuito | Gratuito | Gratuito |
| Precio del programador | No necesita[19] | No necesita[19] | No necesita[19] | No necesita[19] |



[16]La plataforma Arduino usa un reloj de 16 MHz, pero el microcontrolador soporta hasta 20 MHz (20 MIPS). Puesto que todas las instrucciones tardan 1 ciclo en ejecutarse, a 16 MHz el rendimiento es de 16 MIPS.

[17]Si bien no dispone de módulo hardware ni biblioteca software para interactuar con codificadores incrementales, esta funcionalidad se puede implementar por software utilizando pines entrada digitales.

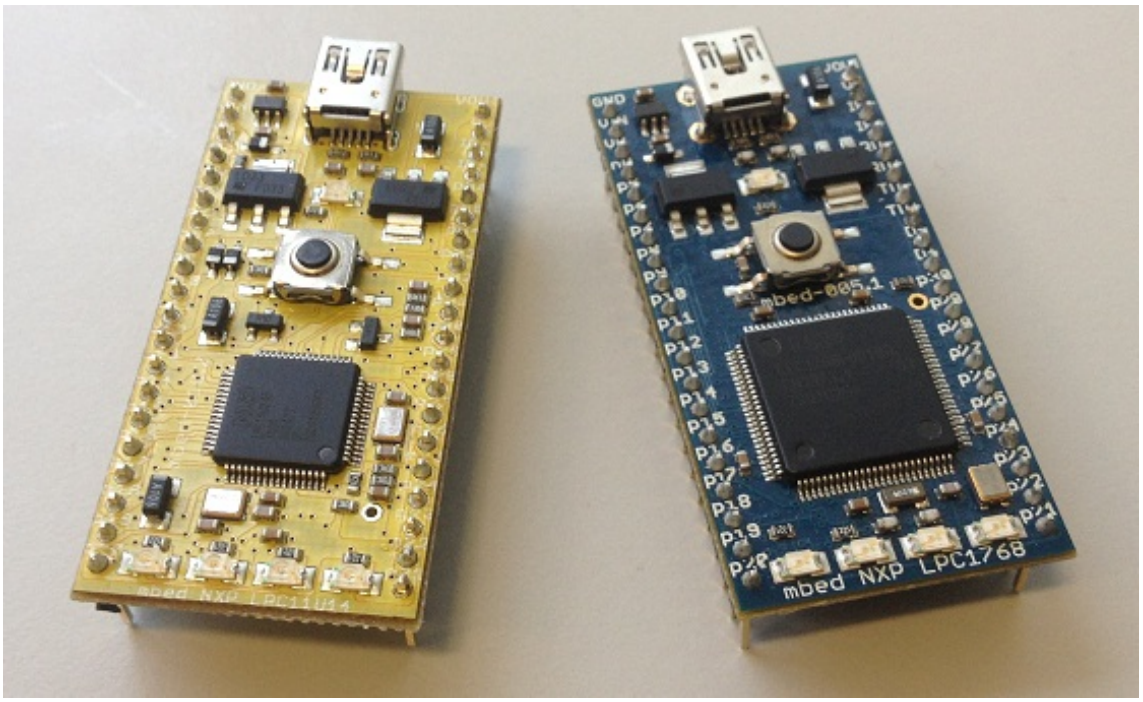
[18]Los precios han sido extraídos de la tienda online oficial de Arduino (http://store.arduino.cc/) en la fecha de redacción de este documento, y no incluyen gastos de envío ni impuestos. Son proporcionados sólo con fines estimativos.

[19]Para programar el Arduino se requiere un cable USB A/B estándar (como los que usan otros periféricos de PC, como las impresoras USB modernas), o un cable USB A/mini-B (como los que usan muchas cámaras digitales), dependiendo de la versión de Arduino elegida.

industriales. Si bien el coste de las herramientas de desarrollo es considerablemente alto, el coste de los chips es bastante razonable para la aplicación final. En cuanto a la programabilidad, estos microcontroladores pueden ser programados en ensamblador o C, y existen bibliotecas para la utilización de los diferentes periféricos de los microcontroladores, como el PWM o la EEPROM, además de funciones de comunicación (UART, I2C, SPI, CAN) y de E/S.
La información comparativa de los microcontroladores dsPIC se puede ver en la Tabla 3.4 (pág. 36).

Por último, la plataforma mbed es una plataforma de desarrollo con microcontroladores basados en la arquitectura ARM Cortex de 32 bits. mbed es otra plataforma que goza de bastante popularidad entre la comunidad de aficionados a la electrónica y que proporciona un gran potencial para la creación de proyectos. Al igual que Arduino, la plataforma mbed también es bastante asequible y las herramientas de desarrollo están disponibles de forma gratuita. Lo más llamativo de esta plataforma es quizá su entorno de desarrollo basado en web, que permite crear programas en C/C++ desde cualquier navegador web moderno en cualquier sistema operativo, y su amplia biblioteca de "recetas" de código [4]. Además, con esta plataforma se han creado proyectos como ERIC, el perro robótico interactivo[20].
De la plataforma mbed destaca su gran capacidad de cómputo, al estar basados en microcontroladores con arquitectura ARM Cortex de 32 bits, y su amplia biblioteca de funciones mantenida por la comunidad de mbed, y que es seguramente la biblioteca de funciones más extensa de entre las tres plataformas analizadas, incluyendo, entre otras, bibliotecas de PWM, QEI y PID [4], que son de gran utilidad en cualquier proyecto de robótica.
La comparativa de la plataforma mbed se puede ver en la Tabla 3.4 (pág. 36).

Figura 3.11: Placas de la plataforma mbed con el NXP LPC11U24 (izquierda) y el NXP LPC1786 (derecha) analizadas en la comparativa




[20] http://youtu.be/XiR61Ecs5JU

Tabla 3.4: Características de los dsPIC30F y la plataforma mbed

| | dsPIC30F3012 | dsPIC30F4013 | dsPIC30F4011 | mbed ARM Cortex-M0 (bajo consumo) | mbed ARM Cortex-M3 (alto desempeño) |
|---|---|---|---|---|---|
| Microcontrolador | Microchip dsPIC30F3012 | Microchip dsPIC30F4013 | Microchip dsPIC30F4011 | NXP LPC11U24 | NXP LPC1768 |
| Voltaje de operación | 2.5 V – 5.5 V | 2.5 V – 5.5 V | 2.5 V – 5.5 V | 2.4 V – 9 V | 4.5 V – 9 V |
| Arquitectura y longitud de palabra | RISC 16 bits | RISC 16 bits | RISC 16 bits | ARM Cortex (RISC) 32 bits | ARM Cortex (RISC) 32 bits |
| Registros de propósito general | 16 registros de 16 bits | 16 registros de 16 bits | 16 registros de 16 bits | No especificado[21] | No especificado[21] |
| Velocidad de la CPU | 30 MIPS (a 30 MHz) | 30 MIPS (a 30 MHz) | 30 MIPS (a 30 MHz) | 48 MHz | 96 MHz |
| Memoria de programa (flash) | 24 KB | 48 KB | 48 KB | 32 KB | 512 KB |
| Memoria de datos (SRAM) | 2 KB | 2 KB | 2 KB | 8 KB | 32 KB |
| Memoria EEPROM | 1 KB | 1 KB | 1 KB | 4 KB | No |
| Pines de E/S | 12 (2 PWM) | 30 (4 PWM) | 30 (6 PWM[22]) | 54 (8 PWM) | 70 (6 PWM) |
| Número de pines | 18 | 40 | 40 | 64 | 100 |
| Número de UARTs | 1 | 2 | 2 | 1 | 3 |
| Módulos I2C | 1 | 1 | 1 | 1 | 2 |
| Módulos SPI | 1 | 1 | 1 | 2 | 2 |
| Módulos QEI | 0[23] | 0[23] | 1[23] | 0[23] | 1[23] |

| | dsPIC30F3012 | dsPIC30F4013 | dsPIC30F4011 | mbed ARM Cortex-M0 (bajo consumo) | mbed ARM Cortex-M3 (alto desempeño) |
|---|---|---|---|---|---|
| Temporizadores | 3 timers de 16 bits, 1 timers de 32 bits | 5 timers de 16 bits, 2 timers de 32 bits | 5 timers de 16 bits, 2 timers de 32 bits | 2 timers de 32 bits, 2 timers de 16 bits | 4 timers de 32 bits |
| Precio del microcontrolador | US$2.30–US$3.06 | US$5.41–US$6.12 | US$5.56–US$7.54 | Incluido en la placa | Incluido en la placa |
| Precio de la placa de desarrollo | US$99.99[24] | US$99.99[24] | US$99.99[24] | US$45[25] | US$49[25] |
| Precio de las herramientas de desarrollo | US$180–US$1845[26] | US$180–US$1845[26] | US$180–US$1845[26] | Gratuito | Gratuito |
| Precio del programador | US$24.99–US$229.99[27] | US$24.99–US$229.99[27] | US$24.99–US$229.99[27] | No necesita | No necesita |

[21]Las hojas de especificaciones de los microcontroladores no indican este dato.

[22]Dispone de un módulo de PWM por hardware con 6 pines de salida, pero estos pines se encuentran emparejados de forma que sólo pueden controlarse hasta 3 motores. Para más información, véase la Sección 4.1.5 (pág. 76).

[23]Aunque no disponga de suficientes módulos QEI por hardware, esta funcionalidad se puede implementar por software usando pines de E/S general.

[24]La placa dsPICDEM 2 tiene zócalos para ubicar dos microcontroladores simultáneamente, uno de propósito general y otro de control de motores, y los microcontroladores son extraíbles, por lo que sólo es necesario adquirir una unidad de la placa, aunque se puede prescindir de una placa si se usan ciertos programadores, como el PICkit. Además, para el sistema final se diseñaría una placa específica más barata que la placa de desarrollo.

[25]El precio de la plataforma mbed ha sido consultado en `http://mbed.org/handbook/Order` y representa el precio base del fabricante, sin impuestos ni gastos de envío. Sí incluye el acceso al entorno de desarrollo online. Este precio se proporciona sólo con fines estimativos.

[26]El entorno de desarrollo MPLAB IDE es gratuito. También hay compiladores gratuitos, pero están limitados en funcionalidad (en cuanto a las optimizaciones de código). Existen diversos compiladores de pago, tanto de Microchip (US$180–US$1845) como de terceros (US$250–US$600).

[27]Existen diversos programadores. Los MPLAB ICD2 e ICD3 están entre los US$189.99 y US$229.99. Los programadores de bajo coste como el PICkit y similares oscilan entre US$24.99 y US$69.99. También existen programadores clónicos fabricados en Asia a precios menores.

Teniendo en cuenta los requisitos del controlador (Sección 3.3.1, pág. 29), los aspectos más importantes a considerar en la plataforma hardware a elegir son

1. la disponibilidad de suficientes salidas de PWM para el control del mayor número de motores posible[28];
2. la disponibilidad de un módulo QEI, para realizar las lecturas de posición de los codificadores de los motores;
3. la disponibilidad de suficientes pines de entrada/salida para el control del sentido de giro de los motores, para la lectura de los interruptores de final de carrera, para la realización de la entrada/salida del controlador, y para suplir las posibles carencias en la disponibilidad de salidas PWM y módulos QEI;
4. la capacidad de la memoria de programa, debido a que el tamaño del código necesario para las tareas de control y los cálculos de la cinemática puede llegar a crecer considerablemente;
5. la capacidad de la memoria de datos, debido a que debe almacenarse el estado del controlador y todos los datos de las rutinas y subrutinas llamadas recursivamente;
6. la disponibilidad de una memoria EEPROM para el almacenamiento persistente de la configuración del controlador y de los datos necesarios para la resolución de la cinemática del manipulador (p.ej. matrices de los brazos)[29];
7. la disponibilidad de suficientes puertos para la comunicación con un PC host y, opcionalmente, también con una paleta de programación (presumiblemente UARTs para la comunicación RS232-C);
8. el coste de la plataforma, ya que se dispone de un presupuesto muy limitado.

Además de todo esto, si se decide implementar el sistema usando un solo microcontrolador, éste deberá tener una potencia de cómputo suficiente como para realizar todas las tareas de cómputo, que incluyen la interpretación de comandos y el control de motores, y unas memorias suficientemente grandes como para alojar el programa completo y los datos, por lo que a estos requisitos se añade una alta frecuencia de reloj y gran capacidad de almacenamiento en la memoria de programa y en la memoria de datos.
Por contra, si se utilizan varios microcontroladores, de manera que se repartan las tareas de cómputo entre ellos, no será necesaria una potencia de cómputo individual elevada ni tanta capacidad de almacenamiento en las memorias de programa y datos, pero se requerirán mecanismos para la intercomunicación y sincronización de los microcontroladores. Esta intercomunicación podría realizarse mediante memoria compartida o por paso de mensajes, presumiblemente mediante los protocolos I2C o SPI.

[28] Puesto que el controlador se orientará al control del Rhino XR-4 y el Rhino SCARA, deberá ser capaz de controlar un mínimo de 6 motores.

[29] La resolución de la cinemática sólo será necesaria si se implementa el movimiento en el espacio cartesiano. Como ya se ha indicado, esto escapa del alcance del proyecto, pero hay que prever que pueda hacerse en un futuro y, por ello, elegir una plataforma y realizar un diseño adaptable y extensible que permita este tipo de ampliaciones.

En la tabla Tabla 3.5 se detallan los requisitos funcionales mínimos que debe cumplir la plataforma hardware para posibilitar el desarrollo del proyecto. Aquellos requisitos que están marcados como opcionales son los requisitos cuya inclusión en el diseño depende de las características de la plataforma elegida o de decisiones de diseño. Así, por ejemplo, si la plataforma dispone de suficientes salidas de PWM por hardware, no hará falta tener 6 pines de E/S digital adicionales para implementar PWM por software, mientras que, de no ser así, habría que buscar una plataforma con 6 pines de E/S digital que se puedan utilizar para implementar PWM por software sin mermar las otras funcionalidades del controlador.

Tabla 3.5: Requisitos funcionales mínimos de la plataforma hardware elegida

| | |
|---|---|
| Señal de PWM | 6 salidas de PWM por hardware independientes, o 6 pines de E/S digital para implementar PWM por software |
| Módulos QEI | 6 módulos QEI por hardware, o 12 pines de E/S digital para implementar QEI por software |
| Pines de E/S digital | • 6 pines de entrada para E/S del controlador<br>• 6 pines de salida para E/S del controlador<br>• 6 pines de entrada para interruptores de final de carrera<br>• 6 pines de salida para PWM por software (opcional)<br>• 6 pines de salida para control del sentido de giro<br>• 12 pines de entrada para implementar QEI por software (opcional) |
| Memoria de programa (flash) | Mínimo 64 KB (en total entre todos los microcontroladores) |
| Memoria de datos (RAM) | Mínimo 4 KB (en total entre todos los microcontroladores) |
| Memoria EEPROM | Menos de 1 KB (suficiente para almacenar la información de configuración del controlador) |

| | |
|---|---|
| Puertos de comunicación | • 1 UART para el PC host<br>• 1 UART para una paleta de programación (opcional)<br>• 1 bus para comunicar varios microcontroladores, presumiblemente I2C o SPI (opcional) |

Llegados a este punto es importante resaltar que, sea cual sea la plataforma que se elija, lo normal sería utilizar la plataforma sólo para el desarrollo. Es decir, la placa adquirida se utilizaría sólo como placa de desarrollo, y no en producción. A la hora de crear el sistema final, lo normal no es utilizar la placa de desarrollo para implementar el sistema en producción (especialmente si se desea producir en masa y a bajo coste), sino más bien diseñar una placa de circuito impreso específica para la aplicación, conteniendo sólo lo necesario, lo cual, en general, contribuye a reducir costes respecto al uso de una placa de desarrollo.

Siguiendo los criterios de la Tabla 3.5 (pág. 39) para la elección de una plataforma para desarrollar e implementar la aplicación, de todos los modelos de Arduino presentados en la Tabla 3.3 (pág. 33), los modelos más modestos no parecen adecuados según las necesidades de cómputo y E/S del proyecto, a menos que se utilice varios microcontroladores en paralelo. Sin embargo, esto no es práctico ni eficiente; ante todo, porque el coste que ello implicaría es del mismo orden (o mayor) que el coste de adquirir un sistema más potente, al menos en comparación a la adquisición de uno de los modelos superiores de Arduino.
De los modelos superiores, tanto el Arduino Mega como el Arduino Due aprueban con creces en los criterios de E/S. En cambio, el Arduino Mega está bastante limitado en la frecuencia de reloj (no así en el tamaño de la memoria). Si bien es muy complicado tener una certeza a priori de si los recursos de cómputo del Arduino Mega serían suficientes para esta aplicación o no, el riesgo de optar por el Arduino Mega es muy grande, mientras que, por el mismo precio, el Arduino Due presenta unas capacidades de E/S muy similares, pero con mayor frecuencia de reloj y capacidad de almacenamiento.
Por tanto, de optar por la plataforma Arduino, se elegiría implementar el sistema en un único microcontrolador utilizando la placa Arduino Due como placa de desarrollo. Las ventajas de esta opción son su bajo coste (la plataforma en sí es barata y no requiere programadores caros), su alto rendimiento y capacidad de almacenamiento, y su facilidad de programación y versatilidad (el lenguaje de programación y el entorno de la plataforma Arduino son simples, gratuitos, y disponen de muchas bibliotecas de alto nivel, a la par que se permite la posibilidad de programar el microcontrolador directamente en C con cualquier compilador de Atmel[30]).

En cuanto a los microcontroladores de la familia dsPIC30F, esta familia está orientada a

[30]La desventaja de esta elección es la elevada complejidad de la arquitectura del núcleo ARM Cortex, lo cual se mitigaría programando el microcontrolador con las herramientas de desarrollo de la plataforma Arduino.

todo tipo de tareas de control, aunque está especializada en tareas de procesamiento de señal[31]. De los microcontroladores disponibles en esta familia, los más adecuados por capacidad de cómputo y recursos proporcionados son los de gama más alta. Esto descarta los microcontroladores de 28 pines o menos en favor de los de 40 pines, con lo que, de los modelos analizados, descartaríamos el 3012 y nos quedaríamos con el 4013 y el 4011.
Si bien estos microcontroladores ofrecen un gran equilibrio entre capacidad de cómputo y precio, con frecuencias de reloj y capacidad de almacenamiento más que decentes a un precio ajustado, uno solo no es suficiente para la aplicación que nos ocupa. En una aplicación de control de motores sin grandes exigencias podría bastar con usar un microcontrolador de control de motores, como el 4011. Sin embargo, debido a la carga de cómputo de nuestra aplicación, es preferible usar dos microcontroladores. En primer lugar, porque las tareas a realizar por el sistema podrían superar las capacidades de un único microcontrolador y, en el mejor de los casos, habría que buscar soluciones para realizar multitarea, pudiendo llegar a ser necesario emplear algún sistema operativo para sistemas embebidos. Y, en segundo lugar, porque la disponibilidad de pines de E/S de un único microcontrolador no satisface las necesidades de E/S de la aplicación.
Además, los microcontroladores dsPIC30F proporcionan una amplia conectividad, con dos UARTs, un módulo I2C y un módulo SPI, con lo que se podrá satisfacer las necesidades de comunicación de la aplicación en cuestión.

Finalmente, la versión de la plataforma mbed con el microcontrolador LPC1768 se presenta como una alternativa muy atractiva, que satisfaría todos los requisitos de la aplicación, puesto que cuenta con unos recursos de cómputo más que suficientes (alta frecuencia de reloj y gran capacidad de almacenamiento), una gran cantidad de pines de E/S y una conectividad sobresaliente.
Sin embargo, puesto que la plataforma sólo dispone de un microcontrolador —aunque de muy altas prestaciones—, sería necesario implementar mecanismos de multitarea para poder garantizar la ejecución de las tareas de interpretación de comandos y de control de motores.

En definitiva, aunque el requisito de usar los microcontroladores dsPIC venía impuesto de antemano, realmente llegamos a la conclusión de que es una muy buena elección. De hecho, es quizá la mejor opción de todas las analizadas, puesto que satisface las necesidades fundamentales de la aplicación a un precio bastante reducido. A pesar de que quizá en E/S el sistema esté muy ajustado —es decir, en el número de pines de E/S disponibles para implementar la E/S del controlador—, esta opción aporta como mayor ventaja la posibilidad de separar las tareas de cómputo general en un microcontrolador y las tareas de control de motores en otro microcontrolador, por lo que se garantiza que tareas como la interpretación de comandos no robarán tiempo de cómputo a las tareas de control de motores, que son críticas en esta aplicación y, por tanto, deben ser prioritarias. Aparte de esto, esta estructura es una apuesta segura, puesto que es similar a la estructura del controlador original.
Si bien las herramientas de desarrollo de la plataforma dsPIC son caras, sólo es necesario adquirirlas una vez, para el desarrollo. Para la implementación y la producción en masa se reducirían mucho los costes, puesto que se fabricaría una placa de circuito impreso sólo con lo necesario

[31] La arquitectura de los dsPIC proporciona instrucciones especializadas en DSP (*Digital Signal Processing*) y operaciones vectoriales y matriciales, lo cual puede ser muy útil para acelerar los cálculos de la cinemática de los manipuladores, que suelen requerir operaciones en 3 y 4 dimensiones.

para la aplicación, y no haría falta adquirir las placas de desarrollo ni los programadores/depuradores para cada unidad del sistema definitivo.
Aunque el Arduino Due y el mbed NXP LPC1768 son alternativas muy atractivas que seguramente tendrían un desempeño sobresaliente, su gran capacidad de almacenamiento (principalmente en la memoria de programa) es quizá excesiva para esta aplicación, así como su frecuencia de reloj, por lo que parte del hardware estaría siendo desaprovechado. Además, tanto el Arduino Due como el mbed NXP LPC1768 adolecen de la ausencia de EEPROM para el almacenamiento persistente de la configuración del sistema, y el mbed NXP LPC1768 dispone de periféricos que no son necesarios en esta aplicación (particularmente una interfaz de red Ethernet).

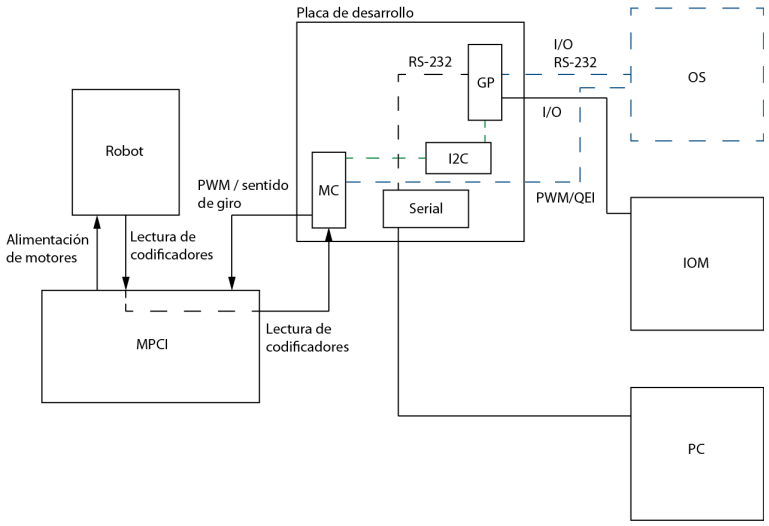
### 3.3.3 Organización esquemática del hardware

El susbsistema RhinoChip se implementará sobre una placa de desarrollo (Microchip dsPICDEM 2) que cuenta con dos zócalos para dos microcontroladores, uno de propósito general y otro para control de motores. También proporciona una conexión RS232 para la comunicación serie entre una UART (de alguno de los dos microcontroladores, seleccionable mediante jumpers en la placa) y un PC u otro dispositivo externo.

Puesto que se requiere la comunicación entre los dos microcontroladores y no se dispone de memoria compartida, la solución más simple y económica es implementar un protocolo de comunicación entre los dos microcontroladores, idealmente mediante SPI o I2C, dando prioridad a este último por su sencillez y porque es un estándar bien conocido.

Con estas consideraciones y los requisitos especificados, la organización del hardware RhinoChip queda como se indica en la Figura 3.12. Aquí puede verse la placa de desarrollo con

Figura 3.12: Organización esquemática del hardware del controlador RhinoChip




los dos microcontroladores, el de propósito general (GP[32]) y el de control de motores (MC[33]).

[32]Del inglés *General Purpose*.
[33]Del inglés *Motor Control*.

El microcontrolador GP se comunica por RS232 con el PC host mediante el módulo de comunicación serial de la placa de desarrollo, y también se comunica mediante I2C con el microcontrolador MC. Además, sobre los microcontroladores se ejecuta el sistema operativo (OS), que realizará las tareas de gestión de la entrada/salida (I/O), comunicación serie y control de motores (PWM/QEI).
Los puertos de entrada/salida estarán localizados en el bloque IOM (*Input/Output Module*), que estará conectado directamente al microcontrolador GP y que proporcionará la interfaz de puertos de entrada/salida y una circuitería de seguridad para aislar la electrónica de fallos externos (p.ej. intensidades o voltajes muy elevados).
Por último, el microcontrolador MC está conectado directamente al módulo MPCI (*Motor Power & Control Interface*), mediante el cual el microcontrolador podrá controlar los motores mediante una cierta señal de PWM y una señal de sentido de giro, así como leer los codificadores de posición de los motores.

Por tanto, la fase de diseño del hardware RhinoChip se ocupará de definir y diseñar los módulos IOM y MPCI, así como las interconexiones en la placa de desarrollo. Por su parte, el módulo OS mostrado en la figura es objeto de la Sección 3.4.

## 3.4 Análisis de RhinoChip OS

El firmware RhinoChip OS es el componente software del controlador RhinoChip, que se ejecutará sobre la plataforma hardware homónima, analizada en la Sección 3.3 (pág. 29).

### 3.4.1 Requisitos de RhinoChip OS

1. Debe gestionar el puerto serie:
   (a) Configurar el puerto serie en el arranque del sistema, para la comunicación con el PC host.
   (b) Manejar y tratar los datos recibidos.
2. Debe ejecutar las acciones asociadas a las órdenes enviadas desde el PC host a través del puerto serie, y enviar la respuesta a cada orden que lo requiera.
3. Debe mantener el estado del sistema en memoria (volátil)
   (a) Variables de articulación y posición de los motores
   (b) Variable del sistema (velocidad, aceleración, ganancias del bucle PID, configuración del controlador, etc.)
4. Debe realizar las tareas de control de motores necesarias para mantener el robot en una configuración determinada o para realizar movimientos con el robot:
   (a) Ejecutar un bucle de control PID para mantener cada articulación en una posición dada y compensar cualquier error de posición.

(b) Ejecutar movimientos de las articulaciones para llevar el robot a una configuración dada, mediante un perfil de velocidad trapezoidal (si las articulaciones implicadas en el movimiento están en modo trapezoidal).

5. Debe gestionar los puertos de E/S digital:

   (a) Configurar los puertos de E/S en el arranque del sistema, para permitir su uso desde el programa de usuario.

   (b) Mantener una estructura de datos con la lista de puertos de entrada por los que se está esperando para la reanudación del programa de usuario (sólo cuando se desee implementar los comandos que requieren el bloqueo del programa).

6. Debe manejar la memoria no volátil (EEPROM) para el almacenamiento persistente de algunas variables del sistema (ganancias, configuración del controlador, etc.)

### 3.4.2 Arquitectura de RhinoChip OS

Puesto que la plataforma RhinoChip dispone de dos unidades de proceso (microcontroladores), las tareas del firmware se repartirán entre dos programas: un programa que se ejecutará en el microcontrolador de propósito general (*gpcore*) y un programa que se ejecutará en el microcontrolar de control de motores (*motcortl*).

Inicialmente se planteará que el programa *gpcore* se encargue de las siguientes tareas:

- Comunicación RS232 con el PC host
- Interpretación de las instrucciones enviadas desde el PC host
- Gestión de la E/S
- Comunicación con GPMCU para el manejo de motores (mediante I2C o SPI)

Por su parte, el programa *motorctl* se encargaría de las siguientes tareas:

- Señalización PWM
- Lectura de los codificadores de posición incrementales (módulo QEI[34])
- Control realimentado de los motores mediante bucle PID
- Comunicación con MCMCU para el manejo de los motores (mediante I2C o SPI)

A medida que se vaya realizando el diseño y desarrollo del sistema, se comprobará si esta división de tareas se adapta bien a las necesidades y posibilidades del sistema y, de no ser así, se reacomodará el reparto de tareas en la misma fase de diseño y experimentación.

[34] Quadrature Encoder Interface

### 3.4.3 Repertorio de instrucciones soportado

El repertorio de instrucciones del controlador Rhino Mark IV se compone de más de 80 instrucciones diferentes, de las cuales muchas no son esenciales para el uso del sistema robótico y su utilización en las prácticas de la asignatura es infrecuente, mientras que un subconjunto del repertorio es fundamental en el uso del sistema y en el desarrollo de las prácticas de la asignatura.

Por ello, y para limitar la extensión de este proyecto, deberá restringirse el conjunto de instrucciones implementadas en RhinoChip OS a aquellas instrucciones fundamentales para el uso del sistema robótico. Esto permitirá, además, producir un primer prototipo funcional del controlador que pueda emplearse para controlar el robot, a pesar de que el controlador continúe en desarrollo activo.

En definitiva, será necesario implementar un subconjunto reducido del repertorio de instrucciones del controlador. Para ello nos basaremos en la lista de comandos de uso frecuente comentada en la Sección 3.2.6 (pág. 26).
Con el fin de simplificar las tareas de desarrollo del primer prototipo y de producir un prototipo funcional desde un estadio temprano del desarrollo, se implementará la mayor parte de los comandos de uso frecuente propuestos, pero no todos. Los tipos de instrucciones a implementar serán:

- instrucciones para la lectura de valores de configuración y del estado del controlador,
- instrucciones para la escritura de la configuración y la modificación del estado del controlador,
- instrucciones para la realización del hard home,
- instrucciones para la lectura de posición de los motores,
- instrucciones de movimiento en el espacio de las articulaciones (tanto movimiento relativo como absoluto).

Con los comentarios realizados en la Sección 3.2.6 (pág. 26) respecto a la frecuencia e importancia de cada comando, y siguiendo los criterios aquí expuestos, se elegirá el subconjunto de comandos a implementar en el mismo momento del diseño e implementación, lo cual se detalla en el capítulo correspondiente.

Nótese que, si bien los requisitos del controlador (Sección 3.3.1, pág. 29) contemplaban que el controlador proporcionara una interfaz de E/S, en los criterios recién expuestos no se incluyen los comandos para la gestión de la E/S. Como ya se comentó en la Sección 3.2.6 (pág. 26), esto es así porque la gestión de la E/S no es prioritaria y se plantea como adición opcional —en caso de que el desarrollo y la temporización del proyecto lo permitan— o como propuesta de trabajo futuro.

### 3.4.4 Herramientas de desarrollo

Las opciones de lenguaje proprocionadas para los dsPIC abarcan el ensamblador de la familia dsPIC y el lenguaje C con las bibliotecas y recursos sintácticos proporcionados por los compiladores de Microchip y de terceros para los PIC y dsPIC.
La programación de los dsPIC en lenguaje ensamblador proporciona escasas ventajas, a la par que conlleva serios inconvenientes:

1. La programación en ensamblador se realiza a muy bajo nivel y favorece los errores de programación.
2. La programación es poco intuitiva y más lenta.
3. El programa desarrollado presentará escasa portabilidad a otras arquitecturas o incluso a arquitecturas similares pero que cuenten con ciertas diferencias en el repertorio de instrucciones.
4. No permite la construcción de abstracciones para los datos.
5. La mejora en el rendimiento debida a la programación a tan bajo nivel es escasa y no compensa las desventajas del uso de este lenguaje.

Por su parte, la utilización del lenguaje C presenta ventajas notables respecto al ensamblador:

1. Permite un desarrollo más rápido y menos propenso a errores.
2. La portabilidad del progama a otras arquitecturas es mucho mayor.
3. Permite la construcción de abstracciones para los datos.
4. Existen bibliotecas de funciones de alto nivel para la realización de diversas tareas comunes.
5. La disminución del rendimiento es escasa o nula debido a las optimizaciones que aplican los compiladores modernos.
6. En caso de que se desee o necesite utilizar instrucciones en ensamblador, el lenguaje C permite escribir secciones de código directamente en lenguaje ensamblador embebido en el código C.

Por tanto, el lenguaje a emplear en el desarrollo será el lenguaje C con las bibliotecas y recursos sintácticos detallados por Microchip en su especificación del compilador C30. Esta especificación proporciona utilidades sintácticas (principalmente macros) especialmente diseñadas para los microcontroladores de Microchip, además de especificar bibliotecas de funciones para el manejo de los diversos periféricos de los microcontroladores (módulos de PWM, QEI, E/S, comunicación, etc.), las cuales pueden resultar útiles en algún momento, para asegurar un desarrollo más rápido y menos propenso a fallos.

En cuanto al entorno de desarrollo, se empleará el entorno MPLAB IDE en su versión 7 ó posterior, ya que es el entorno de desarrollo oficial de Microchip y está completamente integrado con los compiladores y los programadores para los microcontroladores. Puesto que este entorno está sólo disponible para el sistema operativo Windows, se requerirá un PC con Windows XP o posterior para el desarrollo.

### 3.4.5 Modelo de datos de RhinoChip OS

El software RhinoChip OS es una aplicación intensiva en datos, puesto que (1) debe mantener información acerca del estado del sistema y de su configuración, (2) debe responder a las peticiones de datos, tanto de los diferentes componentes del sistema software como del propio usuario, y (3) su comportamiento varía sustancialmente según los datos almacenados. En este sistema, las estructuras de datos y los algoritmos que trabajan con esos datos ocupan el lugar central de atención, por lo cual corresponde hacer un diseño modular con un enfoque de programación estructurada, muy acorde a las características de este tipo de aplicaciones.

Esto significa que deberá realizarse un diseño de un modelo de datos y de las estructuras de datos apropiadas para dar soporte a las necesidades de información de la aplicación. Puesto que el lenguaje de programación a utilizar es el lenguaje C —aunque éste no proporciona mecanismos de orientación a objetos—, se emplearán los mecanismos del lenguaje para crear un diseño modular y definir estructuras y tipos de datos para simular cierto grado de orientación a objetos y así definir tipos abstractos de datos que den soporte a las necesidades de la aplicación.

En la Sección 3.2.5 (pág. 25) se ha enumerado algunos de los datos fundamentales que debe almacenar el sistema. Para estos y otros datos se irá diseñando un modelo de datos adecuado a lo largo de la fase de diseño. En dicha fase se diseñarán las estructuras de datos apropiadas a medida que vayan surgiendo necesidades impuestas por la aplicación.

## 3.5 Análisis de MarkCommander

### 3.5.1 Características del software de utilidad MK4UTIL

El programa de utilidad que originalmente se distribuía con los sistemas robóticos de Rhino Robotics Ltd. es una utilidad en línea de comandos para el sistema operativo MS-DOS denominada MK4UTIL que permite

- el envío de comandos al controlador Rhino Mark IV,
- la transferencia y recepción de programas de usuario, y
- la configuración del controlador.

Si bien existen utilidades en línea de comandos muy avanzadas y cuyo uso es bastante cómodo una vez se está acostumbrado a este tipo de interfaz, la utilidad MK4UTIL no es una

de ellas. Aparte de por no disponer de interfaz gráfica, la forma en que está implementado el terminal de envío de comandos desde el PC host al controlador Mark IV dentro de MK4UTIL hacen que su uso sea muy poco intuitivo. Otra carencia muy destacada es que no muestra los comandos enviados anteriormente en la misma sesión, sino que sólo es posible visualizar el último comando que ha sido enviado y su respuesta.

En la actualidad, con el desfase del sistema MS-DOS y el advenimiento de los sistemas operativos con entorno gráfico, la utilidad MK4UTIL basada en línea de comandos ha quedado totalmente desfasada y, en su lugar, sería deseable disponer de una utilidad con interfaz gráfica de uso cómodo y sencillo. Además, dicha utilidad debería estar desarrollada para plataformas modernas[35] e, idealmente, ser multiplataforma —es decir, que pueda ejecutarse en diferentes sistemas operativos—, ya que en los últimos tiempos los sistemas operativos Linux y Mac OS X se han abierto paso en la docencia y en los PCs de los alumnos, tanto en la Universidad de Las Palmas de Gran Canaria como en otros centros de enseñanza superior.

### 3.5.2 Requisitos de MarkCommander

#### Requisitos generales

1. Debe proporcionar una interfaz gráfica.
2. Debe ser multiplataforma o fácilmente portable a otras plataformas, aunque el desarrollo deberá realizarse primeramente para Windows XP, ya que es el sistema operativo usado en las prácticas de la asignatura hasta el momento.
3. Debe permitir el envío de instrucciones al controlador y la visualización de la respuesta (por medio del puerto serie).
4. Debe permitir hacer el hard home del robot (envío de las instrucciones para ejecutar el hard home mediante acciones de la interfaz).
5. Debe permitir el envío y recepción de programas (cargándolos y guardándolos en ficheros en el sistema operativo).
6. Debe permitir la configuración del controlador (visualización y edición de la configuración mediante acciones de la interfaz).
7. Debe permitir la resolución de la cinemática del manipulador, tanto la cinemática directa como la cinemática inversa.

#### Vistas de la interfaz

La aplicación tendrá una única ventana (Figura 3.13, pág. 50) con una barra de menús, una barra de herramientas con botones para las acciones más frecuentes de los menús, y una

[35] Para ejecutar MK4UTIL bajo Windows XP es necesario utilizar el emulador DOSbox, y el funcionamiento de MK4UTIL bajo DOSbox es deficiente.

vista de pestañas donde se podrá elegir cualquiera de las tres vistas de la aplicación: Terminal, Cinemática directa y Cinemática inversa.

La pestaña del *Terminal* (Figura 3.14, pág. 50) permitirá enviar instrucciones en crudo al controlador, para lo cual mostrará un cuadro de texto que simulará la pantalla o monitor de un terminal de texto, y una línea de entrada de texto que simulará el *prompt* del terminal. Junto a la línea de entrada de texto habrá un botón que ordenará a la aplicación que envíe la instrucción tecleada.

La pestaña de la *Cinemática directa* (Figura 3.15, pág. 51) permitirá introducir los valores de las variables de articulación mediante *widgets* de tipo *spinbox* o mediante ruedas. En la parte inferior se mostrará la matriz resultante de resolver la cinemática directa para los valores de las variables de articulación introducidos, y justo al lado se mostrarán las instrucciones que se enviarán al controlador para realizar el movimiento especificado, y un botón para ejecutar dicho movimiento.

La pestaña de la *Cinemática inversa* (Figura 3.16, pág. 51) permite introducir la matriz del brazo, y mostrará en la parte inferior los valores de las variables de articulación correspondientes a dicha matriz, usando para ello *widgets* de tipo *spinbox* y ruedas de sólo lectura. Justo al lado se mostrarán las instrucciones que se enviarán al controlador para realizar el movimiento, y un botón para ejecutar dicho movimiento.

### Menús de la aplicación

A continuación se detalla la estructura propuesta para la barra de menús de la aplicación MarkCommander:

- **Terminal**
  - **Conectar con el controlador** establece una nueva conexión con el controlador a través del puerto serie.
  - **Guardar texto del terminal...** permite guardar un registro de todos los comandos enviados durante la sesión y la respuesta recibida para cada uno de ellos (si el comando debía producir una respuesta).
  - **Borrar pantalla del terminal** borra la pantalla del terminal, eliminando los comandos enviados durante la sesión en marcha hasta el momento, para que la pantalla quede vacía.
  - *(separador)*
  - **Recibir programa** ordena al controlador que envíe al PC el programa almacenado en memoria, y pregunta al usuario dónde desea guardar el programa en el disco duro local.
  - **Transmitir programa** permite al usuario seleccionar un programa del disco duro local y enviarlo al controlador, para que éste lo almacene en su memoria.
  - *(separador)*

Figura 3.13: Vista de la ventana principal de MarkCommander

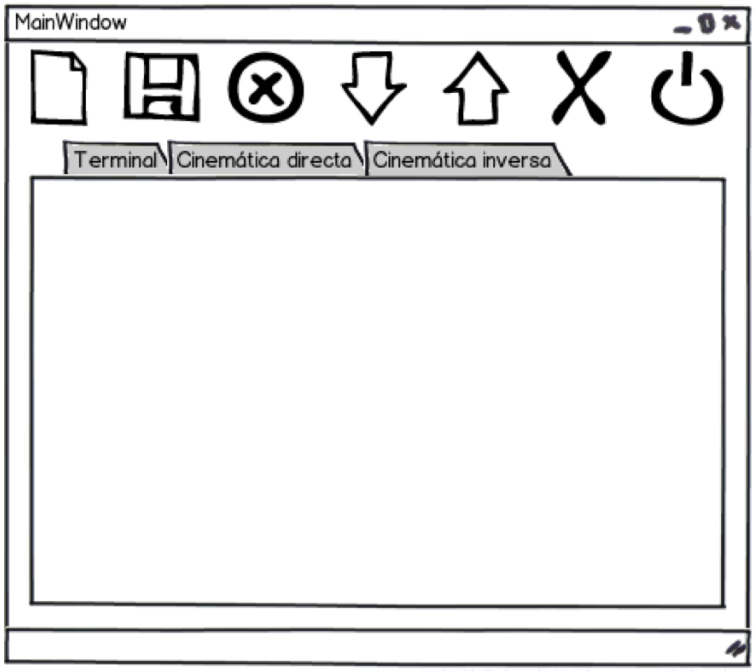



Figura 3.14: Vista de la pestaña *Terminal* de MarkCommander

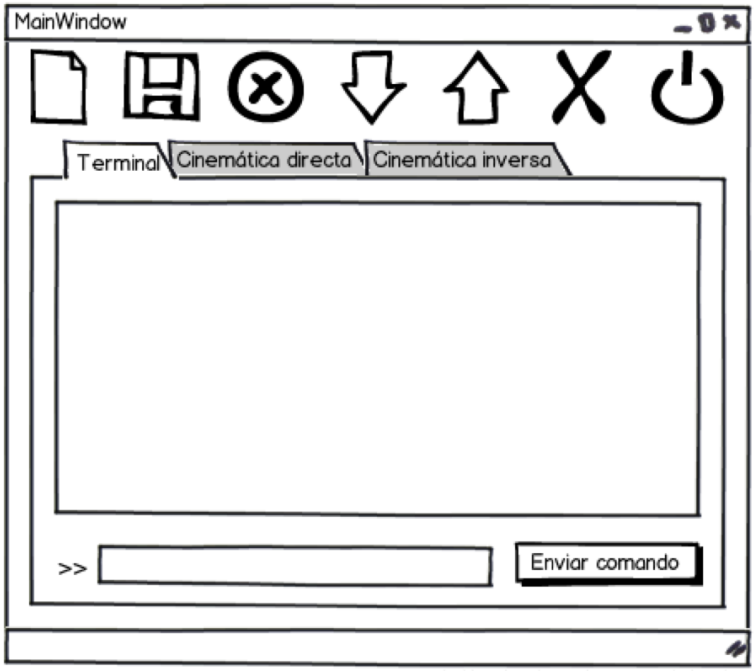


Figura 3.15: Vista de la pestaña *Cinemática directa* de MarkCommander

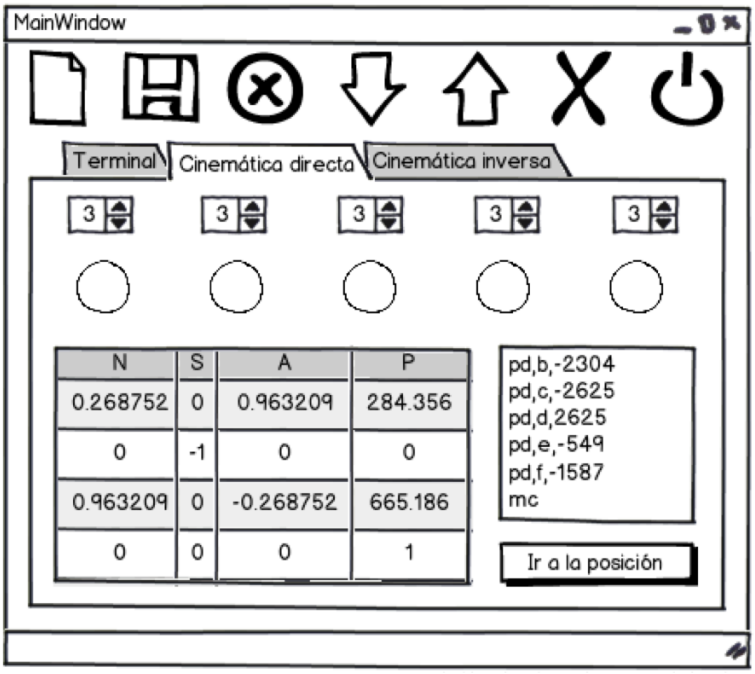
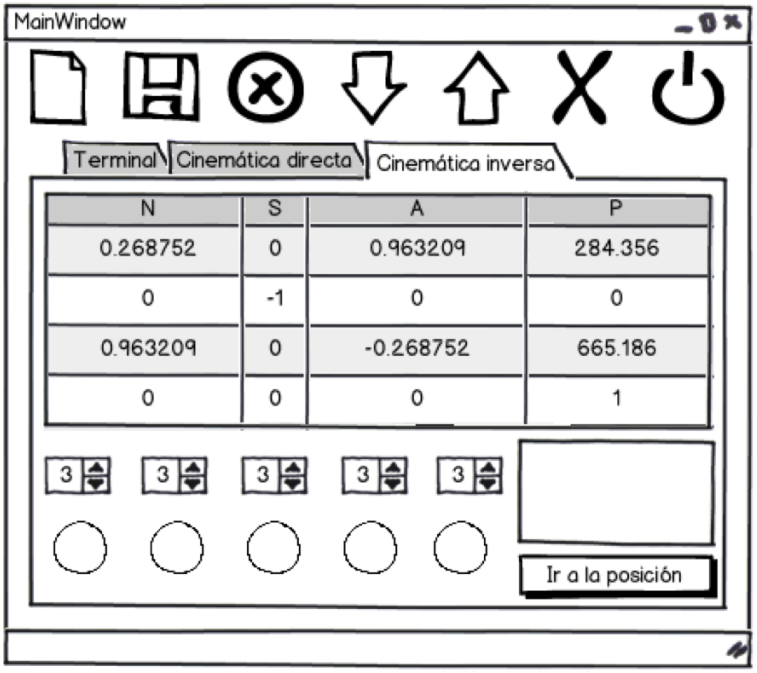



Figura 3.16: Vista de la pestaña *Cinemática inversa* de MarkCommander



  - **Cerrar conexión** termina la sesión de comandos, cerrando el puerto serie de manera limpia, para que no quede bloqueado y pueda ser usado en una nueva sesión de MarkCommander o por otra aplicación del sistema operativo.
  - **Salir de MarkCommander** cierra la conexión abierta (si la hay) y termina la ejecución de la aplicación.

- **Robot**
  - **Hacer hard home** envía al controlador la orden de ejecución del hard home.
  - **Ir a soft home** envía al controlador la orden de ejecución del soft home.
  - **Ir a home de la cinemática** envía al controlador las órdenes de movimiento necesarias para llevar el brazo a la configuración de home propuesta en clase para el estudio de la cinemática del robot.
  - *(separador)*
  - **Configurar controlador...** permite visualizar y modificar la configuración del controlador almacenada en los registros del mismo, proporcionando una interfaz gráfica para los comandos implicados.

- **Ventana**
  - **Terminal** cambia la vista de la aplicación a la pestaña *Terminal*.
  - **Cinemática directa** cambia la vista de la aplicación a la pestaña *Cinemática directa*.
  - **Cinemática inversa** cambia la vista de la aplicación a la pestaña *Cinemática inversa*.

- **Ayuda**
  - **Ayuda de MarkCommander** muestra la guía de uso de MarkCommander.
  - **Referencia de comandos de Rhino Mark IV** muestra una tabla resumen del repertorio de comandos del controlador, sus parámetros y su significado, a modo de referencia rápida para interactuar con el controlador sin tener que acudir al manual cada vez que se desee consultar la forma de uso de un comando.
  - **Códigos de error de Rhino Mark IV** muestra una tabla resumen de los códigos numéricos de error utilizados por el controlador y el significado de cada uno de ellos.
  - *(separador)*
  - **Sobre MarkCommander...** muestra un mensaje acerca de la autoría de MarkCommander y de su utilidad.

### Herramientas de desarrollo

Existe una gran variedad de herramientas que permiten el desarrollo de aplicaciones multiplataforma con interfaz gráfica. Para facilitar la elección de las herramientas, se impondrán los siguientes requisitos:

1. Debe permitir el desarrollo multiplataforma con el menor número de cambios en el código del proyecto.
2. Debe permitir el diseño de interfaces gráficas, a ser posible nativas (es decir que la interfaz gráfica presente el mismo aspecto visual que las aplicaciones desarrolladas con las herramientas de desarrollo propias de cada plataforma).
3. Debe permitir el desarrollo rápido de la aplicación y ser sencillo de usar. El entorno de desarrollo y el uso diario de las herramientas no deben suponer un obstáculo para el avance del desarrollo.
4. Debe soportar la orientación a objetos.
5. Debe disponer de una documentación completa, sencilla y bien estructurada, para facilitar el aprendizaje y la consulta, de manera que no se obstaculice el desarrollo.
6. Debe permitir el manejo del puerto serie o poder integrarse fácilmente con alguna biblioteca multiplataforma para el manejo del puerto serie.
7. Debe permitir el uso de un lenguaje de programación conocido, para evitar en todo lo posible el retraso producido por la necesidad de aprender un nuevo lenguaje.
8. Debe ser gratuito e, idealmente, software libre.

A causa del requisito de que las herramientas de desarrollo permitan el desarrollo multiplataforma y la portabilidad del código con el menor número de cambios posible, el uso de las APIs nativas de cada plataforma queda descartado. Otras opciones populares y que cumplen algunos de estos requisito podrían ser:

- Swing (Java)
- AWT (Java)
- Gtk+ (C++)
- wxWdigets (C++)
- Qt (C++)
- PyQt (Python)
- Gtk# (C# bajo Mono[36])

[36] http://www.mono-project.com/

Si bien todas estas opciones permiten el desarrollo de aplicaciones multiplataforma, algunas no permiten el desarrollo de una interfaz nativa ni la integración plena de la aplicación con el sistema operativo, como serían los casos de las bibliotecas basadas en Java o el caso de Gtk+ y Gtk#. Además, la documentación de algunas de estas opciones, como wxWdigets, no satisface los requisitos impuestos.
En cualquier caso, lo que sí es evidente es que, de entre todos los lenguajes disponibles, C++ parece ser el más prometedor, puesto que posibilita la orientación a objetos a la par que es más probable que permita el manejo del puerto serie, ya que, debido a la popularidad de C++, deben existir bibliotecas para tal fin o, en su defecto, se puede hacer uso de las APIs del sistema, lo cual no sería tan sencillo con los otros lenguajes. Además, todos los demás lenguajes requerirían la instalación de algún tipo de máquina virtual o intérprete, lo cual, si se puede evitar, es deseable evitarlo.

De todas las opciones presentadas, la más interesante es Qt, puesto que satisface los requisitos de portabilidad y multiplataforma, es gratuito para uso no comercial, y es software libre. Además, se basa en C++, con lo que obtenemos los beneficios ya comentados, así como la velocidad de ejecución (especialmente importante para los cálculos matemáticos con matrices). Adicionalmente, el uso de las herramientas y el diseño de las interfaces gráficas con Qt es muy rápido y sencillo —en especial tras la introducción del entorno de desarrollo integrado QtCreator—, y la documentación del entorno y las bibliotecas es completa y está bien estructurada.
Aparte de las APIs para la creación de aplicaciones con interfaz gráfica, el framework de Qt proporciona una gran cantidad de bibliotecas con clases y funciones de todo tipo, desde estructuras de datos comunes (vectores, listas, tablas hash, árboles), pasando por cadenas de caracteres con soporte Unicode, hasta APIs para el uso de mecanismos IPC[37] o programación multi-hilo, todo ello completamente multiplataforma.
Si bien el gran conjunto de bibliotecas proporcionado por Qt parece no incluir ninguna funcionalidad para el manejo del puerto serie, se ha encontrado en internet una biblioteca de software libre creada con Qt y C++ para dicha tarea: QextSerialPort[38]. Esto favorece más aun la elección de Qt como herramienta de desarrollo, y hace que Qt finalmente cumpla todos los requisitos impuestos.

En conclusión, para el desarrollo de MarkCommander se empleará el framework Qt en su versión 2010.05, que es una de las últimas versiones estables disponibles en el momento del análisis de esta aplicación. La implementación se realizará, por tanto, en el lenguaje C++ siguiendo una metodología de diseño orientado a objetos. Finalmente, para el manejo del puerto serie se utilizará la biblioteca QextSerialPort.

## 3.6 Recursos materiales

Dada la naturaleza práctica de este proyecto, el análisis y la planificación de los recursos necesarios es muy importante. Puesto que en el caso de este proyecto no se admite libertad en

[37] *Inter-Process Communication*
[38] https://code.google.com/p/qextserialport/

la planificación de recursos humanos, sólo nos ocuparemos de los recursos materiales y de los temporales (véase la Sección 3.7, pág. 56).

Sin embargo, la estimación de los recursos materiales es un aspecto particularmente complejo. En primer lugar, a causa de la naturaleza de este proyecto, hay que distinguir entre una "fase de prototipado" y una "fase de implementación". Por "fase de prototipado" nos referimos a una fase inicial del proyecto en la que se llevan a cabo experimentos y se investiga la mejor forma de llevar a cabo el diseño. En esta fase será necesario realizar montajes temporales del sistema, particularmente de la parte hardware, para comprobar su funcionamiento y ver si se adecúa a las necesidades del proyecto. Por su parte, en la "fase de implementación" se llevará a cabo la fabricación de un prototipo, entendiendo esto como un montaje igual o muy similar al sistema definitivo, aunque todavía en pruebas. Esto significa que el sistema en la fase de prototipado será muy cambiante y requerirá muchos cambios en el montaje, por lo que serán necesarios muchos componentes que no se pueden estimar a priori. En cambio, en la fase de implementación se usará un conjunto de componentes bien definidos como resultado de las pruebas realizadas durante la fase de prototipado.

Si bien la estimación exacta de todos los componentes necesarios no se puede realizar a priori por los motivos ya expuestos, sí es posible hacerse una idea de qué tipo de elementos y componentes se va a necesitar. De esta forma, teniendo en cuenta las decisiones establecidas en el análisis, tanto para la programación como para el prototipado del hardware se necesitarán los siguientes recursos:

- Placa de desarrollo Microchip dsPICDEM 2
- Microcontrolador de propósito general dsPIC30F4013
- Microcontrolador de control de motores dsPIC30F4011
- Dispositivo de programación y depuración Microchip ICD 2 (In-Circuit Debugger)
- Ordenador personal moderno con sistema operativo Windows XP o superior
- Entorno de desarrollo MPLAB IDE
- Material de electrónica en general
  - Placas de prototipado (protoboards)
  - Cables variados (tanto para realización de conexiones en las placas de prototipado como para la comunicación RS232-C)
  - Resistencias y condensadores
  - Diodos y chips de puente en H
  - Diodos LED
  - Otros chips de funciones diversas
- Dispositivos de medición y monitorización
  - Multímetro digital

  - Osciloscopio
  - Terminales de vídeo (para la supervisión de la comunicación RS232-C)

- Brazos robóticos de Rhino Robotics[39]

## 3.7 Plan de trabajo

Antes de comenzar con el diseño y el desarrollo del proyecto, es necesario familiarizarse con las tecnologías a emplear, principalmente para el desarrollo de la plataforma RhinoChip. Para conocer en profundidad la tecnología de los microcontroladores de la familia dsPIC30F y aprender a utilizar los módulos necesarios para la implementación de las funcionalidades detalladas en la Sección 3.4.2 (pág. 44), se aprovechará el enfoque de desarrollo basado en experimentos propuesto en la Sección 2 (pág. 13) para definir una serie de experimentos que permitan aprender el funcionamiento, utilización y programación de los microcontroladores a la par que se permita utilizar el resultado de los experimentos como base para construir el sistema. Dicho de otra forma, en los experimentos se buscará aprender la forma de utilización de los diferentes componentes de los microcontroladores a la vez que se orientará el desarrollo de los experimentos a diseñar e implementar las funcionalidades finales del prototipo.

La primera fase comenzará por tanto con el aprendizaje de las herramientas y del entorno de desarrollo. Esto incluye la familiarización con la placa de desarrollo Microchip dsPICDEM 2, con el entorno de desarrollo y con la arquitectura de los microcontroladores. Además, se estudiará los programas de ejemplo y se aprenderá a crear el esqueleto de un programa para los microcontroladores.

Una vez conocido el entorno de desarrollo y la arquitectura del hardware, comenzará la siguiente fase, de desarrollo, en la que se comenzará a estudiar el funcionamiento y modo de uso de los diferentes módulos de los microcontroladores mediante el desarrollo de diversos experimentos, cuyo objetivo será doble: por una parte, aprender el modo de uso y programación de cada módulo hardware y, por otra parte, diseñar los subsistemas del sistema final. Por tanto, en esta fase ya comienza el diseño y la implementación del sistema, al mismo tiempo que se continúa el aprendizaje (en mayor profundidad que en la fase inicial de aprendizaje) a medida que se vaya necesitando.
Además, durante esta fase de desarrollo se hará igual énfasis tanto en el diseño e implementación como en la verificación (realización de pruebas) de todos los subsistemas a medida que se van diseñando. Esto evitará por un lado la realización de pruebas a posteriori y mejorará la robustez del prototipo a medida que se va desarrollando, a la vez que permitirá un desarrollo más ágil en el que los posibles fallos se corrijan con mayor prontitud.
Puesto que los sistemas a desarrollar son realmente dos, la plataforma RhinoChip y el software de aplicación MarkCommander, la fase de desarrollo realmente estará dividida en dos: una fase de desarrollo para el primer sistema, RhinoChip, y otra fase de desarrollo para el segundo sistema, MarkCommander; si bien ambas fases de desarrollo seguirán un enfoque similar,

[39] Como mínimo, el brazo Rhino XR-4; el SCARA es opcional dentro del ámbito de este proyecto, pero disponer de él permitiría el desarrollo de pruebas más exhaustivas.

Tabla 3.6: Planificación temporal del proyecto

| Fase | Descripción | Duración estimada | Tareas propuestas |
|---|---|---|---|
| Aprendizaje | Familiarización con las herramientas y el entorno de desarrollo, así como con los microcontroladores y su forma de programación | 3–4 semanas | • Instalación y primeros pasos en el entorno de desarrollo MPLAB<br>• Conexión y puesta a punto de la placa de desarrollo Microchip dsPICDEM 2 y el programador Microchip ICD 2<br>• Estudio, compilación y ejecución de los programas de ejemplo de los microcontroladores |

| Fase | Descripción | Duración estimada | Tareas propuestas |
|---|---|---|---|
| Desarrollo de RhinoChip | Diseño, implementación y verificación de los diferentes componentes del sistema RhinoChip | 4–6 meses | • Estudio del funcionamiento de los motores del manipulador robótico<br>• Realización de experimentos para el aprendizaje del modo de uso de los microcontroladores y sus módulos (puertos de E/S, UARTs, módulo QEI, módulo de PWM, módulos I2C y SPI)<br>• Diseño global del sistema<br>• Especificación del lenguaje de comandos y su gramática<br>• Diseño de los susbsistemas<br>• Diseño de un protocolo de comunicación entre los microcontroladores |
| Desarrollo de MarkCommander | Diseño, implementación y verificación del software de aplicación MarkCommander | 1–2 meses | • Búsqueda y familiarización con bibliotecas multiplataformas para la comunicación RS232 desde un PC<br>• Diseño de la interfaz gráfica de la aplicación<br>• Diseño del modelo de datos y la arquitectura de clases de la aplicación<br>• Verificación del funcionamiento de la aplicación junto con el sistema RhinoChip |

según el cual se realizará tanto el diseño, como la implementación y la verificación del sistema en cuestión, haciendo énfasis también en la verificación (prueba) y la producción de documentación a lo largo del propio proceso de desarrollo, no dejando estas tareas para más tarde.

El desarrollo del proyecto concluirá, finalmente, con la redacción de una memoria explicativa en la que se recogerá todos los detalles del proceso llevado a cabo. La redacción de esta memoria no comenzará desde cero, sino que se recopilará toda la documentación producida a lo largo de todo el transcurso del proyecto y se reunirá en un solo documento, realizando las mejoras y adiciones necesarias.

En la Tabla 3.6 (pág. 57) se puede ver el resumen de la planificación explicada, junto con la duración estimada para cada fase y algunas tareas a desarrollar en cada fase. Si bien la lista de tareas no pretende ser exhaustiva, sí se tiene la expectativa de que se ajuste a las tareas a desarrollar en la realidad.

# Referencias


[1] Robert Joseph Schilling. *Fundamentals of Robotics: Analysis and Control*. Prentice Hall, 1990.

[2] Antonio Barrientos, Luis Felipe Peñín, Carlos Balaguer, and Rafael Aracil. *Fundamentos de Robótica*. McGraw-Hill, 1997.

[3] *Rhino Mark IV Owner's Manual*.

[4] mbed cookbook. https://mbed.org/cookbook/Homepage, .

[5] Sitio web oficial de Arduino. http://www.arduino.cc/, .

[6] Página web de Arduino Uno. http://is.gd/SiSPg1.

[7] Página web de Arduino Leonardo. http://is.gd/j11BTP.

[8] Página web de Arduino Due. http://is.gd/fpHVW3.

[9] Página web de Arduino Robot. http://is.gd/u4nGVL.

[10] Página web de Arduino ADK. http://is.gd/QtiJku.

[11] Página web de Arduino Mega 2560. http://is.gd/VPzRMr.

[12] ATmega328P Data Sheet. http://is.gd/tX87Bp, .

[13] ATmega32U4 Data Sheet. http://is.gd/oBkure, .

[14] ATmega2560 Data Sheet. http://is.gd/76lcqg, .

[15] AT91SAM3X8E Data Sheet. http://is.gd/RJnAaY.

[16] Tienda online de Arduino. http://store.arduino.cc/, .

[17] dsPIC30F3012. http://is.gd/LSzroF, .

[18] dsPIC30F4011. http://is.gd/HgEr3t, .

[19] dsPIC30F4013. http://is.gd/EUp73Z, .

[20] dsPIC30F Digital Signal Controllers Brochure. http://is.gd/yvMIjM.

[21] dsPICDEM 2 Development Board User's Guide. http://is.gd/AhZtS6, .

[22] dsPIC30F Family Reference Manual. http://is.gd/CQ0eLy, .

[23] dsPIC30F Data Sheet General Purpose and Sensor Family. http://is.gd/fwnaUM, .

[24] dsPIC30F Data Sheet Motor Control and Power Conversion Family. http://is.gd/aaN4xR, .

[25] dsPIC30F3014/4013 Data Sheet. http://is.gd/r1kgHl, .

[26] dsPIC30F4011/4012 Data Sheet. http://is.gd/hBsQPY, .

[27] Tienda online de microchip. https://www.microchipdirect.com/.

[28] Sitio web oficial de la plataforma mbed. https://mbed.org/, .

[29] Artículo en Wikipedia sobre el microcontrolador mbed (en inglés). http://is.gd/q1JFhL, .

[30] Artículo en Wikipedia sobre la plataforma mbed (en inglés). http://is.gd/HDYeLe, .

[31] Introducción a la plataforma mbed en TecBolivia.com. http://is.gd/IE6Lzo, .

[32] mbed microcontroller variants. http://is.gd/khi7jA, .

[33] mbed NXP LPC11U24. http://is.gd/QVEvB3, .

[34] mbed NXP LPC1768. http://is.gd/5yCWgQ, .

[35] LPC11U2X data sheet. http://is.gd/oemgwr, .

[36] LPC1768 data sheet. http://is.gd/sQMqhn, .

[37] mbed platforms. http://mbed.org/platforms/, .

[38] Order mbed. http://mbed.org/handbook/Order, .

# 4 Diseño e implementación

## 4.1 Diseño e implementación del controlador RhinoChip

Como se indicó en la Sección 2 (pág. 13), la metodología de desarrollo a emplear en el diseño e implementación del controlador RhinoChip (tanto del hardware como del software) seguirá un enfoque ágil "basado en experimentos" que permitirá (1) aprender el modo de uso y programación de los microcontroladores conforme se va necesitando, (2) elaborar el diseño de forma modular e iterativa, y (3) conseguir la integración de los módulos de hardware y software diseñados en cada etapa.
Nótese que, puesto que se trata de un sistema en el que la integración entre hardware y software es fundamental, no es posible separar el diseño de uno del diseño del otro, por lo que, basándonos en la organización hardware propuesta en la Sección 3.3.3 (pág. 42) (véase la Figura 3.12, pág. 42), comenzaremos con los experimentos necesarios para ir construyendo el sistema, tanto en términos de diseño como de implementación de un prototipo.

### 4.1.1 Programa principal y configuración del microcontrolador

El primer paso para aprender a crear un programa en C para los dsPIC es familiarizarse con el entorno de desarrollo MPLAB IDE y conocer la estructura básica de cualquier programa.

En el CD de las herramientas de desarrollo de Microchip que viene con la placa de desarrollo se puede encontrar un programa de ejemplo en C que utiliza diferentes periféricos del microcontrolador. Dicho código de ejemplo forma parte de un *proyecto de MPLAB IDE*. Si abrimos el *fichero de proyecto* correspondiente al microcontrolador que vayamos a usar, podremos ver los ficheros de código del programa de ejemplo. Siguiendo los pasos de la documentación de MPLAB IDE se puede conectar fácilmente el programador ICD 2, compilar el programa y programar el microcontrolador con el binario recién compilado, para así probar el programa de ejemplo.

Tras la instalación de MPLAB IDE, un primer vistazo al programa puede ser intimidante,

debido a la cantidad de menús y opciones de que dispone. Sin embargo, siguiendo los capítulos introductorios del manual de MPLAB IDE, crear un nuevo proyecto de programa en C para un microcontrolador determinado también resulta bastante simple. Basta con ir al menú *Project → New project with wizard...* y seguir los pasos del asistente que sale en pantalla.
Esto nos proporcionará un proyecto vacío en el que poder crear un nuevo programa. Estudiando el código de ejemplo mencionado antes, podemos ver que un programa en C para el microcontrolador es muy similar a cualquier programa C que hayamos hecho (debe tener una función `main`, que debe devolver un entero, y donde irá todo el código de nuestro programa), con el añadido de que

1. se debe incluir el fichero de cabecera correspondiente al microcontrolador objetivo, y
2. se deben configurar ciertos parámetros del microcontrolador, para que el programa funcione adecuadamente.

Listado 4.1: Estructura básica de un programa en C para los microcontroladores dsPIC30F

```
1  #include <p30fxxxx.h>
2
3  // Set processor configuration bits for the dsPIC30F4013
4  _FOSC(CSW_FSCM_OFF & XT_PLL16); // Fosc = 16x 7.37 MHz, Fcy =
       29.50 MHz
5  _FWDT(WDT_OFF);                 // Turn off the watchdog timer
6  _FBORPOR(MCLR_EN & PWRT_OFF);   // Enable reset pin and turn
       off the power-up timers.
7
8  int main(void)
9  {
10     // code
11
12     return 0;
13 }
```

La inclusión del fichero de cabecera del microcontrolador es vital, puesto que, de no estar presente, no se podrá acceder a los registros del microcontrolador, por lo que no se podrá realizar ningún tipo de tarea que requiera la interacción con los periféricos.
Para los microcontroladores de la familia dsPIC30F, el nombre del fichero de cabecera de cada microcontrolador sigue el patrón `p30fabcd.h`, donde `abcd` debe sustituirse por el número de cuatro cifras que identifica al microcontrolador. Así, por ejemplo, el fichero de cabecera del dsPIC30F4013 se llama `p30f4013.h`, y el del dsPIC30F4011 se llama `p30f4011.h`.
Sin embargo, para mayor sencillez y portabilidad del código, también existe un fichero de cabecera con el nombre `p30fxxxx.h` que identifica el microcontrolador objetivo en tiempo de compilación (de acuerdo al microcontrolador objetivo especificado en la creación del proyecto) e incluye el fichero de cabecera correspondiente.

El otro añadido respecto a cualquier otro programa en C es la configuración de parámetros del microcontrolador. Este aspecto también es de vital importancia para que el programa

funcione correctamente, puesto que algunos de los parámetros que se pueden configurar son la frecuencia de reloj o el *watchdog*[1].
La configuración de estos y otros parámetros se puede llevar a cabo mediante ciertos registros especiales del microcontrolador destinados precisamente a almacenar la configuración. Sin embargo, en el fichero de cabecera del microcontrolador se definen unas macros de C que también permiten modificar esta configuración, pero de una forma más cómoda.
Estas macros son las que se utilizan en el código del Listado 4.1 (pág. 62) para (1) establecer la frecuencia de reloj a 29.50 MHz, (2) desactivar el *watchdog*, y (3) habilitar el reinicio del microcontrolador cuando se pulsa el botón de *reset* de la placa.

Sabiendo estas cosas ya es posible empezar a crear programas para los microcontroladores, tanto para el dsPIC30F4013 como para el dsPIC30F4011. Sin embargo, antes conviene hacer una última aclaración respecto a la frecuencia de ciclo.
Como se puede leer en la hoja de especificaciones de la familia dsPIC30F, hay diferentes opciones para generar la señal de reloj del microcontrolador. Entre ellas se encuentran un oscilador RC interno al microcontrolador y la utilización de un oscilador externo (que debe instalar el diseñador de la placa donde se montará el dsPIC). En nuestro caso estamos usando el oscilador externo de cuarzo disponible en la placa dsPICDEM 2, que oscila con una frecuencia de 7.37 MHz.
Mediante el uso de la macro `XT_PLL16` se está activando los bits del registro de configuración del reloj del sistema que indican al microcontrolador que se usará un cristal externo (`XT` como abreviatura de *crystal*) y que utilice un bucle PLL[2] de ganancia 16 para generar la señal de reloj del sistema. Esto hace que la frecuencia del ciclo de instrucción sea

$$f_{\mathrm{CY}} = \frac{f_{\mathrm{OSC}} \cdot G_{\mathrm{PLL}}}{4}, \tag{4.1}$$

donde $f_{\mathrm{OSC}}$ es la frecuencia del oscilador (en nuestro caso, 7.37 MHz) y $G_{\mathrm{PLL}}$ es la ganancia del bucle PLL (en nuestro caso, 16). El denominador igual a 4 toma su valor debido a que cada ciclo del reloj del sistema es dividido internamente por cuatro para producir el reloj del ciclo de instrucción.
Puesto que en cada ciclo de reloj se ejecuta exactamente una instrucción, y esto es válido para cualquier tipo de instrucción de la arquitectura, una frecuencia de ciclo de 30 MHz equivale a una velocidad de CPU de 30 MIPS (millones de instrucciones por segundo).

[1] El *watchdog timer* es un temporizador que, si está activado, reinicia el microcontrolador cada vez que se produce el evento de *timeout*, a menos que el programa reinicie el temporizador por software antes de que el *timeout* ocurra. Esto se hace para evitar que el microcontrolador se quede bloqueado en un bucle infinito o en estado de interbloqueo, ya que los microcontroladores se usan en tareas de control críticas (p.ej. de procesos industriales) en las que no es admisible que el programa se quede bloqueado, puesto que ello significaría que el proceso ya no estaría siendo controlado.

[2] Un *Phase-Locked Loop* es un sistema de control en bucle cerrado que, dada una señal de entrada de cierta frecuencia, genera una señal de salida cuya frecuencia "persigue" la frecuencia de entrada hasta que la señal de salida tenga la misma frecuencia que la de entrada (o, de forma más general, un múltiplo de ella, como es nuestro caso).

### 4.1.2 Manejo de los pines de E/S

El siguiente paso en el proceso de familiarización con el microcontrolador y su programación consiste en aprender a manejar los puertos de E/S, haciendo para ello un programa que encienda y apague unos diodos LED periódicamente. Esto requerirá la configuración de los puertos de E/S que se vaya a usar para encender y apagar los LED.

Para empezar, es necesario elegir un pin de E/S y conectar un diodo en serie con una resistencia. Si se está usando la placa dsPICDEM 2, como es el caso, se puede utilizar los diodos `D3` y `D4` de la placa, que están conectados a los pines `RB0` y `RB1`, respectivamente. De no ser así, se deberá conectar un diodo al pin elegido, en una protoboard, con una resistencia en serie dimensionada adecuadamente.
Puesto que los pines de E/S del microcontrolador pueden proporcionar una corriente de hasta $25\,\mathrm{mA}$, se puede conectar un extremo del circuito directamente al pin de E/S y el otro extremo a tierra. Para dimensionar la resistencia, debe tenerse en cuenta que se alimentará el circuito con $V_{\mathrm{DD}} = 5\,\mathrm{V}$ y que la intensidad no puede superar los $25\,\mathrm{mA}$. Con una intensidad de entre 12 mA y 16 mA es más que suficiente para encender el LED, por lo que se puede usar una resistencia de $330\,\Omega$. De esta forma, la intensidad que se obtiene del pin será

$$I = \frac{V}{R} = \frac{5\,\mathrm{V}}{330\,\Omega} \simeq 15\,\mathrm{mA} < 25\,\mathrm{mA}. \tag{4.2}$$

Todos los microcontroladores disponen de múltiples puertos de E/S, cada uno de ellos con varios pines. En nuestro caso, utilizaremos los pines `RB0` y `RB1` (pines 0 y 1) del puerto B, tanto si usamos la placa dsPICDEM 2 como si no[3]. Para poder utilizar estos pines como pines de entrada o de salida, es necesario configurarlos en el registro `TRISx` correspondiente, donde `x` debe sustituirse por la letra del puerto de E/S a configurar. Estableciendo el bit $i$ del registro `TRISx` a uno (1) estaremos configurando el pin $i$ del puerto `x` como pin de entrada[4], y estableciendo el bit $i$ a cero (0) estaremos configurando el pin $i$ como pin de salida[5].
Para modificar uno de los bits del registro `TRISx` se puede utilizar operaciones bit a bit, o se puede aprovechar una estructura definida en el fichero de cabecera del microcontrolador y que permite acceder a un bit directamente. Esta estructura, de nombre `TRISxbits`, permite acceder a un bit concreto del registro `TRISx` mediante campos de nombre `TRISxi`, donde la `x` debe sustituirse por la letra del puerto y la `i` por la posición del bit a modificar.
Con todo esto, para configurar el pin `RB0` como pin de salida (para poder encender y apagar el LED), habría que asignar a la variable `TRISBbits.TRISB0` el valor cero (0) en el cuerpo de la función `main`, antes de poder modificar el valor de salida del pin.

Tanto para escribir un valor en un pin configurado como pin de salida, así como para leer dicho valor, el microcontrolador dispone de los registros `LATx`, que siguen una convención de nombres similar a los registros `TRISx`. Para leer o escribir un valor en el pin de salida `Ri`, bastaría con modificar el bit $i$ del registro `LATx` mediante operaciones binarias, o mediante la estructura `LATxbits`, lo cual se hace accediendo al campo `LATxi`.

[3] Aunque la placa ya dispone de LEDs, nosotros haremos la prueba de las dos formas, es decir, con los LEDs de la placa y con LEDs propios montados en una protoboard.
[4] La forma del símbolo 1 recuerda a la I de *Input*.
[5] La forma del símbolo 0 recuerda a la O de *Output*.

Por tanto, para encender el LED tendremos que poner el pin de salida correspondiente a 1, lo cual hará que el voltaje de salida sea $V_{DD} = 5\,\mathrm{V}$. Para ello, basta con asignar el valor uno (1) a la variable `LATBbits.LATB0` (suponiendo que queremos encender el LED conectado al pin `RB0`).

### 4.1.3 Interfaz para la comunicación entre GPMCU y el PC host

El siguiente experimento planteado consiste en el manejo de la UART para llevar a cabo la comunicación entre el microcontrolador de propósito general y un PC host mediante el estándar RS232-C, lo cual permitirá la recepción de los comandos enviados desde el PC host y el envío de respuestas a dichos comandos.
Para comprender el funcionamiento y modo de uso de la UART, se puede consultar el código de ejemplo del microcontrolador y las hojas de especificaciones. Luego se hará un pequeño programa que envíe un mensaje desde el GPMCU al PC host. Dicho mensaje podrá visualizarse en el PC host mediante un programa de terminal. HyperTerminal es una buena opción si se usa Windows XP, puesto que funciona bien y viene incluido en la instalación del sistema operativo. A partir de Windows 7, HyperTerminal ya no viene incluido con el sistema y requiere el pago de una licencia[6], pero se puede utilizar Termite[7] en su lugar, que es gratuito.
Finalmente, una vez comprendido el modo de uso de la UART del microcontrolador, se podrá crear un nuevo módulo de software para el programa *gpcore* del firmware RhinoChip OS que proporcione las funciones necesarias para abstraer el manejo de la UART al resto del programa. Esto permitirá construir un módulo de terminal o *shell* (intérprete de comandos) que pueda recibir los comandos enviados por el PC host y enviar las respuestas pertinentes sin necesidad de preocuparse por los detalles de manejo de la UART.

El microcontrolador dsPIC30F4013 dispone de dos UARTs, que se manejan por medio de varios registros especiales. En estos registros se configura el funcionamiento de la UART, se lee los datos recibidos y se escribe los datos a ser enviados. Dependiendo de la configuración de la UART realizada a través de los registros, existen dos formas de manejar el envío y recepción de datos: (1) mediante polling, o (2) mediante interrupciones. Si se decide usar polling, habrá que consultar los bits de los registros de control de la UART que indican si hay una transmisión en curso o si se han recibido datos antes de poder escribir en el buffer de transmisión o leer del buffer de recepción, respectivamente. En cambio, si se configura las interrupciones, no será necesario comprobar dichos bits de control, puesto que se producirá una interrupción cada vez que se reciba uno o más datos —ejecutándose en tal caso una rutina de interrupción para leer datos del buffer de recepción— o cada vez que haya hueco libre en el buffer de transmisión —ejecutándose entonces una rutina de interrupción para volver a escribir en el buffer de transmisión los siguientes datos a enviar—, según la configuración de interrupciones que se establezca.

El modo de uso más sencillo es el polling. Por ello, se desarrollará en primer lugar un pequeño programa que envíe un mensaje al PC host mediante polling. Para poder hacer esto, primero habrá que configurar la UART con los mismos parámetros de transmisión que en el PC

[6] http://www.hilgraeve.com/hyperterminal/
[7] http://www.compuphase.com/software_termite.htm

host. Por compatibilidad con el controlador Mark IV, se usará los parámetros de configuración de la Tabla 4.1. Una vez configurada la transmisión y activada la UART, se deberá consultar,

Tabla 4.1: Parámetros de configuración de la UART para la comunicación RS232-C

| Tasa de transferencia | 9600 baudios |
|---|---|
| Bits de datos | 8 |
| Bits de parada | 1 |
| Paridad | sin paridad |

mediante espera activa, el bit `UTXBF` del registro `U1STA` (si se ha elegido usar la UART 1) o del registro `U2STA` (si se ha elegido usar la UART 2). Mientras este bit tenga el valor 1, el buffer de transmisión estará lleno y no se podrá escribir más datos para ser enviados. En cuanto pase a tener el valor 0, significará que ha quedado al menos una posición libre en el buffer en la que se podrá escribir más datos. En ese momento se podrá añadir un dato al buffer FIFO escribiendo un byte en el registro `U1TXREG` o `U2TXREG`, según qué UART se desee utilizar. El Listado 4.2 muestra un ejemplo con la UART 2[8].

Listado 4.2: Ejemplo de transmisión RS232-C mediante polling

```
1  char buf[] = "Esto es una prueba";
2  char *next;
3
4  for (next = buf; *buf != '\0'; ++buf)
5  {
6      // Espera hasta que haya espacio libre en el buffer de
          transmision
7      while (U2STAbits.UTXBF);
8      // Escribe el siguiente byte en el buffer de transmision
9      U2TXREG = *next;
10 }
```

Mediante esta prueba se consiguen dos objetivos:

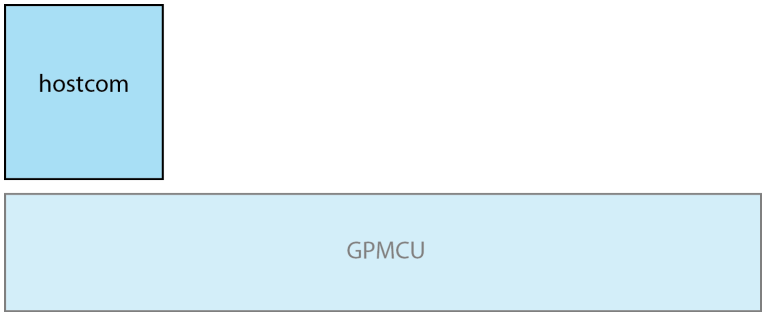
1. **Verificar que se ha configurado la UART correctamente:** si el mensaje se recibe en el PC host sin ninguna alteración, estando el progama de terminal del PC host configurado con los mismos parámetros que el microcontrolador, entonces se tendrá la certeza de que la UART está bien configurada.

2. **Conseguir un mecanismo para enviar datos al PC host:** una vez probado el envío de datos, se podrá reutilizar el código para crear una función que permita enviar un mensaje al PC host y que, por tanto, pueda utilizarse en cualquier parte del programa para abstraer a los demás componentes del modo de uso de la UART.

[8]Inicialmente se empezó a hacer pruebas con la UART 1, pero luego se necesitó usar las dos UARTs y se decidió emplear la UART 2 para la comunicación con el PC host, en lugar de la UART 1. Por ello, en lo sucesivo se utilizará la UART 2.

Tomando el código escrito, se ha creado el módulo `hostcom`[9] del programa *gpcore* con las funciones `hostcom_setup` y `hostcom_send`, para configurar la UART y para realizar el envío de una ristra de caracteres, respectivamente. La Figura 4.1 muestra cómo queda la arquitectura modular por capas de RhinoChip OS tras la creación del módulo `hostcom` como parte del programa *gpcore*, que se ejecutará en el GPMCU.

Figura 4.1: Arquitectura de RhinoChip OS tras la creación del módulo `hostcom`




El siguiente paso en el experimento es probar la recepción de datos. Si bien la recepción se puede implementar también mediante polling, es mucho más práctico implementarla usando interrupciones, puesto que el polling requeriría que el microcontrolador no tuviese otras tareas de las que ocuparse mientras realiza la espera activa y, de no ser así, se correría el riesgo de perder datos, ya que, mientras el microcontrolador se ocupa de las otras tareas, si el buffer de recepción está lleno y llegan nuevos datos, los nuevos datos se descartarían, porque no caben en el buffer de recepción.
Para evitar la pérdida de datos, por tanto, es necesario implementar la recepción mediante interrupciones. Puesto que, en última instancia, este módulo de manejo de la UART será usado para recibir comandos desde el PC host para su posterior interpretación, es necesario recibir todos los bytes del comando hasta la marca de fin de comando antes de poder proceder a su interpretación. Dado que el buffer de recepción de la UART sólo dispone de espacio para 4 bytes, deberá crearse un buffer de mayor tamaño en el que almacenar todos los bytes recibidos hasta que se reciba un comando completo que el intérprete de comandos pueda interpretar.
Por ello, se configurará la UART para que genere una interrupción con cada byte nuevo que reciba, lo cual disparará la rutina de manejo de la interrupción, que se encargará de mover todos los bytes que haya en el buffer de recepción de la UART al buffer de recepción del sistema, mientras los haya. De esta forma se libera el buffer de recepción de la UART para permitir que reciba nuevos bytes de datos, una vez la rutina de interrupción termine.
Consecuentemente, será necesario diseñar una estructura de datos para el buffer de recepción del sistema. Inicialmente se pensó en un buffer circular, pero las pruebas iniciales no tuvieron éxito, por lo que sustituyó por un buffer lineal para poder continuar el desarrollo. Puesto que el buffer es lineal, cada vez que el intérprete de comandos consume un comando del buffer, es necesario desplazar los datos restantes del buffer hacia el principio. El motivo para usar un buffer circular era evitar el *overhead* introducido por la operación de desplazamiento, pero en las pruebas se comprobó que tal operación no actuaba en detrimento del rendimiento ni provocaba la pérdida de datos, puesto que el ritmo al que se reciben los comandos es muy inferior al tiempo que se tarda en realizar el desplazamiento de los datos.

[9] Véase los ficheros `hostcom.h` y `hostcom.c`.

La especificación del compilador Microchip C30 define un nombre estándar para cada rutina de manejo de interrupción del microcontrolador. Las dos UARTs de que disponen ambos microcontroladores, el dsPIC30F4013 y el dsPIC30F4011, son capaces de producir una interrupción cuando se reciben datos —la interrupción de recepción— y una interrupción cuando se puede enviar datos —la interrupción de transmisión—. Según el estándar de Microchip C30, el nombre de la función que maneja la interrupción de recepción para la UART 2 debe ser `_U2RXInterrupt`, y el nombre de la función que maneja la interrupción de transmisión de la UART 2 debe ser `_U2TXInterrupt`[10]. Además, el estándar del compilador especifica que debe utilizarse una macro concreta que indica al compilador que la función será un manejador de interrupciones[11]. Teniendo todo esto en cuenta, la definición de ambas funciones quedaría como se muestra en el Listado 4.3.

Listado 4.3: Definición de los manejadores de interrupción para las interrupciones de la UART 2

```
1  void __attribute__((interrupt, auto_psv)) _U2RXInterrupt(void)
2  {
3      // Desactiva la interrupcion de recepcion de la UART 2 (
           opcional)
4      IFS1bits.U2RXIE = 0;
5
6      // Cuerpo del manejador...
7
8      // Limpia el flag de interrupcion
9      IFS1bits.U2RXIF = 0;
10     // Re-activa la interrupcion de recepcion de la UART 2 (
           opcional)
11     IFS1bits.U2RXIE = 1;
12 }
13
14 void __attribute__((interrupt, auto_psv)) _U2TXInterrupt(void)
15 {
16     // Desactiva la interrupcion de transmision de la UART 2 (
           opcional)
17     IFS1bits.U2TXIE = 0;
18
19     // Cuerpo del manejador...
20
21     // Limpia el flag de interrupcion
22     IFS1bits.U2TXIF = 0;
```

[10] Para la UART 1 la convención es similar; sólo habría que cambiar el `2` en el nombre de la función por un `1`.

[11] Al definir la función como manejador de interrupciones, el compilador se encarga del código de salvaguardado y restauración de registros, y del retorno de la interrupción. De esta manera, el programador no tiene que preocuparse de esos detalles y puede escribir el cuerpo de la función como haría con cualquier otra función, sin instrucciones especiales para el salvaguardado de registros o el retorno del manejador de interrupción. Lo único que sí tendría que hacer es limpiar el flag de interrupción antes de salir de la función. Opcionalmente, también puede desactivar y luego reactivar las interrupciones, para evitar el anidamiento durante la ejecución del manejador.

```
23      // Re-activa la interrupcion de transmision de la UART 2 (
            opcional)
24      IFS1bits.U2TXIE = 1;
25  }
```

Tal como se ha planteado el diseño hasta ahora, sólo definiremos el manejador de interrupciones `_U2RXInterrupt`, puesto que la transmisión se implementa por polling. Además, tal como se ha configurado la UART, la interrupción se producirá cada vez que se reciba un nuevo byte de datos. Por ello, la rutina deberá mover todos los bytes del buffer de recepción de la UART al buffer de recepción del sistema mientras haya datos en el buffer de recepción de la UART y haya espacio libre en el buffer de recepción del sistema.
Para dar soporte a este procedimiento, se ha diseñado un tipo de dato abstracto `buffer_t` (Listado 4.4) que almacena el tamaño máximo del buffer (campo `size`) y el espacio del buffer que está siendo usado (campo `used`).

Listado 4.4: Tipo abstracto de datos `buffer_t` (fichero `datastruc/buffer.h`)

```
11  typedef struct {
12      char    data[BUFFER_SIZE];
13      int     size;
14      int     used;
15  } buffer_t;
```

De esta forma, la extracción de datos del buffer de recepción de la UART se realizaría, según la condición mencionada, mediante el bucle `while` del Listado 4.5, que involucra el bit `URXDA` del registro `U2STA`, que indica el estado del buffer de recepción de la UART (el bit es `1` si el buffer contiene bytes por extraer, y cambia a `0` cuando el buffer está vacío).

Listado 4.5: Manejador de interrupción `_U2RXInterrupt` (fichero `gpcore/hostcom.c`)

```
48  void __attribute__((interrupt, auto_psv)) _U2RXInterrupt(void)
49  {
50      // While UART2 receive buffer has data and the '
            hostcom_rcv_buf' has free
51      // space...
52      while (U2STAbits.URXDA && hostcom_rcv_buf.used <
            hostcom_rcv_buf.size)
53      {
54          // Read the received byte from the UART2 receive
                register
55          hostcom_rcv_buf.data[hostcom_rcv_buf.used] = U2RXREG;
56
57          // If this byte is a command separator and no command
                separator has been
58          // found previously, remember its position in the
                buffer
```

```
59          if (first_cmdend < 0 &&
60              hostcom_rcv_buf.data[hostcom_rcv_buf.used] == *
                   CMDEND)
61          {
62              first_cmdend = hostcom_rcv_buf.used;
63          }
64
65          // Increment the count of elements stored in the
               buffer
66          ++hostcom_rcv_buf.used;
67      }
68
69      // Clear the UART2 receiver interrupt flag
70      IFS1bits.U2RXIF = 0;
71  }
```

Como ya se ha mencionado anteriormente, puesto que los comandos del PC host no se reciben enteros (ya que el buffer de recepción de la UART sólo admite 4 bytes), es necesario mover los datos a un buffer más grande (el buffer de recepción del sistema), de manera que, cuando se reciba un comando entero, el intérprete de comandos pueda analizar el comando completo.
Para facilitar la labor del intérprete de comandos, la propia rutina de servicio de la interrupción (ISR[12]) vista en el Listado 4.5 (pág. 69) detecta cuándo se recibe el carácter que marca el fin de un comando e inicio del siguiente[13], y guarda su posición en la variable `first_cmdend`. De esta manera, `first_cmdend` almacenará siempre el valor de la posición del primer carácter de fin de comando que haya en el buffer, ó $-1$ si no ha llegado todavía ningún carácter de fin de comando. A causa de esto, en el mismo módulo `hostcom` se puede implementar una función `hostcom_read_cmd` que lea del buffer de recepción del sistema todos los bytes de un comando (es decir, todos los bytes hasta llegar al primer carácter de fin de comando), para que, por medio de esta función, el intérprete de comandos pueda obtener un comando completo, y así abstraer al intérprete de comandos de los detalles de la comunicación con el PC host y del manejo del buffer de recepción del sistema.
Esta función (Listado 4.6, pág. 71) deberá, por tanto, recorrer el buffer de recepción del sistema hasta el primer carácter de fin de comando, si lo hay, y, en tal caso, copiar el comando encontrado a un buffer de salida, tras lo cual deberá eliminar los bytes leídos del buffer. Como ya se ha indicado, este procedimiento sería computacionalmente muy simple si se usase un buffer circular, pero se ha optado por realizar una primera implementación con un buffer lineal, puesto que el resto de operaciones (principalmente la escritura en el buffer), son algorítmicamente más sencillas con un buffer lineal que con un buffer circular, a pesar de que la extracción en el buffer lineal sea computacionalmente más costosa (pero algorítmicamente sencilla, también). Esta decisión se ha tomado porque permite conseguir un sistema plenamente funcional en un menor tiempo de desarrollo (debido a que el buffer lineal es más sencillo de implementar) y, de ser necesario o deseable implementar otro tipo de buffer, podría hacerse en una iteración futura

[12] *Interrupt Service Routine*

[13] Según la especificación de comandos del Mark IV, el fin de comando viene indicado por el retorno de carro (*carriage return*, o `\r` en C). Este carácter se representa en el programa mediante la macro `CMDEND`.

del desarrollo del prototipo.
Finalmente, para permitir que el intérprete de comandos pueda comprobar cuándo hay comandos completos en el buffer listos para ser interpretados, se puede proporcionar también una función `hostcom_cmd_available` que devuelva verdadero cuando haya al menos un comando disponible en el buffer.

Listado 4.6: Función `hostcom_read_cmd` (fichero `gpcore/hostcom.c`)

```
73  int hostcom_read_cmd(char buf[], int size, bool_t *full)
74  {
75      int copied;
76      int i, j;
77
78      // Copy received data to the user's buffer
79      for (copied = 0; copied <= first_cmdend && copied < size;
            ++copied)
80          buf[copied] = hostcom_rcv_buf.data[copied];
81
82      // Set the full flag accordingly
83      if (copied > first_cmdend)
84          *full = true;
85      else
86          *full = false;
87
88      // Shift the data in the buffer to remove already copied
            data and search
89      // for the next command end marker
90      if (copied)
91      {
92          // Disable UART2 receive interrupt to prevent the ISR
                from messing
93          // around with 'hostcom_rcv_buf.used'
94          IEC1bits.U2RXIE = 0;
95
96          for (i = 0, j = copied; j < hostcom_rcv_buf.used; ++i,
                ++j)
97              hostcom_rcv_buf.data[i] = hostcom_rcv_buf.data[j];
98          hostcom_rcv_buf.used -= copied;
99
100         for (i = 0;
101              i < hostcom_rcv_buf.used && hostcom_rcv_buf.data[
                    i] != *CMDEND;
102             ++i);
103         if (i < hostcom_rcv_buf.used)
104             first_cmdend = i;
105         else
106             first_cmdend = -1;
```

```
107
108             // Enable UART2 receive interrupt again
109             IEC1bits.U2RXIE = 1;
110         }
111
112         return copied;
113     }
```

Una última prueba a realizar consiste en la implementación de la transmisión mediante interrupciones, ya que inicialmente se realizó mediante polling. Sin embargo, en las pruebas realizadas la transmisión mediante interrupciones no ha funcionado correctamente, por lo que se ha mantenido la transmisión mediante polling.
A diferencia de la recepción, en el caso de la transmisión no es tan grave usar polling. Por supuesto que obliga al microcontrolador a desperdiciar algunos ciclos encerrado en un bucle de espera activa, pero la espera activa no supone mayor inconveniente que el desperdicio de unos pocos ciclos de instrucción (a menos que el microcontrolador deba ejecutar una tarea de control con requisitos estrictos de temporización). Puesto que en la aplicación que nos compete hemos escogido usar dos microcontroladores, y en la división de tareas se ha asignado las tareas críticas de control de motores al MCMCU, es totalmente seguro implementar la transmisión mediante polling.

### 4.1.4 Diseño del intérprete de comandos del PC host

Con la construcción del módulo `hostcom` tal como se ha detallado en la sección anterior, se dispone de una capa de abstracción entre el hardware (manejo de la UART para la recepción de comados del PC host) y el intérprete de comandos que deberá construirse a continuación. El módulo `hostcom` se encarga de la recepción de comandos del PC host y los almacena en un buffer. Sobre el módulo `hostcom` se puede diseñar e implementar ahora el módulo `shell`, que representa el intérprete de comandos del PC host, y que interactuará con el módulo `hostcom` mediante las funciones `hostcom_cmd_available` y `hostcom_read_cmd`, que sirven de interfaz entre `hostcom` y `shell`. Pero, antes de empezar a diseñar el módulo `shell`, es necesario analizar cuidadosamente la sintaxis del lenguaje de comandos del controlador Mark IV.

El lenguaje de comandos del controlador Mark IV consta de comandos de dos letras que pueden proporcionar cero, uno o dos parámetros, separados del nombre de la orden por una coma. El comando debe finalizar con un retorno de carro, para marcar el final del comando y el principio del siguiente comando. Los parámetros pueden ser una letra, un número entero o real en precisión de coma fija (*fixed point*), o, en ocasiones, una cadena de caracteres delimitada por comillas dobles. En el Listado 4.7 se presenta una gramática formal del lenguaje de comandos del controlador Mark IV que detalla de forma más rigurosa esta información.

Listado 4.7: Gramática formal del lenguaje de comandos del controlador Mark IV en EBNF

```
1 <INSTR>       ::= <CMD>[,<PARAM>[,<PARAM>]]\r
```

```
<CMD>          ::= <LETTER><LETTER>
<LETTER>       ::= A .. Z | a .. z
<PARAM>        ::= <INT> | <DEC> | <LETTERPARAM> | <STR>
<INT>          ::= [-]<NUM>
<NUM>          ::= <DIGIT>[<NUM>]
<DIGIT>        ::= 0 .. 9
<DEC>          ::= [-]<NUM>.<DIGIT><DIGIT>
<LETTERPARAM>  ::= A .. H | X .. Z | T
<STR>          ::= "<STRCHARS>"
<STRCHARS>     ::= <SINGLECHAR><STRCHARS> | <SINGLECHAR>
<SINGLECHAR>   ::= <REGULAR> | <SPECIALS>
<REGULAR>      ::= ASCII[33] | ASCII[35] .. ASCII[91] |
                   ASCII[93] .. ASCII[126]
<SPECIALS>     ::= \" | \\
```

Como se puede observar en la gramática, se trata de un lenguaje muy simple y sin ambigüedad que se puede analizar con un analizador sintáctico LL(0). Este tipo de analizador sintáctico se puede implementar muy fácilmente por el método de descenso recursivo [1].

Para implementar este intérprete será necesario construir un analizador léxico y un analizador sintáctico por descenso recursivo. El modo de funcionamiento será el siguiente:

1. El intérprete espera hasta que se reciba un comando completo, lo cual comprueba mediante la función `hostcom_cmd_available`.
2. Cuando se recibe un comando completo, el intérprete extrae el comando del buffer de recepción del sistema mediante la función `hostcom_read_cmd` y lo sitúa en el buffer del intérprete.
3. El intérpete pone en marcha el analizador sintáctico para interpretar el comando recién extraido, para lo cual el analizador sintáctico invoca reiteradamente al analizador léxico, que devuelve los tokens encontrados en el buffer del intérpete, y el analizador sintáctico va reconociendo los tokens hasta que pueda interpretar y ejecutar las acciones del comando, o hasta que detecte un error en el comando.

Del primer y segundo paso se ocupa la función `next_cmd` (Listado 4.8), que es invocada por la función `shell_run_interactive`, encargada de ejecutar una sesión interactiva del intérprete de comandos. Como puede verse en el listado, `next_cmd` espera en un bucle hasta que haya un comando completo en el buffer de recepción, tras lo cual lo extrae del buffer de recepción al buffer del intérprete.

Listado 4.8: Función `next_cmd` (fichero `gpcore/shell.c`)

```
239 void next_cmd(void)
240 {
241     bool_t full;
242     int copied;
```

```
243
244         while (!hostcom_cmd_available());
245         copied = hostcom_read_cmd(cmd_buf, CMD_BUF_SIZE, &full);
246         cmd_buf_pos = 0;
247         param1.present = false;
248         param2.present = false;
249     }
```

En cuanto termina la función `next_cmd` invocada desde `shell_run_interactive`, la siguiente función invocada es `parse_cmd`. La función `parse_cmd` es la función donde está implementado el analizador sintáctico propiamente dicho, y es la que dirige el tercer paso, la interpretación del comando.

Lo primero que hace la función `parse_cmd` es invocar al analizador léxico, implementado en la función `next_token`, para que analice el buffer donde está almacenado el comando en busca del primer token, para así tener el primer token precargado en memoria. Una vez precargado el primer token, invoca la función `instr`, que corresponde al símbolo raíz de la gramática, y, por tanto, se pone en marcha el análisis sintáctico. Este fragmento del código de la función `parse_cmd` se puede ver en el Listado 4.9.

Listado 4.9: Fragmento de la función `parse_cmd` que pone en marcha el análisis sintáctico del comando recibido (fichero `gpcore/shell.c`)

```
254 void parse_cmd(void)
255 {
256     int retval;
257
258     // Fetch the next token and parse it (lexical parser)
259     next_token();
260
261     // The command can be a single instruction...
262     retval = instr();
```

De esta manera, según el algoritmo de descenso recursivo, cada símbolo no terminal de la gramática lleva asociada una subrutina que realiza el análisis del subárbol sintáctico que desciende de dicho símbolo no terminal, y dichas subrutinas se van invocando recursivamente según los símbolos terminales que se vaya encontrando en el buffer del intérprete. Cuando el siguiente token en el buffer no corresponde con ningún símbolo terminal, la subrutina devuelve un error sintáctico, y termina el análisis sin que el comando sea reconocido. En cambio, si el comando finalmente es reconocido por el analizador sintáctico como una instrucción válida del repertorio de instrucciones del controlador, el analizador invocará la función `interpret_cmd`, que, según el comando recibido, ejecutará la acción correspondiente a dicho comando.

La función `interpret_cmd` está implementada por medio de estructuras condicionales (concretamente usando la estructura `switch` de C) y, en función de las dos primeras letras `X` e `Y` que identifican el comando recibido, invoca la rutina `hostcomcmd_xy` correspon-

diente. Así, si, por ejemplo, se recibe el comando `SA`, `interpret_cmd` invocará la rutina `hostcomcmd_sa`[14].
Ni las subrutinas del analizador sintáctico que representan los símbolos no terminales de la gramática ni la función `interpret_cmd` tienen en ningún momento conocimiento del número de parámetros que debe aceptar un comando determinado. Las subrutinas del analizador sintáctico reconocen los parámetros y los almacenan junto con otra información usando la estructura de datos `param_t` (Listado 4.10), que indica si el parámetro está presente, su tipo y su valor[15]. Sin embargo, en ningún momento verifican que el número de parámetros proporcionado sea el adecuado para el comando recibido. Por simplicidad, esta tarea se ha relegado al análisis semántico que debe ser realizado en la propia función `hostcomcmd_xy` correspondiente al comando. Esta función, por tanto, comprueba si los parámetros que necesita el comando (cero, uno o dos parámetros) están presentes (mediante el campo `present` de `param_t`), verifica si tienen el tipo correcto (mediante el campo `type`) y finalmente obtiene su valor (mediante el campo `value`).

Listado 4.10: Estructura de datos `param_t` y variables para almacenar la información de los parámetros del comando analizado (fichero `gpcore/shell.c`)

```
94  typedef struct {
95      bool_t            present; // whether the parameter has
            been specified or not
96      token_type_t      type;    // type of the parameter
97      token_value_t     value;   // value of the parameter
98  } param_t;
99  param_t param1;
100 param_t param2;
```

Con todo esto ya se tiene un analizador léxico y un analizador sintáctico plenamente funcionales. El único componente que falta para tener el intérprete completo es la implementación de las funciones `hostcomcmd_xy` que ejecutan las acciones correspondientes a cada comando. Si bien no todas estas funciones se implementarán en esta fase (ni siquiera todas se implementarán en este proyecto, por lo comentado en la Sección 3.2.6, pág. 26), aún es posible avanzar un poco más la implementación del intérprete si se esboza la implementación de los comandos más simples, fundamentalmente aquellos que sólo sirven para la inspección del sistema (preguntar el estado del sistema, la configuración, etc.). Esto requiere entonces el almacenamiento en memoria del estado y la configuración del sistema, añadiendo por tanto registros adicionales a los ya vistos en la Sección 3.2.5 (pág. 25). Dos de los registros principales son los que corresponden a los comandos `SS` y `SC` del controlador, que devuelven el estado y la configuración del sistema, respectivamente.
Por este motivo, para almacenar en memoria el estado del sistema, se añadirá el módulo `controller_status` en el que se definirá una estructura `controller_status_t` cuyos campos contendrán los registros del estado del sistema, junto con funciones útiles que

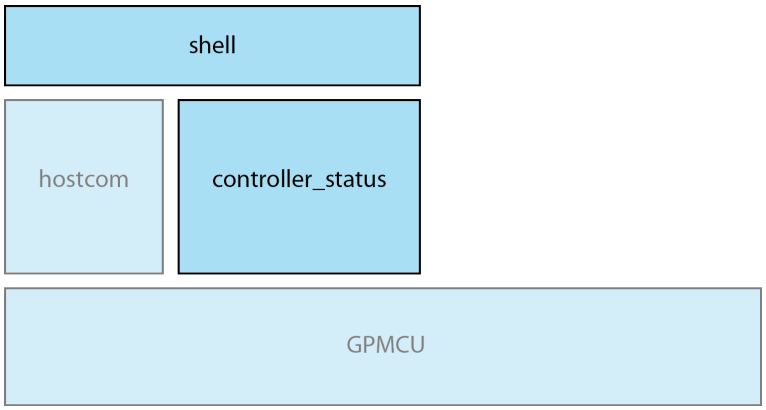
[14]Como requisito de diseño se ha impuesto la no distinción entre mayúsculas y minúsculas en el nombre del comando a la hora de ser interpretado éste, puesto que el controlador Mark IV tampoco hace tal distinción.

[15]La información del primer parámetro se almacena en la variable `param1` y la del segundo parámetro en la variable `param2`, ambas de tipo `param_t`.

permitan la obtención de la información acerca del estado del sistema necesaria para la operación del controlador (p.ej. qué tipo de robot se está usando, si se ha ejecutado un hard home, etc.). Puesto que estas funciones responden a preguntas relativamente sencillas acerca del estado del sistema, pero que se necesitarán conforme se implemente los comandos, no es posible conocer a priori todas las funciones necesarias, por lo que no se puede hacer un diseño a priori de este módulo. Más bien, las funciones se irán añadiendo conforme vayan siendo necesarias (a medida que se implemente los comandos), favoreciendo así el desarrollo ágil del prototipo.

En definitiva, el intérprete de comandos se encontrará implementado en el módulo `shell`, que se basará en el módulo `hostcom` para abstraer el funcionamiento de la comunicación mediante la UART y poder así dedicar los esfuerzos de desarrollo al intérprete propiamente dicho, que se compondrá de un analizador léxico y un analizador sintáctico LL(0) por descenso recursivo.
Además, para almacenar en memoria el estado y la configuración del controlador se plantea la creación del módulo `controller_status`, que proporcionará las estructuras de datos y funciones necesarias para almacenar e inspeccionar el estado del sistema, lo cual será útil para la interpretación de los comandos.
De esta manera, con la adición de los módulos `shell` y `controller_status` al diseño arquitectónico, y teniendo en cuenta que `hostcom` y `controller_status` proporcionan capas de abstracción al módulo `shell`, el diseño por capas de RhinoChip OS después de esta fase de diseño queda como se ilustra en la Figura 4.2.

Figura 4.2: Arquitectura de RhinoChip OS tras la creación del módulo `shell`




### 4.1.5 Módulo de control de motores mediante PWM

Ahora que ya se tiene la funcionalidad fundamental del programa *gpcore* para el dsPIC30F4013 prototipada, el siguiente paso es empezar a construir la funcionalidad de control de motores.

El control de motores consistirá en que, ante una orden de movimiento a una cierta posición de destino, el microcontrolador dsPIC30F4011 deberá mover el motor desde la posición actual (en la que se encuentra el motor al inicio del movimiento) hasta la posición de destino comandada. Además, el control de motores no sólo deberá realizar el movimiento comandado, sino que deberá corregir la posición del motor en caso de que perturbaciones externas lo

desplacen de la posición deseada incluso después de que el movimiento comandado haya concluido, lo cual se realizará mediante un sistema de control en bucle cerrado con realimentación de posición.

Sin embargo, antes de implementar un sistema de control con realimentación de posición o algún algoritmo de movimiento del robot, lo primero que debe hacerse es plantear un sistema para controlar la velocidad y sentido de giro de los motores. Para hacer esto, se usará la técnica de Modulación por Ancho de Pulso (PWM[16]) en conjunto con un circuito de puente en H.

La forma más elemental de controlar la velocidad de un motor de corriente continua (motor DC) es variar el voltaje de alimentación entre $0\,\mathrm{V}$ y el voltaje máximo documentado por el fabricante. De esta manera, a $0\,\mathrm{V}$ el motor estará parado, y al voltaje máximo el motor girará a máxima velocidad. Además, el sentido de giro se puede controlar inviertiendo la polaridad de la tensión de alimentación.

Sin embargo, puesto que implementar este tipo de control sería bastante complejo, lo que se hará es utilizar una señal de PWM que no alimentará directamente al motor entre $0\,\mathrm{V}$ y $24\,\mathrm{V}$ (que es la tensión máxima de los motores de articulación de los robots), sino que se usará con niveles de lógica TTL para controlar la etapa de amplificación de potencia (el circuito de puente en H), es decir, para habilitar y deshabilitar la conducción de los transistores del puente en H. Además, una señal adicional permitirá seleccionar el sentido de giro del motor, la cual controlará los transistores de manera que, según el sentido de giro, se invierta la polaridad de la tensión de alimentación.
Para la realización de los experimentos, se dispone de una Arduino Motor Shield de DFRduino[17] (Figura 4.3, pág. 78), una placa de control de motores compatible con Arduino y basada en el chip L298P, que implementa un puente en H (Figura 4.4, pág. 78) y que permite controlar hasta dos motores DC con una corriente máxima de $2\,\mathrm{A}$. Esto evitará tener que realizar el circuito del puente en H con transistores individuales y el resto de la circuitería necesaria, como los diodos de protección, puesto que ya está todo integrado en la placa. Más adelante, de implementar el sistema final para su uso en producción, sería conveniente diseñar una placa específica con soporte para seis motores, en lugar de utilizar múltiples placas de Arduino.

Para la realización de un módulo `pwm` de RhinoChip OS que implemente el control de velocidad y sentido de giro de los motores, será necesario estudiar las capacidades de los microcontroladores para generar señales de PWM.
Como ya se comentó en la Sección 3.3.2 (pág. 30), el microcontrolador dsPIC30F4011 proporciona un módulo de PWM por hardware con 6 pines de salida. Estos pines de salida están organizados por parejas. Cada pin se puede habilitar individualmente para la salida PWM o para ser usado como pin de E/S digital de propósito general, pero, si los dos pines de una misma pareja son habilitados como salidas de PWM, ambos pines pueden funcionar en uno de dos modos: en modo independiente o en modo complementario. El modo complementario significa que un pin de la pareja producirá la señal del otro pin, invertida. El modo independiente significa que ambos producirán una señal de PWM de forma independiente, pero, puesto que sólo se dispone de un registro de ciclo de trabajo por pareja de pines, ambos pines producirán exactamente la misma señal de PWM. Por este motivo, sólo se podría llegar a controlar 3 motores

[16] *Pulse Width Modulation*
[17] `http://is.gd/pQDR3Z`

Figura 4.3: Arduino Motor Shield de DFRduino

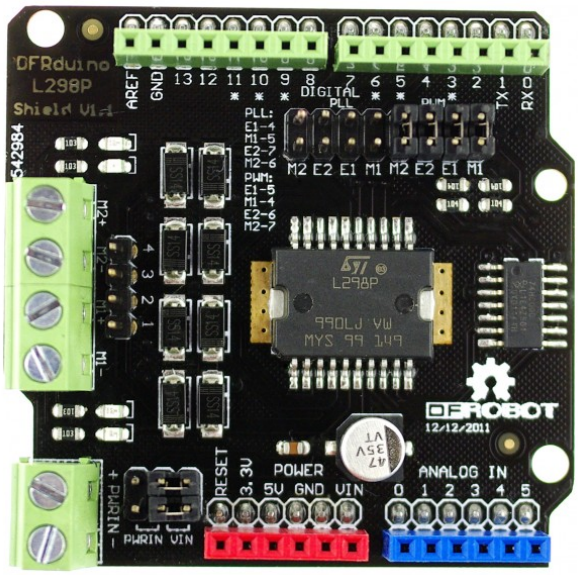
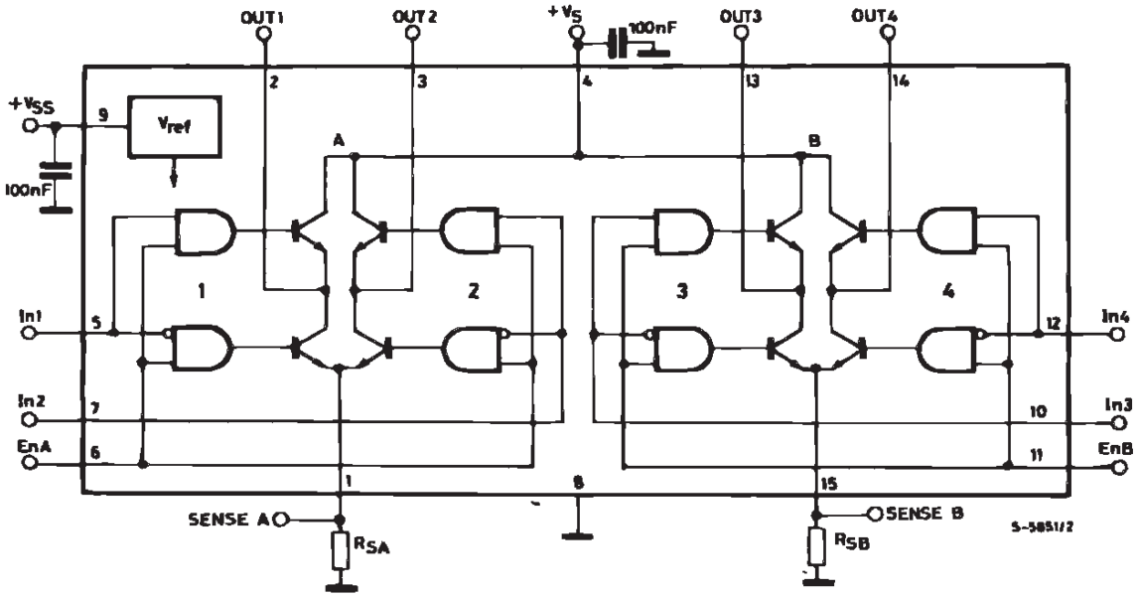



Figura 4.4: Puente en H implementado en el chip L298



con PWM por hardware.
Puesto que nuestra aplicación requiere el control de, al menos, 6 motores, plantearemos como solución que 3 de estos pines se usen para generar una señal de PWM por hardware, y habilitar los otros 3 pines como pines de salida digital, para así generar una señal de PWM por software[18].

Debido a que sólo tres de los seis pines del módulo PWM del MCMCU se usarán en modo PWM, mientras que los otros 3 se usarán como pines de salida digital, y puesto que se necesitarán otros 6 pines de salida digital para la selección del sentido de giro, la distribución de pines elegida para el módulo `pwm` queda como se muestra en la Tabla 4.2.

Tabla 4.2: Asignación de pines del módulo PWM

| Pin | Nombre de la señal | Tipo | Descripción |
|---|---|---|---|
| RE0 | PWM1 | Salida de PWM | Señal de PWM del canal 1 (motor A) |
| RE1 | PWM2 | Salida digital | Señal de PWM del canal 2 (motor B) |
| RE2 | PWM3 | Salida de PWM | Señal de PWM del canal 3 (motor C) |
| RE3 | PWM4 | Salida digital | Señal de PWM del canal 4 (motor D) |
| RE4 | PWM5 | Salida de PWM | Señal de PWM del canal 5 (motor E) |
| RE5 | PWM6 | Salida digital | Señal de PWM del canal 6 (motor F) |
| RB6 | DIR1 | Salida digital | Selección de sentido de giro del canal 1 (motor A) |
| RB7 | DIR2 | Salida digital | Selección de sentido de giro del canal 2 (motor B) |
| RB8 | DIR3 | Salida digital | Selección de sentido de giro del canal 3 (motor C) |
| RC13 | DIR4 | Salida digital | Selección de sentido de giro del canal 4 (motor D) |
| RC14 | DIR5 | Salida digital | Selección de sentido de giro del canal 5 (motor E) |
| RE8 | DIR6 | Salida digital | Selección de sentido de giro del canal 6 (motor F) |

Para poder usar tres pines del módulo PWM del MCMCU como pines de salida de PWM y los otros tres como pines de salida digital, es necesario

1. configurar los canales de PWM 1 a 3 en modo de salida independiente (mediante los bits `PMODi` del registro `PWMCON1`, donde `i` indica el número del canal de PWM)

```
1  PWMCON1bits.PMOD1 = 1;
2  PWMCON1bits.PMOD2 = 1;
3  PWMCON1bits.PMOD3 = 1;
```

2. habilitar los pines de salida de PWM `RE0`, `RE2` y `RE4` como tales (mediante los bits `PENiL` del registro `PWMCON1`)

[18]Más adelante en el desarrollo del proyecto, cuando tuvimos que sustituir la Arduino Motor Shield por la placa del Apéndice A (pág. 127), fue necesario descartar el PWM por hardware y realizar todas las salidas de PWM por software, para lo cual bastó con replicar el código de PWM por software (explicado más adelante en esta sección) para los pines que antes generaban PWM por hardware.

```
1  PWMCON1bits.PEN1L = 1;
2  PWMCON1bits.PEN2L = 1;
3  PWMCON1bits.PEN3L = 1;
```

3. habilitar los pines de salida digital `RE1`, `RE3` y `RE5` como tales (mediante los bits `PENiH` del registro `PWMCON1`)[19]

```
1  PWMCON1bits.PEN1H = 0;
2  PWMCON1bits.PEN2H = 0;
3  PWMCON1bits.PEN3H = 0;
```

El siguiente paso será entonces configurar el timer del módulo PWM del MCMCU para que genere una señal de PWM con una cierta frecuencia. En base a la experiencia, para que un motor DC se mueva, es necesario proporcionarle un voltaje durante alrededor de $18\,\text{ms}$ ó $20\,\text{ms}$. Nosotros elegiremos unos $20\,\text{ms}$, por lo que el periodo de nuestra señal de PWM será $T_{\text{PWM}} = 20\,\text{ms}$, de manera que la frecuencia será $f_{\text{PWM}} = \frac{1}{T_{\text{PWM}}} = 50\,\text{Hz}$.
El módulo PWM utiliza 15 bits del registro `PTPER`[20] para almacenar el valor máximo al que debe contar para generar una señal de PWM de frecuencia $f_{\text{PWM}}$. Este valor se calcula como

$$\text{PTPER} = \left\lfloor \frac{f_{\text{CY}}}{f_{\text{PWM}} \times \text{PWMPRESCALE} - 1} \right\rfloor, \tag{4.3}$$

donde PWMPRESCALE es el valor de pre-escalado del timer del módulo PWM. Éste se utiliza para escalar el periodo del timer cuando dicho periodo es demasiado grande para caber en el registro de configuración. De esta manera, se consigue que el timer permita un rango de frecuencias mayor que el que se tendría si no hubiese pre-escalado.
A su vez, el módulo PWM utiliza los 16 bits de los registros `PDC1`, `PDC2` y `PDC3` para almacenar el valor hasta el que debe contar para generar la parte activa de la señal de PWM de los canales de PWM 1, 2 y 3, respectivamente. El valor de estos registros dependerá, por tanto, del ciclo de trabajo deseado para la señal de cada canal. Si el valor del registro es 0xFFFF (valor máximo), el ciclo de trabajo de la señal generada será el 100 %. La fórmula para el cálculo del valor del registro `PDCi` del canal `i` que permita generar una señal de PWM con ciclo de trabajo $d$, donde $d \in [0,\ 1]$, es

$$\text{PDC} = d \cdot 2 \cdot \text{PTPER}. \tag{4.4}$$

Puesto que la frecuencia de la señal de PWM deseada es $f_{\text{PWM}} = 50\,\text{Hz}$ y la frecuencia de ciclo es $f_{\text{CY}} = 29.50\,\text{MHz}$, por medio de la Ecuación 4.3 obtenemos que, con un valor de pre-escalado igual a 1, el registro `PTPER` debería configurarse con el valor

$$\text{PTPER} = \left\lfloor \frac{29.50\,\text{MHz}}{50\,\text{Hz} \times 1 - 1} \right\rfloor = 602040. \tag{4.5}$$

[19] Esta configuración se hace en los registros del módulo de PWM, pero también es necesario configurar el registro `TRISE` para indicar si los pines serán de entrada o de salida. Puesto que en este caso deseamos usar los pines como salidas digitales, que es la configuración por defecto, no hará falta modificar el registro `TRISE` correspondiente.

[20] El registro es de 16 bits, pero el bit más significativo no se usa (su valor es siempre 0).

Sin embargo, el número de bits necesarios para representar este valor es

$$\lceil \log_2 \text{PTPER} \rceil = \lceil \log_2 602040 \rceil = 20\,\text{bits}. \tag{4.6}$$

Por tanto, deberemos utilizar un valor de pre-escalado[21] mayor que 1. En este caso, usaremos un valor de pre-escalado igual a 64 para poder conseguir que el valor calculado mediante la Ecuación 4.3 (pág. 80) quepa en el registro `PTPER`. Con este valor de pre-escalado, obtenemos que `PTPER` debe ser

$$\text{PTPER} = \left\lfloor \frac{29.50\,\text{MHz}}{50\,\text{Hz} \times 64 - 1} \right\rfloor = 9221. \tag{4.7}$$

Este número requiere

$$\lceil \log_2 9221 \rceil = 14\,\text{bits} \tag{4.8}$$

para ser representado, por lo que ahora sí cabe en el registro `PTPER`.
Con todo esto, la configuración del timer del módulo PWM se realizaría como se muestra en el Listado 4.11.

Listado 4.11: Instrucciones para la configuración del timer del módulo PWM

```
1  #define FOSC    7372800 // On-board crystal freq: 7.3728 MHz
2  #define PLLMODE 16      // On-chip PLL setting
3  #define FCY     (FOSC * PLLMODE / 4)
4
5  #define TPWM        0.020      // PWM period: 20 ms
6  #define FPWM        (1 / TPWM) // PWM frequency: 50 Hz
7  #define PWMPRESCALE 64         // PWM prescale value of 1:64
8  #define PWMPER      (FCY / (FPWM * PWMPRESCALE) - 1)
9
10 // Set prescale of 1:64 (0=1:1, 1=1:4, 2=1:16, 3=1:64)
11 PTCONbits.PTCKPS = 3;
12 // Set PWM period
13 PTPER = PWMPER;
14 // Enable the hardware PWM module
15 PTCONbits.PTEN = 1;
```

Para implementar PWM por software el enfoque será similar, pero implementando un contador por software mediante un timer de propósito general. El periodo y la frecuencia de la señal de PWM serán los mismos, es decir, $T_{\text{PWM}} = 20\,\text{ms}$ y $f_{\text{PWM}} = 50\,\text{Hz}$. Sin embargo, para el cálculo de la frecuencia del timer debemos considerar qué resolución de PWM deseamos que tenga el PWM por software, esto es, cada cuánto se muestreará la señal de PWM que se desea generar.
Para ello considérese el modo de funcionamiento del PWM por software: Un timer genera una interrupción con una cierta frecuencia $f$ y la ISR del timer incrementa un contador (`pwmcount`) y activa un pin de salida (a través del cual se obtiene la señal de PWM). Cuando el contador supera un valor determinado (`pwmduty`), se sabe que se ha completado el ciclo de

[21] Los valores de pre-escalado soportados por el timer de PWM son 1, 4, 16 y 64.

trabajo, por lo que la ISR desactiva el pin de salida y continúa contando hasta que el contador alcanza el valor `PWMPER`, que marca la compleción de un periodo de la señal de PWM. En ese momento se reinicia la cuenta y se vuelve a generar el tramo activo de la señal de PWM.
En base a este modo de funcionamiento, la frecuencia con la que se generan muestras de la señal de PWM, es decir, la frecuencia de la interrupción del timer ($f$), marca la resolución que se puede alcanzar en el ciclo de trabajo. Por tanto —suponiendo que el tiempo de ejecución de la ISR `_T1Interrupt` es despreciable—, la resolución en la duración del ciclo de trabajo será igual a $T$, donde $T = \frac{1}{f}$. De esta forma, el módulo de PWM por software será capaz de generar con precisión aquellas señales de PWM cuyo ciclo de trabajo sea múltiplo de $T$. Si el ciclo de trabajo especificado no es múltiplo de $T$, habrá un cierto error de muestreo en la generación de la señal.
Por tanto, para poder especificar el ciclo de trabajo $D$ como un porcentaje entero de 0 a 100, el periodo del timer debe ser

$$T = \frac{T_{\text{PWM}}}{100}. \tag{4.9}$$

Puesto que $T_{\text{PWM}} = 20\,\text{ms}$, tenemos entonces que

$$T = \frac{T_{\text{PWM}}}{100} = \frac{20\,\text{ms}}{100} = 0.2\,\text{ms}. \tag{4.10}$$

De manera similar a como se hizo para el timer del módulo PWM del MCMCU, para el timer 1 —que es el que hemos elegido para implementar PWM por software— será necesario configurar el registro de periodo del timer, que en este caso se llama `PR1`. El valor que debe contener dicho registro para que el timer genere una interrupción con frecuencia $f = \frac{1}{T}$ viene dado por

$$\text{PR1} = \left\lfloor \frac{f_{\text{CY}}}{f \times \text{T1PRESCALE}} \right\rfloor = \left\lfloor \frac{T \cdot f_{\text{CY}}}{\text{T1PRESCALE}} \right\rfloor. \tag{4.11}$$

En este caso, con un valor de pre-escalado igual a 1, el valor calculado mediante la Ecuación 4.11 ya cabe en el registro `PR1`.
Aparte de esto, habrá que crear unas variables `pwmcount` y `pwmduty` para simular los registros `PTMR` y `PDCi` del módulo de PWM por hardware. Puesto que se deberá dar soporte a 3 canales, se creará unas estructuras como las del Listado 4.12.

Listado 4.12: Estructuras de datos para la generación de señales PWM por software

```
1  /**
2   * PWM count register.
3   */
4  struct {
5      unsigned int channel4;
6      unsigned int channel5;
7      unsigned int channel6;
8  } pwmcount;
9
10 /**
11  * PWM duty cycle register.
12  */
```

```
13  struct {
14      unsigned int channel4;
15      unsigned int channel5;
16      unsigned int channel6;
17  } pwmduty;
```

De esta forma, la inicialización del PWM por software requerirá la puesta a cero de los campos de los registros `pwmcount` y `pwmduty`, así como (1) el borrado del flag de interrupción del timer 1, (2) la habilitación de las interrupciones del timer 1, (3) la configuración del periodo y el valor de pre-escalado del timer 1, y (4) la habilitación del módulo hardware del timer 1. Las instrucciones para la configuración del timer 1 se ilustran en el Listado 4.13.

Para realizar toda esta configuración y para permitir que el resto de módulos puedan modificar el ciclo de trabajo de alguno de los canales de PWM, el módulo `pwm` proporcionará las funciones cuya declaración prototipo se muestra en en el Listado 4.14.
La función `pwm_setup` se utilizará al inicio del sistema para realizar toda la configuración detallada anteriormente. Por su parte, el resto de funciones modificarán los registros del ciclo de trabajo de cada uno de los canales de PWM (hardware o software) de acuerdo con el parámetro proporcionado, que puede ser un valor entero entre 0 y 100.

Listado 4.13: Instrucciones para la configuración del timer 1 para la generación de señales PWM por software

```
1  #define T1PRESCALE     1
2  #define PWMRESOL       100
3  #define T1PERIOD       (TPWM / PWMRESOL)
4  #define PR1VAL         ((T1PERIOD * FCY) / T1PRESCALE)
5  // Clear the Timer 1 interrupt flag
6  IFS0bits.T1IF = 0;
7  // Enable Timer 1 interrupts
8  IEC0bits.T1IE = 1;
9  // Set Timer 1 prescaler (0=1:1, 1=1:8, 2=1:64, 3=1:256)
10 T1CONbits.TCKPS = 0;
11 // Set Timer 1 period
12 PR1 = PR1VAL;
13 // Start Timer 1
14 T1CONbits.TON = 1;
```

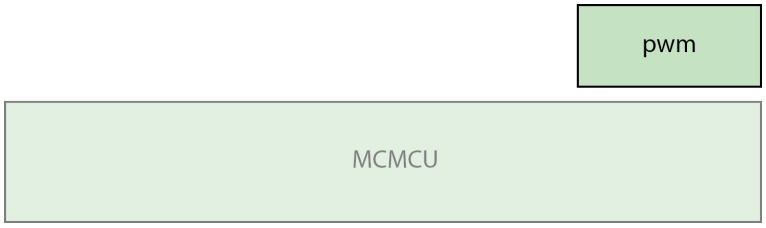
Listado 4.14: Funciones del módulo `pwm` (fichero `motorctl/pwm.h`)

```
45 inline void pwm_setup(void);
46 inline void pwm_set_duty1(int duty);
47 inline void pwm_set_duty2(int duty);
48 inline void pwm_set_duty3(int duty);
49 inline void pwm_set_duty4(int duty);
50 inline void pwm_set_duty5(int duty);
```

```
51  inline void pwm_set_duty6(int duty);
```

Tras la implementación del módulo `pwm`, la arquitectura por capas del programa `motorctl` de RhinoChip OS queda como se muestra en la Figura 4.5.

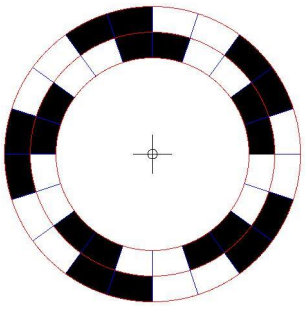
Figura 4.5: Arquitectura de RhinoChip OS tras la creación del módulo `pwm`



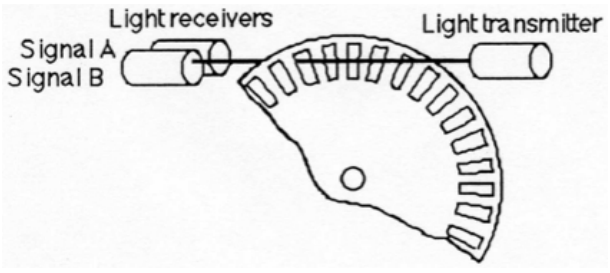

### 4.1.6 Interfaz para los codificadores de posición de los motores

Ahora que ya se tiene un mecanismo para poner en marcha y parar los motores, controlando su velocidad y sentido de giro, podemos prestar atención al otro aspecto fundamental para poder realizar el control de motores: es necesario conocer la posición en que el motor se encuentra en cada momento. Para ello, cada motor dispone de un codificador de posición incremental.
Este tipo de sensores de posición están formados por un disco con dos o más pistas divididas en sectores de colores alternos, usualmente negro y blanco (o algún material reflectante en lugar del blanco), que absorben o reflejan la luz, de manera que mediante un LED y un fotodiodo se puede detectar cuál de los dos colores se está midiendo en cada pista[22] (Figura 4.6(a)), lo cual producirá, por cada pista, una señal cuadrada que alterna entre 0 y 1 según si el sector medido absorbe o refleja la luz. Mediante una de estas señales cuadradas ya se puede medir la

Figura 4.6: Esquemas del diseño de varios codificadores de posición incrementales



(a) Codificador incremental de dos pistas



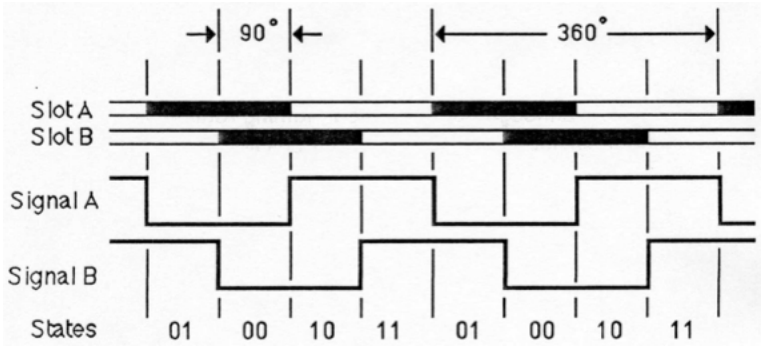

(b) Codificador incremental con ranuras

posición del motor relativa a una posición inicial conocida a priori. Sin embargo, una sola señal no permite conocer el sentido de giro del motor. Para conocer el sentido de giro, se necesita

[22] Algunos codificadores incrementales están realizados mediante discos con ranuras que permiten el paso de la luz de un LED, pero los robots Rhino disponibles en el laboratorio usan codificadores con sectores de colores.

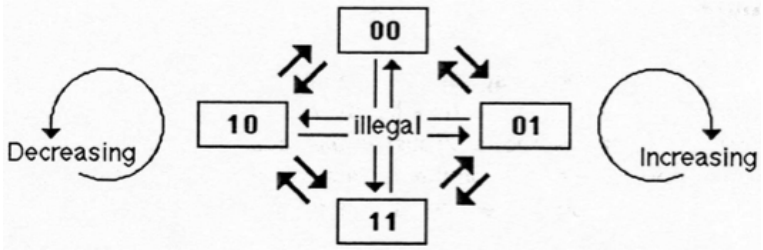
que el codificador tenga dos pistas y que estas pistas estén desplazadas entre sí de manera que las señales resultantes estén desfasadas $90^o$ eléctricos. De esta forma, con dos pistas en el codificador se obtendrían dos señales cuadradas $A$ y $B$ desfasadas $90^o$ (Figura 4.7) y que permiten medir la posición y el sentido de giro del motor.

Figura 4.7: Señales $A$ y $B$ de un codificador de posición incremental




Para conocer el sentido de giro del motor debe considerarse el valor binario de ambas señales en cada instante, lo cual da como resultado cuatro estados posibles según el valor de las señales $A$ y $B$ en cada momento: 00, 01, 10 y 11. Entonces, el sentido de giro se puede calcular observando las transiciones de estado del codificador, tal como se muestra en la Figura 4.8.

Figura 4.8: Transiciones de estado del codificador de posición y obtención del sentido de giro




El caso de los robots Rhino es un poco particular, puesto que los codificadores de posición de estos robots no tienen dos pistas, sino que están formados por una única pista dividida en sectores de colores alternos, como se muestra en la Figura 4.9 (pág. 86). En este caso, en lugar de tener dos pistas y dos sensores de reflexión en la misma posición, lo que se ha hecho es separar los sensores de reflexión de manera que las señales estén igualmente desfasadas (un enfoque similar al usado en el codificador de ranuras de la Figura 4.6(b), pág. 84), por lo que todo lo explicado sigue siendo válido.

Por tanto, el siguiente paso para el diseño e implementación del programa *motorctl* para el control de motores es diseñar el módulo software que se va a encargar de leer los codificadores y actualizar la posición de los motores, para así conocer en todo momento en qué posición se encuentran los motores, ya sea que estén parados o estén en mitad de un movimiento.
El módulo software a diseñar en este punto —que llamaremos `qei`— puede realizarse tanto por hardware como por software.
En la opción de usar hardware para realizar este módulo existen dos vertientes: (1) usar hardware específico diseñado por nosotros, o (2) usar hardware proporcionado por el microcontrolador. Esta segunda vertiente consiste en utilizar un módulo hardware especial del microcontrolador,

Figura 4.9: Codificador de posición incremental de una sola pista

que suele estar presente en los microcontroladores de control de motores, como el *módulo QEI* del dsPIC30F4011, y que hace de interfaz entre el microcontrolador y los codificadores. La otra vertiente sería diseñar nuestro propio módulo QEI por hardware usando para ello los circuitos integrados necesarios según el circuito que diseñemos[23]. La ventaja de esta opción es que aceleraría el procesamiento y disminuiría la carga de cómputo del microcontrolador, puesto que hay un hardware dedicado que funciona en paralelo a la CPU. Por contra, la desventaja sería que esta implementación requeriría mucho hardware —en caso de realizar un diseño propio— y esto exige espacio en la placa de circuito impreso e incrementaría los costes del sistema, además de que se dificulta la actualización del sistema y la corrección de errores.
La opción alternativa es realizar el módulo `qei` totalmente por software, lo cual se haría muestreando las dos señales $A$ y $B$ con la frecuencia suficiente como para no perder datos y garantizar que se detectan los cambios de estado cuando ocurren. Esto aumentaría la carga de trabajo de la CPU y requeriría un número considerable de pines de entrada (2 pines por cada codificador), pero también presentaría las ventajas de que reduciría los costes del sistema (debido a que se elimina la necesidad de hardware adicional) y que facilitaría la tarea de actualización y corrección de errores en el módulo QEI (puesto que una actualización de software es más sencilla y barata de realizar que una actualización de hardware).

A causa del abaratamiento de costes que supone una implementación por software, y debido a que el dsPIC30F4011 sólo dispone de un módulo QEI por hardware con soporte para un único codificador, se ha optado por realizar el módulo QEI enteramente por software, para así abaratar costes. Además, puesto que la frecuencia de ciclo es elevada (pudiendo llegar a los 30 MHz), el *overhead* causado por la implementación software del módulo QEI será prácticamente nulo, por lo que no debería tenir ningún impacto negativo en el rendimiento del sistema ni en la temporización (no deberían producirse retrasos ni pérdida de datos).

Para implementar el módulo `qei` se plantea, por tanto, realizar un muestreo periódico de las señales $A$ y $B$ de cada codificador. Para ello se necesitarán dos pines de entrada digital[24]

[23] Esta área está bastante estudiada y hay multitud de circuitos diseñados para implementar una interfaz con codificadores incrementales. Véase por ejemplo [2, 3, 4].

[24] Las señales a muestrear son señales digitales con niveles TTL, por lo que no es necesario utilizar un conversor analógico/digital. Por tanto, basta con utilizar pines de entrada digital.

por cada codificador[25] y el uso de un timer del microcontrolador (se usará el timer 2).
Para poder realizar el muestreo mediante un timer es necesario primero conocer la frecuencia máxima de las señales $A$ y $B$, puesto que, si se muestrea a una frecuencia inferior, se producirá la pérdida de datos. Esta frecuencia se puede determinar fácilmente mediante un osciloscopio alimentando el motor a $24\,\mathrm{V}$, que es el voltaje de alimentación máximo soportado por el motor, y con el que el motor girará a máxima velocidad, por lo que la frecuencia de las señales $A$ y $B$ será máxima. Haciendo este experimento, se ha comprobado que el periodo mínimo de las señales $A$ y $B$ es de alrededor de $0.5\,\mathrm{ms}$, lo cual equivale a una frecuencia de unos $2\,\mathrm{kHz}$.
Dado que las señales $A$ y $B$ están desfasadas $90^{\mathrm{o}}$ una respecto a la otra, en un ciclo (la duración de un periodo) se producen 3 cambios de estado, por lo que se obtienen los 4 estados del codificador. Dado que el módulo QEI debe contar los cambios de estado, la frecuencia de muestreo debe ser suficientemente alta como para que el QEI no se pierda ningún cambio de estado. Esto significa que la frecuencia de muestreo debe ser

$$f_{\text{sample}} \geq 4\, f_{\text{signal}}, \tag{4.12}$$

donde $f_{\text{signal}}$ es la frecuencia de las señales $A$ y $B$. Puestro que, en nuestro caso, $f_{\text{signal}} = 2\,\mathrm{kHz}$, obtenemos que $f_{\text{sample}} \geq 8\,\mathrm{kHz}$. Esto se comprueba viendo que el número de muestras tomadas es

$$N = \frac{T_{\text{signal}}}{T_{\text{sample}}} = \frac{f_{\text{sample}}}{f_{\text{signal}}}, \tag{4.13}$$

y que, debido a que en un ciclo de una de las señales $A$ o $B$ se producen los 4 estados del codificador, el número de muestras por estado será

$$n = \frac{N}{4} = \frac{T_{\text{signal}}}{4\,T_{\text{sample}}} = \frac{f_{\text{sample}}}{4\,f_{\text{signal}}}. \tag{4.14}$$

Utilizando la frecuencia de muestreo mínima $f_{\text{sample}} = 8\,\mathrm{kHz}$, la Ecuación 4.14 queda como

$$n = \frac{f_{\text{sample}}}{4\,f_{\text{signal}}} = \frac{8\,\mathrm{kHz}}{4 \times 2\,\mathrm{kHz}} = 1, \tag{4.15}$$

que es el mínimo número de muestras por estado necesarias para que no haya pérdida de información.
Por tanto, para mayor seguridad, elegiremos $f_{\text{sample}} = 10\,\mathrm{kHz}$, lo cual producirá $\frac{10\,\mathrm{kHz}}{4 \times 2\,\mathrm{kHz}} = 1.25$ muestras/estado[26].

En definitiva, se deberá configurar el timer 2 con una frecuencia de $10\,\mathrm{kHz}$, la cual se puede conseguir a partir de la frecuencia de ciclo $f_{\text{CY}}$ con un valor de pre-escalado igual a 1.
Además, para dar soporte al autómata de la Figura 4.8 (pág. 85) y poder detectar las transiciones de estado, será necesario guardar en memoria el estado actual y el estado anterior, y ambas variables deberán ser inicializadas en el arranque del sistema, para que el módulo QEI no detecte cambios de estado cuando el sistema se inicia (y los motores todavía no se han movido). Para que esto no suceda, se leerá el estado actual de los pines de entrada y se asignará dicho

[25] Dado que un requisito del sistema es poder controlar al menos 6 motores, se necesitarán como mínimo 12 pines de entrada.

[26] En los experimentos realizados, esta frecuencia de muestreo es más que suficiente, pero, si se quisiese aun más robustez, podría elegirse una frecuencia de muestreo del doble que la mínima, lo cual produciría 2 muestras/estado.

estado tanto en la variable que almacena el estado actual como en la variable que almacena el estado anterior. De esa manera, no se detectará ninguna transición en el inicio del sistema y, por tanto, no se contarán pasos del motor cuando todavía no los ha habido.
Finalmente, hay que recordar que, antes de todo esto, también hay que configurar los pines de entrada. Puesto que harán falta 12 pines para dar soporte a 6 motores, se optará por usar los pines `RB0`–`RB5` del puerto B, los pines `RD2`–`RD3` del puerto D y los pines `RF0`, `RF1`, `RF4` y `RF5` del puerto F, asignados tal como se muestra en la Tabla 4.3. Debido a que los pines del

Tabla 4.3: Asignación de pines del módulo QEI

| Pin | Nombre de la señal | Tipo | Descripción |
|---|---|---|---|
| `RB0` | `QEA_MA` | Entrada digital | Señal $A$ del motor A |
| `RB1` | `QEB_MA` | Entrada digital | Señal $B$ del motor A |
| `RB2` | `QEA_MB` | Entrada digital | Señal $A$ del motor B |
| `RB3` | `QEB_MB` | Entrada digital | Señal $B$ del motor B |
| `RB4` | `QEA_MC` | Entrada digital | Señal $A$ del motor C |
| `RB5` | `QEB_MC` | Entrada digital | Señal $B$ del motor C |
| `RD2` | `QEA_MD` | Entrada digital | Señal $A$ del motor D |
| `RD3` | `QEB_MD` | Entrada digital | Señal $B$ del motor D |
| `RF0` | `QEA_ME` | Entrada digital | Señal $A$ del motor E |
| `RF1` | `QEB_ME` | Entrada digital | Señal $B$ del motor E |
| `RF4` | `QEA_MF` | Entrada digital | Señal $A$ del motor F |
| `RF5` | `QEB_MF` | Entrada digital | Señal $B$ del motor F |

puerto B están multiplexados con el conversor analógico/digital del microcontrolador, habrá que habilitar la entrada digital (en lugar de la analógica) mediante el registro `ADPCFG`.
Por todo esto, el módulo `qei` contendrá las variables de tipo *array* `prev_encoder_state` y `curr_encoder_state`, encargadas de almacenar el estado anterior y el estado actual del codificador de cada motor, y la función `qei_setup`, encargada de (1) configurar los pines de entrada, (2) inicializar las variables `prev_encoder_state` y `curr_encoder_state` tal como se ha explicado, y (3) configurar el timer 2 para que genere una interrupción a una frecuencia de 10 kHz.

Para concluir el diseño del módulo QEI, el muestreo de las señales y la detección de las transiciones de estado se realizará en la rutina de interrupción del timer 2, `_T2Interrupt`. Esta rutina deberá, para cada motor, realizar las siguientes acciones:

1. Almacenar el "estado actual" del codificador en la variable "estado anterior", puesto que la variable que almacena el estado actual contiene el valor correspondiente al estado actual en la anterior invocación de la ISR.

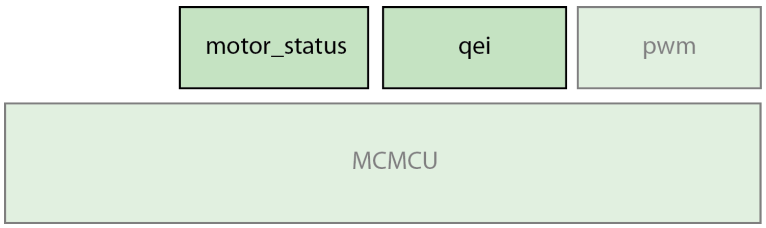
2. Tomar una muestra de las señales $A$ y $B$ y almacenar el estado actual de dichas señales en la variable correspondiente.

3. Utilizar los nuevos valores de las variables "estado actual" y "estado anterior" para detectar si ha habido una transición de estado y, si la ha habido, incrementar el contador de

pasos del motor (una variable que cuenta los pasos del motor, es decir, los cambios de estado del codificador). Este incremento será positivo o negativo (decremento), dependiendo del sentido de giro del motor.

Esto requiere, por tanto, la definición de la ISR `_T2Interrupt`[27] y, además, de una variable que almacene el número de pasos de cada motor. Puesto que esta variable —que llamaremos `motor_steps`— almacena información acerca del estado de los motores, crearemos un módulo `motor_status` en el que definir esta y otras variables que se necesiten en el futuro para representar el estado de los motores.
Por tanto, después de la creación de los módulos `qei` y `motor_status`, la arquitectura por capas del programa *motorctl* de RhinoChip OS queda como se ilustra en la Figura 4.10.

Figura 4.10: Arquitectura de RhinoChip OS tras la creación del módulo `qei`




En este momento, el módulo QEI ya está diseñado e implementado, ofreciendo soporte para 6 motores. Sin embargo, en las pruebas se ha visto que no realizaba la cuenta de los pasos del motor correctamente (el valor del contador no era el correcto).
Entonces comenzamos a buscar una solución por hardware, pero procurando evitar que fuera necesario utilizar mucho hardware (por los motivos expuestos anteriormente). Se intentó, por tanto, utilizar los circuitos propuestos en [2, 3, 4] para detectar las transiciones en las señales $A$ y $B$ de cada codificador, pero sin incluir más hardware del necesario[28].
Al ver que el funcionamiento de las diferentes soluciones hardware probadas no llegaba a ser satisfactorio y que éstas requerían el uso de mucho hardware y muchas conexiones en la placa de prototipado, volvimos a iterar sobre el diseño software, iniciando un proceso de diagnóstico del problema, momento en el cual nos dimos cuenta de que podía ser que las señales $A$ y $B$ de los codificadores presentaran un ruido notable que estuviese causando que la cuenta de pasos no correspondiese a los pasos que realmente había dado el motor.
Para solucionar esto, pensamos en usar un buffer *Schmitt trigger* para filtrar las señales de los codificadores antes de introducirlas a los puertos de entrada del microcontrolador. Estudiando la placa de circuito impreso del controlador Mark IV, descubrimos que precisamente se utilizaba un buffer *Schmitt trigger* —concretamente el chip 74244— situado entre las entradas de los motores y los pines de entrada del microcontrolador 8751. Por tanto, tras diagnosticar el problema y utilizar un chip 74LS244 para estabilizar las señales de los codificadores, el problema fue resuelto y la solución QEI por software comenzó a funcionar correctamente.

[27] La definición puede verse en el fichero `motorctl/qei.c` dentro del directorio de código fuente de RhinoChip OS.

[28] Por ejemplo, sin utilizar contadores binarios, ya que la cuenta se puede implementar por software en el microcontrolador, si la circuitería de detección de transiciones genera una interrupción que sea manejada por el MCMCU.

### 4.1.7 Interfaz para la intercomunicación entre GPMCU y MCMCU

Puesto que las tareas de cómputo general —como es la interpretación de comandos— y las tareas de control de motores se realizan en microcontroladores diferentes, es necesario habilitar una forma de intercomunicación entre los dos microcontroladores, GPMCU y MCMCU, para que puedan sincronizarse y cooperar. Sin esta intercomunicación, el GPMCU no tendría forma de comunicar al MCMCU los movimientos (y otras acciones) que debe ejecutar, de acuerdo con los comandos enviados por el PC host al GPMCU.

En una primera fase de análisis, planteamos la posibilidad de utilizar el protocolo I2C o el protocolo SPI para tal efecto. Por su simplicidad y por ser un protocolo bien conocido, nos decantamos por I2C. Además, en una primera tanda de experimentos del diseño de este módulo de intercomunicación —que llamaremos `mcuicom`[29]— planteamos un protocolo de comunicación basado en comandos binarios, para así reducir el *overhead* en la comunicación debido al envío de caracteres ASCII, que requerirían la transmisión de más bits de los necesarios.
Sin embargo, los experimentos con el módulo I2C de los microcontroladores fueron infructuosos. La comunicación no se realizaba correctamente y los microcontroladores entraban en un estado de bloqueo mutuo. Por ello tomamos dos nuevas decisiones de diseño:

1. desechar el protocolo binario y adoptar un protocolo ASCII en su lugar, puesto que eso permitiría una depuración más sencilla, ya que se podrían visualizar los comandos fácilmente mediante terminales de texto; y

2. descartar los protocolos I2C y SPI, y utilizar en su lugar el protocolo RS232-C, que permite tasas de transferencia mayores y su correcto funcionamiento ya ha sido comprobado al implementar el módulo `hostcom`.

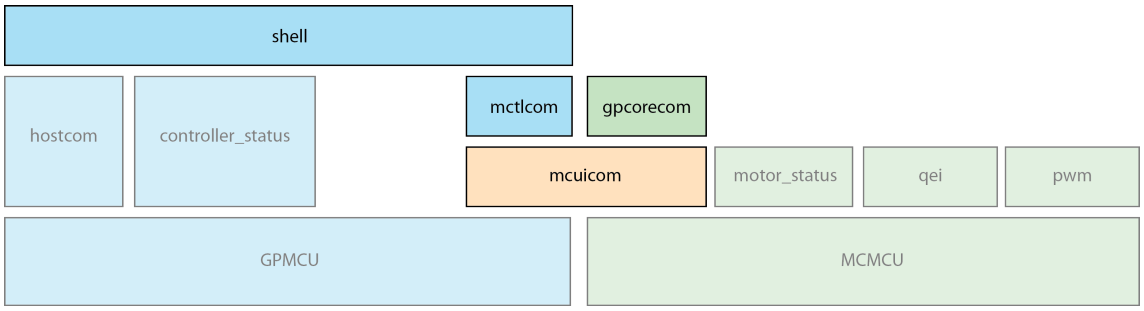
Como consecuencia, se implementará un protocolo de comandos de texto mediante el cual el GPMCU pueda comunicar al MCMCU los movimientos a ejecutar, así como la lectura y escritura de ciertos valores de configuración y del estado de los motores, como, por ejemplo, la lectura de la posición de los motores o la lectura y escritura de la velocidad, aceleración, etc.
Dado que la comunicación se hará mediante comandos de texto y los parámetros de los comandos serán de longitud arbitraria, será necesario elegir una marca de final de comando. Si además se restringe el formato de los comandos para que el nombre del comando esté formado únicamente por dos caracteres y sólo permitan cero, uno o dos parámetros, pero no más, se tendrá un formato de comandos igual al del repertorio de instrucciones del PC host, cuyo intérprete se ha implementado mediante los módulos `hostcom` (Sección 4.1.3, pág. 65) y `shell` (Sección 4.1.4, pág. 72).

Ya que el código del intérprete (`shell`) es considerablemente grande y está separado del módulo de comunicación —manejo de la UART— en el que se basa (`hostcom`), también para el módulo `mcuicom` adoptaremos este diseño, de manera que el módulo `mcuicom` implemente el manejo de la UART y luego un módulo aparte se encargue de la interpretación de los comandos. Esto es muy conveniente, además, por otro motivo: en general, tanto el GPMCU

[29] *MCU (Micro-Controller Unit) Inter-Communication*

como el MCMCU deberán implementar su propio intérprete, pero las funciones de comunicación y manejo de la UART ejecutarán las mismas acciones, por lo que, separando el módulo `mcuicom`, ambos microcontroladores podrán compartir dicho módulo, mientras que luego cada uno de los microcontroladores tendrá un módulo adicional para llevar a cabo la interpretación de comandos y que no será compartido.
Por tanto, tendremos un módulo que implementa el manejo de la UART y que será compartido por ambos microcontroladores (`mcuicom`), y otros dos módulos que implementan la interpretación de los comandos (`gpcorecom` para el MCMCU, y `mctlcom` para el GPMCU). Este diseño se ilustra en la Figura 4.11. En ella se ve cómo el módulo `mcuicom`, que realizará el

Figura 4.11: Arquitectura de RhinoChip OS tras la creación del módulo `mcuicom`




manejo de la UART 1 de ambos microcontroladores, se utilizará como capa base para construir el módulo `gpcorecom` para el MCMCU —que será el que implemente un intérprete de comandos similar al del módulo `shell`—, a la par que también servirá como base para el módulo `mctlcom` del GPMCU —que, en este caso, no implementará un intérprete de comandos (puesto que no es necesario por el momento), pero sí implementará las funciones requeridas para poder recibir las respuestas del MCMCU a los comandos enviados por el GPMCU (por ejemplo, cuando el GPMCU solicite un dato al MCMCU, como podría ser la posición de un motor, y, por tanto, el GPMCU espere una respuesta del MCMCU)—.
Además, en la Figura 4.11 también se muestra que, aparte de la creación de los módulos `mcuicom`, `gpcorecom` y `mctlcom`, el módulo `shell` también deberá sera modificado para empezar a implementar y probar la comunicación entre GPMCU y MCMCU al interpretar ciertos comandos del PC host.

Como consecuencia, el módulo `mcuicom` estará basado en el código del módulo `hostcom` ya comentado en la Sección 4.1.3 (pág. 65), donde únicamente cambiará el uso de la UART 2 por el uso de la UART 1, así como el nombre de las funciones. En la Tabla 4.4 (pág. 92) se muestra la correspondencia entre las funciones de los módulos `hostcom` y `mcuicom`.
A su vez, el módulo `gpcorecom` perteneciente al programa *motorctl* (que se ejecuta sobre el MCMCU) realizará la recepción e interpretación de los comandos enviados por el GPMCU, de forma similar a como el GPMCU recibe e interpreta los comandos enviados por el PC host, por lo que se basará en el diseño y la implementación del módulo `shell` comentado en la Sección 4.1.4 (pág. 72).
Finalmente, el módulo `mctlcom` se encargará de recibir las respuestas enviadas por el MCMCU a los comandos del GPMCU que requieran respuesta. Por este motivo, no es necesario implementar un intérprete de comandos completo, sino que bastará con que dicho módulo proporcione una función, que llamaremos `mctlcom_get_response`, y que se encargue de

Tabla 4.4: Comparativa de las funciones del módulo `hostcom` frente a las funciones del módulo `mcuicom`

| Función de `hostcom` | Función de `mcuicom` | Descripción de la función |
|---|---|---|
| `hostcom_setup` | `mcuicom_setup` | Configura la UART |
| `hostcom_send` | `mcuicom_send` | Envía un comando mediante polling |
| `_U2RXInterrupt` | `_U1RXInterrupt` | Mueve los bytes recibidos del buffer de la UART al buffer del sistema |
| `hostcom_cmd _available` | `mcuicom_cmd _available` | Devuelve verdadero si hay al menos un comando completo disponible en el buffer del sistema, o falso en caso contrario |
| `hostcom_read_cmd` | `mcuicom_read_cmd` | Extrae del buffer del sistema el primer comando completo |

extraer el primer mensaje disponible en el buffer hasta que se encuentre la marca de fin de comando.

Para la implementación de la función `mctlcom_get_response`, la primera cuestión que viene a la mente es si se deberá realizar la acción de forma síncrona (bloqueante) o asíncrona.
Realizar esta comunicación de forma síncrona tiene la ventaja de que la implementación es mucho más sencilla y que, al no bloquearse el programa *gpcore*, éste podría seguir recibiendo e interpretando comandos del PC host. En cambio, tiene la desventaja de que, si no se recibe ninguna respuesta del MCMCU por cualquier motivo (fallo de software, fallo de hardware, etc.), se produciría un interbloqueo que provocaría que el sistema dejase de responder a las órdenes del PC host y quizá incluso que no concluyese el movimiento del robot.
Por su parte, la realización de forma asíncrona tiene como mayor ventaja que garantizaría que no se produciría un interbloqueo debido a la espera por la respuesta, ya que la comunicación asíncrona permitiría al programa *gpcore* continuar su ejecución mientras espera la respuesta del MCMCU. El problema fundamental es que la implementación de este mecanismo es sumamente compleja, ya que a priori no existe una forma de pausar la ejecución del intérprete en un punto concreto del programa (función de interpretación del último comando), saltar a otra parte del programa (por ejemplo, el analizador sintáctico para interpretar el siguiente comando) y, cuando llegue la respuesta, retomar la ejecución de la función de interpretación del comando anterior para procesar la respuesta del MCMCU. Además, esto podría motivar problemas adicionales debido a la ejecución fuera de orden de los comandos del PC host.
Para resolver el problema de implementar la comunicación asíncrona y el problema de la ejecución fuera de orden, se podría utilizar un sistema operativo en tiempo real que implemente la multitarea y algún mecanismo de comunicación entre procesos (como semáforos o cerrojos), respectivamente. Sin embargo, esto se sale del alcance del proyecto y tampoco es estrictamente necesario, puesto que no se trata de un sistema industrial crítico. La solución más simple es, por tanto, implementar la comunicación síncrona (que es mucho más sencilla de implementar) y tratar de garantizar que no se producen interbloqueos por fallos en el software. Esto, por supuesto, no evitaría un interbloqueo causado por fallos en el hardware o por fallos en la comu-

nicación a nivel físico[30], pero, revisando bien las conexiones del prototipo, así como el diseño de la placa del sistema final, la probabilidad de un fallo hardware se reduce drásticamente. En cualquier caso, de producirse un interbloqueo, bastaría con reiniciar el controlador para sacar al sistema del estado de excepción y poder seguir utilizando el sistema robótico.

En definitiva, la implementación de la función `mctlcom_get_response` mediante comunicación síncrona es mucho más simple y garantiza la ejecución en orden de los comandos del PC host. Para ello, esta función puede utilizar las funciones del módulo `mcuicom`, tal como se muestra en el Listado 4.15.

Listado 4.15: Fragmento del cuerpo de la función `mctlcom_get_response` (fichero `gpcore/mctlcom.c`)

```
10 // Wait until the receive buffer has at least one full command
11 while (!mcuicom_cmd_available());
12
13 // Otherwise, if a command has been fully received, get it and
       extract the response code
14 // in order to return it to the callee
15 copied = mcuicom_read_cmd(response, size, &full);
```

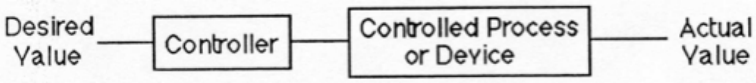
Para evitar los interbloqueos, se pondrá especial cuidado en la implementación y verificación del software, de manera que se tengan garantías de que no se produce tal estado de excepción en la ejecución del programa. Además, se propondrá como trabajo futuro la inclusión de un mecanismo de seguridad adicional mediante la fijación de un `timeout`, de manera que se interrumpa la espera cuando haya pasado un cierto tiempo sin recibirse ninguna respuesta. En cualquier caso, si se llegase a producir este estado de excepción, se podría sacar al sistema de dicho estado mediante la funcionalidad de reinicio del sistema, de manera que el sistema pueda seguir siendo utilizado mientras se investiga y resuelve el fallo.

### 4.1.8 Bucle de control PID

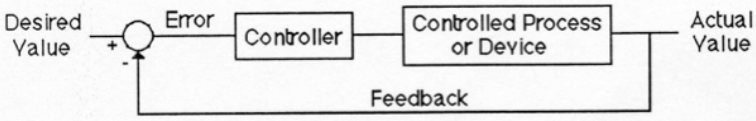
Hasta ahora, en lo relativo al control de motores, hemos diseñado y prototipado (1) una interfaz para los codificadores de posición (QEI) que permite medir la posición de cualquier motor en cada instante, (2) un módulo para la generación de señales de PWM y control del sentido de giro de los motores, y (3) una interfaz de intercomunicación entre el GPMCU y el MCMCU. Todo esto se ha hecho para tener un fundamento sólido sobre el que construir los algoritmos de movimiento de los motores, así como un sistema de control realimentado para la corrección de la posición de los motores. En esta sección nos ocuparemos del diseño y realización de este sistema de control realimentado, puesto que es la base para disponer de un sistema robótico robusto que mantenga su configuración a pesar de las perturbaciones externas, y luego será el fundamento sobre el que construyamos los algoritmos de movimiento (generación de un perfil de velocidad trapezoidal).

[30]Esto sucedería, por ejemplo, si los cables de conexión fallasen.

Figura 4.12: Sistemas de control




(a) Sistema de control en bucle abierto (sin realimentación)




(b) Sistema de control en bucle cerrado (realimentado)

La implementación de un sistema de control realimentado es esencial para un sistema robótico, puesto que se necesita una gran precisión en los movimientos a un punto concreto del espacio de trabajo, y dicho punto debe alcanzarse, idealmente, sin error de posición alguno. Un sistema de control realimentado busca suprimir el error entre la variable controlada —por ejemplo, la posición— y el valor medido para dicha variable haciendo uso de sensores —en nuestro caso, codificadores de posición incrementales—, para garantizar que se alcanza el valor deseado de la variable con el mínimo error posible. Un sistema de control realimentado, además, aumenta la tolerancia del sistema ante perturbaciones externas, puesto que, cuando se producen perturbaciones, el error en la medición aumenta y el sistema de control reacciona para rectificar dicho error. Esto implica que, si, por ejemplo, una fuerza externa actuase sobre el brazo robótico provocando su desplazamiento de la posición consignada, el sistema de control reaccionaría oponiéndose a la fuerza externa, para así mantener el brazo robótico en dicha posición, evitando que la fuerza externa lo desplace.

Existen diversas técnicas de control realimentado, pero una de las técnicas clásicas más frecuentemente utilizadas en la industria es el control PID[31], llamado así porque la salida del sistema de control en cada instante es

$$y(t) = K_P e(t) + K_I \int_0^t e(\tau)\,\mathrm{d}\tau + K_D \frac{\mathrm{d}e(t)}{\mathrm{dt}}, \tag{4.16}$$

que contiene un término **proporcional** (con constante $K_P$) al error $e$ en tiempo $t$, un término proporcional (con constante $K_I$) a la **integral** del error, y un término proporcional (con constante $K_D$) a la **derivada** del error.
Este tipo de sistema de control presenta ciertas propiedades cuyo análisis escapa del alcance de este proyecto, pero que se puede consultar en la bibliografía [5, 6][32]. A efectos prácticos, baste decir que es un método de control razonablemente bueno, y más que suficiente para la aplicación que nos ocupa, que, por tratarse de una aplicación educativa, no requiere técnicas de control más precisas y robustas. Además, el control PID es la técnica de control implementada en el sistema original, y es fácil de implementar y no exige grandes recursos computacionales, por lo que es una técnica idónea para el desarrollo de un prototipo funcional del controlador.

En la Ecuación 4.16 hemos presentado la expresión matemática de un sistema de control

[31] Proporcional-Integral-Derivativo

[32] Asimismo, el cálculo de los valores óptimos para las constantes $K_P$, $K_I$ y $K_D$ no se abordará aquí, puesto que también escapa del alcance del proyecto.

PID, donde la entrada es el error $e$ y la salida es la señal de control $y$. Sin embargo, en la realización del sistema de control mediante un computador digital (como es un microcontrolador), no podemos realizar operaciones en tiempo continuo, sino en tiempo discreto, con la consecuente discretización (o muestreo) de las señales de entrada y de control.
En la implementación del bucle de control PID en el microcontrolador, usaremos un timer que ejecutará una ISR con una frecuencia $f$, de manera que el periodo de muestreo será

$$T = \frac{1}{f}. \tag{4.17}$$

Por tanto, despreciando el tiempo de ejecución de la ISR, la expresión equivalente de la Ecuación 4.16 (pág. 94) en tiempo discreto será

$$y_k = K_P e_k + K_I \sum_{j=0}^{k} e_j T + K_D \frac{e_k - e_{k-1}}{T}, \tag{4.18}$$

para $k \in \mathbb{N} - \{0\}$, con la condición inicial $e_0 = 0$.

Para implementar el algoritmo de control PID, crearemos un nuevo módulo en el programa *motorctl*, que llamaremos con el mismo nombre. En el módulo `motorctl` tendremos que configurar un timer —usaremos el timer 3— para que muestree periódicamente la posición de los motores y ejecute el bucle de control PID. Para ello se deberá crear (1) una función `motorctl_setup` que realice la configuración del timer, (2) la ISR `_T3Interrupt` del timer, y, finalmente, (3) la función `motorctl`, que ejecutará el bucle de control PID para cada motor. Como frecuencia del timer deberá elegirse un valor $f$ menor que la frecuencia de muestreo de los codificadores de posición de la Sección 4.1.6 (pág. 84), $f_{\text{sample}} = 10\,\text{kHz}$. Por tanto, elegiremos la mitad, es decir, $f = 5\,\text{kHz}$, para lo cual basta un valor de pre-escalado igual a 1.

Listado 4.16: Definición de la función `motorctl_setup` (fichero `motorctl/motorctl.c`)

```
616 inline void motorctl_setup(void)
617 {
618     // Set up data structures for PID position control
619 
620     setup_pid_info();
621 
622     // Set up Timer 3 to implement PID control
623 
624     IFS0bits.T3IF = 0; // Clear the timer 3 interrupt flag
625     IEC0bits.T3IE = 1; // Enable timer 3 interrupts
626     PR3 = PR3VAL;      // Set the timer period
627     T3CONbits.TON = 1; // Start the timer
628 }
```

Adicionalmente, para dar soporte al bucle de control PID, se creará una estructura de datos `pid_info_t` que contenga toda la información necesaria para el control PID (como los va-

lores de las ganancias y el error). Durante la configuración inicial (llevada a cabo por la función `motorctl_setup`) también será necesario inicializar la estructura `pid_info_t` de cada motor, para lo cual se creará la función `setup_pid_info`. Además, se definirá un flag `pid_enabled` para cada motor, de manera que se pueda habilitar y deshabilitar el control PID de cada motor individualmente.
Finalmente, el bucle PID será implementado por la función `pid_loop`, que permitirá ejecutar una iteración del algoritmo de control PID para cualquier motor, tomando los valores necesarios por parámetro. De esta forma, la función `motorctl` invocará a la función `pid_loop` una vez por cada motor, proporcionándole los parámetros adecuados, y el valor devuelto por la función `pid_loop` será utilizado como valor del ciclo de trabajo del motor, que será adoptado mediante la invocación de la función `pwm_set_duty` correspondiente al canal de PWM del motor (véase Sección 4.1.5, pág. 76).

Listado 4.17: Fragmento de la función `motorctl` para el control de un motor (fichero `motorctl/motorctl.c`)

```
641 // If the PID control for motor A is enabled, control motor A,
642 // correcting its position according to the PID gains.
643 if (pid_enabled[MOTOR_A])
644 {
645     // Re-calculate PWM duty cycle using the PID controller
646     int duty = pid_loop(&pid_info[MOTOR_A], motor_steps[
        MOTOR_A], motor_desired_pos[MOTOR_A]);
647     // Translate the PWM duty cycle into a PWM level and a PWM
         direction
648     // and update the corresponding registers
649     unsigned char direction;
650     duty = abs_neg(duty, &direction);
651     direction = 1 - direction; // 'direction' needs to be
        inverted
652     // Perform the movement
653     pwm_set_duty1(duty);
654     DIR1 = direction;
655 }
```

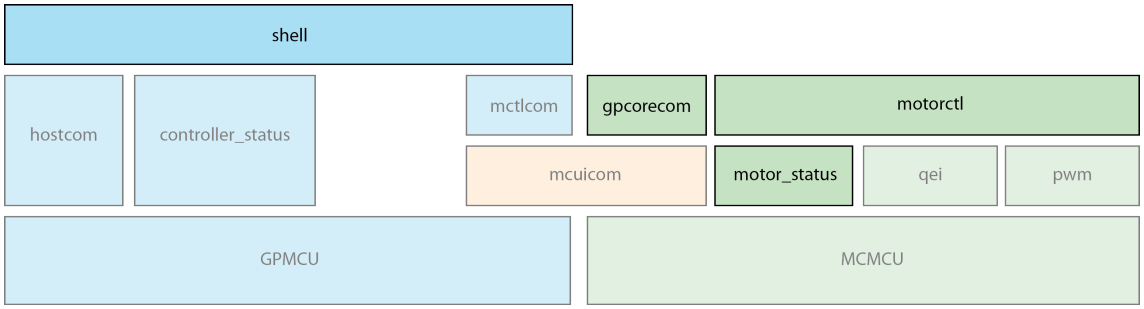
Listado 4.18: Fragmento de la función `pid_loop` para el cálculo de la Ecuación 4.18 (pág. 95) (fichero `motorctl/motorctl.c`)

```
486 pid_output = pid_info->KP * pid_info->curr_error
487           + pid_info->KI * pid_info->error_sum
488           + pid_info->KD * error_diff * T3FREQ;
```

Con lo realizado hasta el momento, el control de posición de los motores ya está implementado, pero todavía no es posible utilizarlo hasta que no se implementen comandos host para ello. Por este motivo, en esta serie de experimentos también se modificará los módulos `shell` y `gpcorecom` para implementar comandos de movimiento, junto con el módulo

motor_status para almacenar información de configuración y estado de los motores que, según el planteamiento inicial, estaría almacenada en el GPMCU, pero que se necesita en el MCMCU para poder realizar el control de motores. Con estas modificaciones, la arquitectura por capas de RhinoChip OS queda como se muestra en la Figura 4.13.

Figura 4.13: Arquitectura de RhinoChip OS tras la creación del módulo motorctl




Por este motivo, se implementarán los comandos de movimiento PD y PR (en el módulo shell) de manera que permitan indicar la posición de destino deseada en términos absolutos y en términos relativos, respectivamente. En este momento se implementará de forma que el movimiento sea directo, sin necesidad de enviar los comandos MI o MC —que es como debería funcionar en el sistema final—, puesto que por ahora sólo se usará estos comandos para la verificación del diseño y la implementación. Además, la implementación de los comandos PD y PR requiere la creación de nuevos comandos para la intercomunicación entre el GPMCU y el MCMCU (en el módulo gpcorecom), para permitir que, cuando el GPMCU reciba los comandos PD o PR, pueda comunicar al MCMCU qué movimientos debe ejecutar[33].
El movimiento en sí —y, por tanto, los comandos internos del MCMCU— tendrá lugar simplemente almacenando la posición de destino deseada —especificada con el comando PD o PR— en el registro motor_desired_pos del módulo motor_status. Tras la escritura en este registro, el valor de motor_desired_pos ya no coincidirá con el valor de la posición actual, almacenada en el registro motor_steps, por lo que habrá un error de posición no nulo y automáticamente el bucle de control PID activará los motores correspondientes para corregir dicho error. El movimiento terminará entonces cuando se consiga anular el error de posición, lo cual sólo ocurrirá cuando se alcance la posición de destino deseada, es decir, cuando la posición actual (almacenada en motor_steps) alcance el mismo valor que el registro motor_desired_pos.

En las pruebas realizadas inicialmente se ha utilizado los mismos valores de las ganancias $K_P = 46$, $K_I = 10$ y $K_D = 110$ que estaban configurados en el controlador Mark IV. Sin embargo, el comportamiento dinámico del sistema no resultó satisfactorio. Dicho comportamiento presentaba muchas oscilaciones y, además, el motor giraba de forma intermitente, por lo que presentaba aceleraciones y deceleraciones bruscas. Debido a que el análisis dinámico del sistema y la optimización de las ganancias del bucle PID escapa del alcance de este proyecto, se ha probado algunas combinaciones de valores, sin éxito, hasta que finalmente se ha optado por usar únicamente un control P (proporcional).
El control P es el enfoque clásico más simple, aunque a causa de sus desventajas se desarrollaron los tipos de control PI, PD y, finalmente, PID. Fundamentalmente, el control P no es capaz

[33]Los comandos internos del MCMCU se encuentran detallados en el Apéndice C (pág. 147).

de corregir el error en estado estacionario, por lo que, en general, si se comanda un movimiento desde la posición inicial $\theta_0$ a la posición final $\theta_f$, la posición en la que concluye el movimiento puede acabar siendo $\theta_f - e_{ss}$, donde $e_{ss} \neq 0$ es el error estacionario. Evidentemente, este fenómeno no es deseable. Sin embargo, con valores pequeños de $K_P$ se ha comprobado que esto nunca ocurre en el entorno de pruebas. Por tanto, se puede seguir utilizando el control P para el desarrollo y quizá incluso en producción, y se dejaría para fases futuras del proyecto la tarea de hacer un estudio dinámico del sistema y calcular las ganancias del bucle PID óptimas para el control.

### 4.1.9 Rutina de hard home

Debido a que los codificadores de posición de los motores son incrementales, no es posible conocer la posición absoluta de los motores cuando se reinicia el sistema a menos que se disponga de una posición de referencia conocida respecto a la cual medir los desplazamientos.

Para solventar este problema, los robots de Rhino Robotics disponen de un interruptor de final de carrera en cada articulación que permite llevar el brazo a una configuración (posición de los motores) bien conocida. De esta forma, cada vez que se inicia el sistema, es necesario realizar una operación de búsqueda de estos interruptores para encontrar dicha posición de referencia, tras lo cual se reiniciará los contadores de posición de los motores para tomar dicha posición como la posición cero. A este procedimiento se lo conoce como *hard home*, y, cada vez que se reinicia el sistema, se dice que se debe "ejecutar (o realizar) un *hard home*".

Por los motivos expuestos, la rutina de *hard home* es esencial para el uso del sistema, por lo que debe estar incluida ya en las primeras versiones del prototipo. Por supuesto, esta rutina podría ser implementada por software desde un PC si el controlador proporciona los comandos de movimiento e inspección del estado de los interruptores de final de carrera necesarios para ello, pero es mucho más práctico que el propio controlador ejecute esta acción por sí mismo, tal como hace el Mark IV.

Inicialmente se ha intentado realizar la rutina de hard home enteramente en el MCMCU. Sin embargo, debido a las necesidades de E/S del resto de módulos (PWM, QEI), el número de pines de E/S restantes en el MCMCU es insuficiente para monitorizar los interruptores de final de carrera, por lo que finalmente se ha decidido implementar la rutina de *hard home* usando los dos microcontroladores. Para ello, el GPMCU utilizará 6 pines de E/S para monitorizar el estado de los interruptores de final de carrera, e irá enviando las órdenes de movimiento al MCMCU para que mueva los motores.

El algoritmo de *hard home* es bastante simple. En primer lugar, hay que mover el motor, haciéndolo girar en un sentido determinado, hasta que alcance el interruptor de final de carrera y éste se active. Este movimiento se puede hace a velocidad alta, para disminiuir el tiempo que el usuario debe esperar para que se complete la rutina de *hard home*.
Entonces, una vez alcanzado el interruptor, deberá pararse el motor inmediatamente, lo cual dejará el interruptor pulsado. Entonces podrá continuar moviéndose el motor, esta vez más lentamente para tener mayor precisión, hasta que se desactive el interruptor. En todo momento se ha estado monitorizando el codificador del motor para conocer su posición, por lo que,

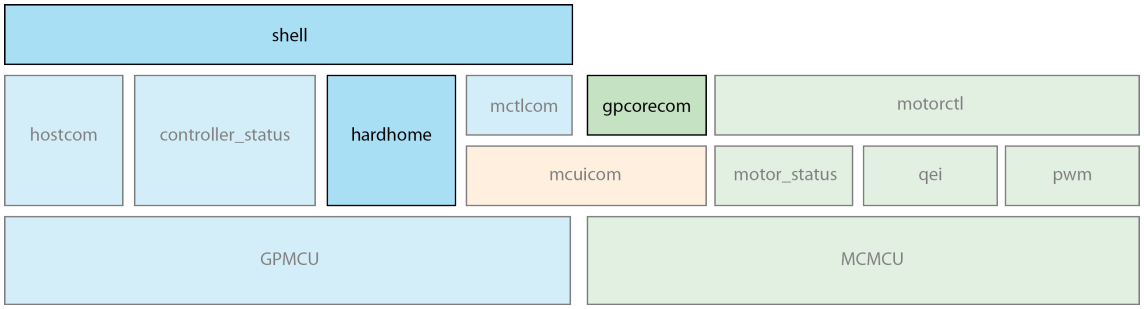
obteniendo la lectura del codificador en ese punto, se conseguirá la posición del que tomaremos como punto $B$.
En ese momento se deberá invertir el sentido de giro del motor y continuar moviéndolo a velocidad baja. El interruptor volverá a activarse, pero no deberá pararse el motor hasta que se pase el interruptor, es decir, mientras no vuelva a desactivarse. Cuando se desactive, se parará el motor nuevamente, y, obteniendo la lectura del codificador de posición, se tendrá el otro extremo del interruptor, que tomaremos como punto $A$.
Finalmente, puesto que conocemos las posiciones de los puntos $A$ y $B$ —los dos extremos del interruptor—, podrá obtenerse la posición exacta del interruptor como el punto medio de los puntos $A$ y $B$. Dicho de otra forma, el punto donde se encuentra el interruptor es el punto medio

$$\frac{A+B}{2}. \tag{4.19}$$

Para concluir la rutina de *hard home*, lo que deberemos hacer será mover el motor exactamente al punto medio tal como está expresado en la Ecuación 4.19, y reiniciar el contador de pasos del motor a cero. Entonces se tendrá el motor en una posición de referencia conocida, a partir de la cual se podrá medir todos los desplazamientos que se realice.

La implementación de la rutina de *hard home*, que se ejecutará cuando el PC host envíe el comando `HH`, conllevará, por tanto, modificaciones en ambos programas, tanto el programa *gpcore* que se ejecuta sobre el GPMCU como el programa *motorctl* que se ejecuta sobre el MCMCU. Esto se ilustra en la Figura 4.14, donde se ve la creación de un nuevo módulo

Figura 4.14: Arquitectura de RhinoChip OS tras la creación del módulo `hardhome`




`hardhome` en el programa *gpcore*, y la modificación de los módulos `shell` y `gpcorecom` para implementar los comandos necesarios. Estas modificaciones implican la implementación del comando host `HH`, así como la implementación de los comandos del MCMCU necesarios para que el GPMCU pueda controlar el movimiento de los motores durante la ejecución de la rutina de *hard home*.

Tras algunas pruebas preliminares, el enfoque final que se seguirá para la implementación de la rutina de *hard home* será mover primeramente el motor a velocidad alta sin control PID, controlando directamente el ciclo de trabajo de la señal de PWM, hasta que se alcance el interruptor de final de carrera. Entonces se volverá a habilitar el control PID y se empezará a aumentar la posición deseada de cuatro en cuatro, de manera que el bucle PID corrija la posición avanzando de cuatro en cuatro pasos, lo cual producirá un movimiento lento. Esto se hará hasta que se encuentre la posición de los dos puntos $A$ y $B$ mencionados anteriormente. Todo

esto se realizará en un nuevo módulo del programa *gpcore* que llamaremos `hardhome`, y donde habrá una función `hardhome_motor_x` para cada motor `x` (donde `x` puede ser cualquier letra de la `A` a la `F` según el motor que controle la función).
En este módulo deberá definirse primeramente una función `lmtswitch_setup` que configure los pines del puerto B a utilizar para la entrada digital, puesto que se necesitará los seis primeros pines del puerto B del GPMCU para monitorizar el estado de los interruptores de final de carrera.
Una vez configurado el puerto B para monitorizar los interruptores de final de carrera, podremos crear una función `hardhome` que llame a cada una de las subrutinas `hardhome_motor_x`. Entonces se desarrollará cada una de las funciones de forma que ejecute los pasos de la rutina de *hard home* tal como se ha explicado anteriormente, para lo cual se deberá usar los comandos del MCMCU implementados para tal efecto (habilitar y deshabilitar el control PID, mover de cuatro en cuatro pasos, etc.).

Tabla 4.5: Asignación de pines del módulo `hardhome`

| Pin | Nombre de la señal | Tipo | Descripción |
|---|---|---|---|
| RB0 | LMT_MA | Entrada digital | Señal de microswitch del motor A |
| RB1 | LMT_MB | Entrada digital | Señal de microswitch del motor B |
| RB2 | LMT_MC | Entrada digital | Señal de microswitch del motor C |
| RB3 | LMT_MD | Entrada digital | Señal de microswitch del motor D |
| RB4 | LMT_ME | Entrada digital | Señal de microswitch del motor E |
| RB5 | LMT_MF | Entrada digital | Señal de microswitch del motor F |

### 4.1.10 Perfil de velocidad trapezoidal

#### Generación programática de un perfil de velocidad trapezoidal

El controlador Mark IV mueve los motores que se encuentran en modo trapezoidal siguiendo un perfil de velocidad trapezoidal. Esto significa que el movimiento de las articulaciones no se lleva a cabo con la misma velocidad en cada punto de la trayectoria, sino que se divide el movimiento en tres fases:

1. una fase de despegue, en la que el motor de la articulación va incrementando su velocidad desde 0 hasta la velocidad máxima deseada,
2. una fase de velocidad constante, en la que el motor ha alcanzado la velocidad máxima deseada y, por tanto, la velocidad se satura y no sigue aumentando; y
3. una fase de asentamiento, en la que el motor comienza a disminuir su velocidad hasta que la velocidad llegue a ser cero cuando se alcanza el punto final de la trayectoria.

En esta situación, la curva de la velocidad del motor frente al tiempo será una recta de pendiente positiva en la fase de despegue, una recta de pendiente nula en la fase de velocidad constante,

y una recta de pendiente negativa en la fase de asentamiento, tal como se representa en la Figura 4.15(b) (pág. 102). Como es obvio, puesto que la aceleración instantánea es la derivada de la velocidad respecto al tiempo, la aceleración en la fase de despegue es una constante positiva, en la fase de velocidad constante es nula, y en la fase de asentamiento es una constante negativa, como se observa en la Figura 4.15(c) (pág. 102).

Para generar este perfil de velocidad se ha optado inicialmente por un enfoque programático basado en un método intuitivo sin un desarrollo formal, inspirado en el método descrito en [7], según el cual se parte de la posición inicial de la trayectoria —es decir, la posición actual del motor— con velocidad cero y aceleración constante, y se va sumando la aceleración a la velocidad hasta que llegue a la velocidad máxima deseada. De este modo, la velocidad aumentará con una tasa de cambio constante. A su vez, el valor de la velocidad en cada iteración del algoritmo para la fase de despegue se sumará o restará sucesivamente a la posición inicial, dependiendo de si el desplazamiento es positivo o negativo, respectivamente.
Cuando la velocidad llega a la velocidad máxima deseada, la velocidad se satura, lo cual significa que no se le sigue sumando el valor de aceleración. Sin embargo, sí que se sigue sumando (o restando) la velocidad a la posición, para continuar el movimiento trapezoidal.
Finalmente, cuando la articulación llega a la fase de asentamiento, comienza el proceso inverso del ejecutado en la fase de despegue, es decir, se va restando el valor de aceleración a la velocidad hasta que ésta última se convierta en cero. Asimismo, en cada iteración se suma (o resta) la velocidad a la posición. Este proceso continuará hasta que la articulación llegue a la posición de destino.

En el algoritmo aquí descrito, la posición a la que hemos hecho referencia es la que se tomará como posición de consigna del bucle PID. El bucle de control PID buscará seguir dicha posición, para lo cual corregirá el error en cada momento hasta que llegue a dicha posición de consigna. Puesto que la posición de consigna es incrementada a intervalos muy cortos con una frecuencia relativamente alta (desde el punto de vista macroscópico), el efecto que esto tendrá a ojos de un observador es que el robot realizará un movimiento continuo.

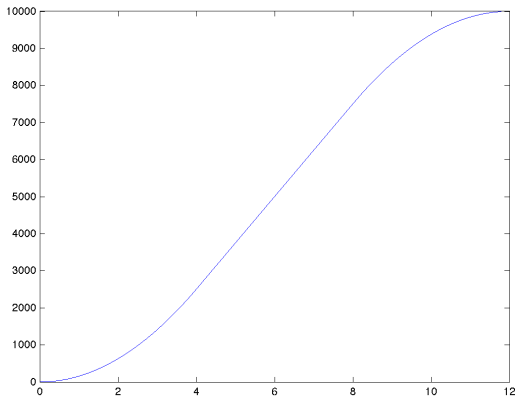
En el momento de las pruebas, este algoritmo producía un movimiento continuo, pero no ejecutaba el perfil de velocidad trapezoidal de forma precisa, y tampoco existía una forma analítica de especificar la velocidad y la aceleración deseadas haciendo uso de unidades de velocidad y aceleración intuitivas que permitiesen hacerse una idea de la velocidad y aceleración que tendría el motor al utilizar dichos valores como parámetros del algoritmo. Por tanto, este algoritmo no ha resultado satisfactorio. Por este motivo, hemos optado por estudiar cómo se comportaría el robot si se utilizase un método formal de planificación de trayectoria basado en interpolación polinómica [8].

### Planificación de trayectoria 2–1–2

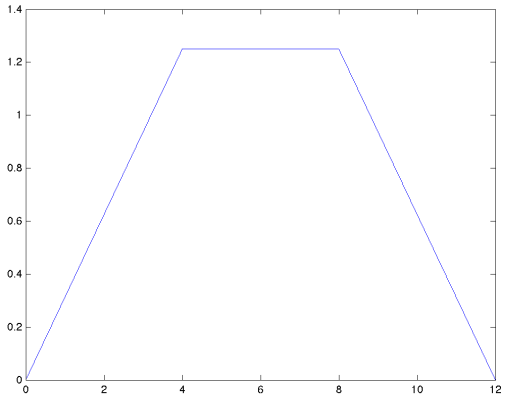
Un método para la planificación de trayectoria de un robot en el espacio de las articulaciones consiste en dividir la trayectoria en varios segmentos, usualmente tres, y utilizar polinomios para interpolar la trayectoria deseada [8].

Observando el perfil de velocidad trapezoidal de la Figura 4.15(b) (pág. 102), es evidente

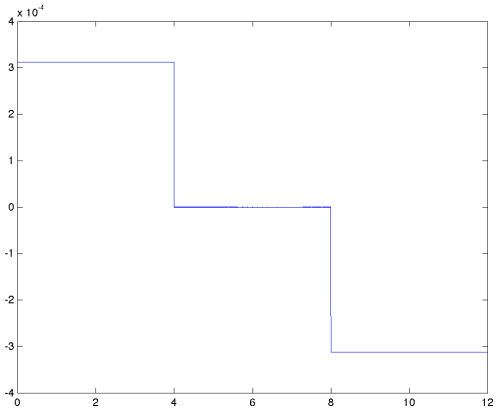
Figura 4.15: Representación gráfica de la trayectoria, velocidad y aceleración de un motor en modo trapezoidal (simulado en MATLAB)




(a) Trayectoria en el espacio de la articulación




(b) Perfil de velocidad de la articulación




(c) Aceleración instantánea del motor

que en el segmento 1 (fase de despegue) la velocidad viene dada por una función polinómica de grado 1, en el segmento 2 (fase de velocidad constante) viene dada por una función polinómica de grado 0 y en el segmento 3 (fase de asentamiento) viene dada nuevamente por una función polinómica de grado 1. Puesto que la posición se puede obtener como la antiderivada de la velocidad, es consecuencia directa que el segmento 1 de la posición debe ser un polinomio de grado 2, el segmento 2 debe ser un polinomio de grado 1 y el segmento 3 debe ser también un polinomio de grado 2. Dicho de otra forma, el segmento 1 de la posición es una parábola, el segmento 2 es una recta y el segmento 3 vuelve a ser una parábola.

Matemáticamente, podemos dividir la trayectoria $h$ en tres segmentos $h_1$, $h_2$ y $h_3$ de la forma

$$h(\tau) \triangleq \begin{cases} h_1(\tau), & \tau_0 \leq \tau < \tau_1 \\ h_2(\tau), & \tau_1 \leq \tau \leq \tau_2 \\ h_3(\tau), & \tau_2 < \tau \leq \tau_f \end{cases}, \tag{4.20}$$

donde la variable $\tau$ indica el tiempo, $\tau_0$ y $\tau_f$ son los instantes de tiempo inicial y final del movimiento, y $\tau_1$ y $\tau_2$ son los instantes de tiempo que marcan las fronteras de los segmentos. Puesto que comenzaremos a medir el tiempo desde el instante inicial de la trayectoria, tenemos que $\tau_0 = 0$ y, por tanto, la Ecuación 4.20 queda como

$$h(\tau) = \begin{cases} h_1(\tau), & 0 \leq \tau < \tau_1 \\ h_2(\tau), & \tau_1 \leq \tau \leq \tau_2 \\ h_3(\tau), & \tau_2 < \tau \leq \tau_f \end{cases}, \tag{4.21}$$

donde $h_1$ y $h_3$ serán parábolas, y $h_2$ será una recta. Sin embargo, antes de expresar estas funciones matemáticamente, debemos considerar unos cambios de variable para simplificar los cálculos sucesivos.

Dada la variable de tiempo real $\tau$ y las constantes de tiempo real $\tau_{i-1}$ y $\tau_i$, considérese el tiempo normalizado $t$ en el segmento $i$ tal que

$$t \triangleq \frac{\tau - \tau_{i-1}}{\tau_i - \tau_{i-1}}. \tag{4.22}$$

Con esta definición de $t$, para cualquier valor de $\tau \in [\tau_{i-1},\, \tau_i]$, tenemos que $t \in [0,\, 1]$.
De forma similar, la duración del segmento $i$ vendrá dada por

$$t_i \triangleq \tau_i - \tau_{i-1}. \tag{4.23}$$

Además, introduciremos un cambio de variable adicional sólo para el segmento 3, tal que

$$\bar{t} \triangleq t - 1. \tag{4.24}$$

Entonces, con los cambios de variable de las Ecuaciones 4.22 y 4.24, la expresión matemática de la trayectoria en los segmentos 1, 2 y 3 en términos del tiempo normalizado $t$ será

$$h_1(t) = a_{13}t^2 + a_{12}t + a_{11}, \tag{4.25}$$

$$h_2(t) = a_{22}t + a_{21}, \tag{4.26}$$

$$h_3(\bar{t}) = a_{33}\bar{t}^2 + a_{32}\bar{t} + a_{31}. \tag{4.27}$$

Puesto que la velocidad instantánea es la derivada de la posición $h$ respecto del tiempo real $\tau$, la velocidad instantánea $v_i$ en el segmento de trayectoria $h_i$ viene dada por

$$v_i(t) \triangleq \frac{\mathrm{d}h_i(t)}{\mathrm{d}\tau} = \frac{\mathrm{d}h_i(t)}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}\tau}, \tag{4.28}$$

donde debe aplicarse la regla de la cadena para poder calcular la derivada.
Según la definición de $t$ dada por la Ecuación 4.22 (pág. 103), la derivada de $t$ respecto de $\tau$ es

$$\frac{\mathrm{d}t}{\mathrm{d}\tau} = \frac{\mathrm{d}}{\mathrm{d}\tau}\left[\frac{\tau - \tau_{i-1}}{\tau_i - \tau_{i-1}}\right] = \frac{1}{\tau_i - \tau_{i-1}}. \tag{4.29}$$

Teniendo en cuenta la Ecuación 4.23 (pág. 103), esta derivada se puede expresar como

$$\frac{\mathrm{d}t}{\mathrm{d}\tau} = \frac{1}{\tau_i - \tau_{i-1}} = \frac{1}{t_i}. \tag{4.30}$$

Por tanto, la ecuación Ecuación 4.28 queda como

$$v_i(t) = \frac{1}{t_i}\left(2a_{i3}t + a_{i2}\right). \tag{4.31}$$

Asimismo, la aceleración instantánea $a_i$ en el segmento de trayectoria $h_i$ viene dada por

$$a_i(t) \triangleq \frac{\mathrm{d}v_i(t)}{\mathrm{d}\tau} = \frac{\mathrm{d}v_i(t)}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}\tau}. \tag{4.32}$$

Sustituyendo las Ecuaciones 4.31 y 4.30 en la Ecuación 4.32, obtenemos que

$$a_i(t) = \frac{2}{t_i^2}a_{i3}. \tag{4.33}$$

Como consecuencia, las velocidades correspondientes a los segmentos 1, 2 y 3 son

$$v_1(t) = \frac{1}{t_1}\left(2a_{13}t + a_{12}\right), \tag{4.34}$$

$$v_2(t) = \frac{1}{t_2}a_{21}, \tag{4.35}$$

$$v_3\left(\bar{t}\right) = \frac{1}{t_3}\left(2a_{33}\bar{t} + a_{32}\right). \tag{4.36}$$

De igual modo, las aceleraciones en los segmentos de trayectoria 1, 2 y 3 tienen la forma

$$a_1(t) = \frac{2}{t_1^2}a_{13}, \tag{4.37}$$

$$a_2(t) = 0, \tag{4.38}$$

$$a_3\left(\bar{t}\right) = \frac{2}{t_3^2}a_{33}. \tag{4.39}$$

Ahora que conocemos las expresiones matemáticas de las curvas de posición, velocidad y aceleración en términos de sus coeficientes, podemos hallar dichos coeficientes imponiendo ciertas ligaduras en los puntos inicial, final e intermedios de la trayectoria. Las ligaduras que podremos considerar son:

1. La posición de despegue debe ser igual a la posición inicial del movimiento[34]:

$$h_1\left(t=0\right)=\theta_0 \tag{4.40}$$

2. La velocidad de despegue debe ser nula:

$$v_1\left(t=0\right)=0 \tag{4.41}$$

3. La aceleración de despegue debe ser igual a la aceleración $\alpha$ deseada[35]:

$$a_1\left(t\right)=\alpha \tag{4.42}$$

4. En $\tau_1$ debe haber continuidad en posición:

$$h_1\left(t=1\right)=h_2\left(t=0\right) \tag{4.43}$$

5. En $\tau_1$ debe haber continuidad en velocidad:

$$v_1\left(t=1\right)=v_2\left(t=0\right) \tag{4.44}$$

6. En $\tau_2$ debe haber continuidad en posición:

$$h_2\left(t=1\right)=h_3\left(\bar{t}=-1\right) \tag{4.45}$$

7. En $\tau_2$ debe haber continuidad en velocidad:

$$v_2\left(t=1\right)=v_3\left(\bar{t}=-1\right) \tag{4.46}$$

8. La posición al final de la fase de asentamiento debe coincidir con la posición de destino:

$$h_3\left(\bar{t}=0\right)=\theta_f \tag{4.47}$$

9. La velocidad al final de la fase de asentamiento debe ser nula:

$$v_3\left(\bar{t}=0\right)=0 \tag{4.48}$$

10. La aceleración durante la fase de asentamiento debe ser opuesta a la aceleración $\alpha$:

$$a_3\left(\bar{t}\right)=-\alpha \tag{4.49}$$

[34]Ésta es la posición actual del motor cuando el movimiento es comandado, así que es un parámetro conocido de antemano.

[35]Su valor se obtiene a partir de uno o varios parámetros de la configuración del controlador, por lo que su valor es conocido.

**Resolución de las ligaduras de posición, velocidad y aceleración**

Antes de proseguir con la aplicación de las ligaduras para el cálculo de los coeficientes, debemos tener en cuenta que los valores de la velocidad máxima del motor[36], la velocidad máxima deseada $\omega_d$[37] y la aceleración deseada $\alpha$[37] son conocidos a priori, lo cual nos permite calcular el tiempo de despegue $\tau_a = t_1$. Además, para mayor simplicidad, impondremos como requisito que la duración de los tres segmentos de trayectoria sea la misma[38], es decir, que

$$t_1 = t_2 = t_3 = \tau_a. \tag{4.50}$$

Este cálculo se lleva a cabo considerando que la aceleración en el segmento 1 es

$$\alpha = \frac{\omega_d - 0}{\tau_1 - \tau_0} = \frac{\omega_d}{\tau_1} = \frac{\omega_d}{\tau_a}. \tag{4.51}$$

Por tanto, se tiene que

$$\tau_a = \frac{w_d}{\alpha}. \tag{4.52}$$

Tras estas consideraciones, ya podemos entrar de lleno en la resolución de las ecuaciones para obtener los coeficientes que definen la trayectoria deseada dados los parámetros $\theta_0$, $\theta_f$, $\omega_d$ y $\alpha$.

Para ello comenzamos con la ligadura de la Ecuación 4.40 (pág. 105):

$$h_1\left(t = 0\right) = \theta_0 \Longleftrightarrow a_{11} = \theta_0. \tag{4.53}$$

Asimismo, la ligadura de la Ecuación 4.41 (pág. 105) nos lleva a

$$v_1\left(t = 0\right) = 0 \Longleftrightarrow \frac{1}{t_1} a_{12} = 0 \Longleftrightarrow a_{12} = 0. \tag{4.54}$$

Finalmente, a partir de la ligadura de la Ecuación 4.42 (pág. 105) obtenemos que

$$a_1\left(t = 0\right) = \alpha \Longleftrightarrow \frac{2}{t_1} a_{13} = \alpha \Longleftrightarrow a_{13} = \frac{\alpha t_1^2}{2} = \frac{\omega_d^2}{2\alpha}. \tag{4.55}$$

Puesto que las ligaduras iniciales y finales son las más sencillas, continuaremos con la ligadura de la Ecuación 4.47 (pág. 105):

$$h_3\left(\bar{t} = 0\right) = \theta_f \Longleftrightarrow a_{31} = \theta_f. \tag{4.56}$$

[36] Ésta es la velocidad máxima a la que se puede mover el motor cuando se alimenta con el voltaje máximo especificado por el fabricante.

[37] Éste es un parámetro que se calcula a partir de la velocidad máxima del motor y los valores de la configuración del controlador introducidos por el usuario.

[38] En teoría esto no es realista, puesto que, en general, $t_1 = t_3$, pero la duración $t_2$ es arbitraria y tal que $t_1 \neq t_2 \neq t_3$, puesto que depende del desplazamiento de la trayectoria. Sin embargo, en los experimentos realizados con esta y otras formas de calcular los tiempos (por ejemplo, imponiendo $t_2 = 2t_1$ o calculando $t_2$ para cumplir que el motor vaya a la velocidad deseada), se ha visto que los otros enfoques no tienen un buen desempeño y éste sigue siendo el mejor y, además, tiene un desempeño satisfactorio si se ajustan bien los parámetros de configuración.

Igualmente, de la ligadura de la Ecuación 4.48 (pág. 105):

$$v_3\left(\bar{t}=0\right)=0 \Longleftrightarrow \frac{1}{t_3}a_{32}=0 \Longleftrightarrow a_{32}=0. \tag{4.57}$$

Por último, de la ligadura de la Ecuación 4.49 (pág. 105):

$$a_3\left(\bar{t}\right)=-\alpha \Longleftrightarrow \frac{2}{t_3^2}a_{33}=-\alpha \Longleftrightarrow a_{33}=-\frac{\alpha t_3}{2}=-\frac{\omega_d^2}{2\alpha}=-a_{13}. \tag{4.58}$$

Ahora ya sólo quedan por calcular los coeficientes $a_{22}$ y $a_{21}$. A partir de la ligadura de la Ecuación 4.43 (pág. 105) tenemos que

$$h_1\left(t=1\right)=h_2\left(t=0\right) \Longleftrightarrow a_{13}+a_{12}+a_{11}=a_{21} \Longleftrightarrow a_{21}=\frac{\alpha t_1^2}{2}+\theta_0. \tag{4.59}$$

Para concluir, por medio de la ligadura de la Ecuación 4.44 (pág. 105) llegamos a que

$$v_1\left(t=1\right)=v_2\left(t=0\right) \Longleftrightarrow \frac{1}{t_1}\left(2a_{13}+a_{12}\right)=a_{22} \Longleftrightarrow a_{22}=\alpha t_1=\omega_d. \tag{4.60}$$

Por tanto, obtenemos las siguientes expresiones finales para los segmentos de la trayectoria:

$$h_1\left(t\right)=\frac{\omega_d^2}{2\alpha}t^2+\theta_0, \tag{4.61}$$

$$h_2\left(t\right)=\omega_d t+\left(\frac{\omega_d^2}{2\alpha}+\theta_0\right), \tag{4.62}$$

$$h_3\left(\bar{t}\right)=-\frac{\omega_d^2}{2\alpha}\bar{t}^2+\theta_f. \tag{4.63}$$

Ahora sólo resta deshacer los cambios de variable de las Ecuaciones 4.22 y 4.24 (pág. 103), tras lo cual obtenemos:

$$h_1\left(\tau\right)=\frac{\omega_d^2}{2\alpha}\left(\frac{\tau}{\tau_1}\right)^2+\theta_0, \tag{4.64}$$

$$h_2\left(\tau\right)=\omega_d\frac{\tau-\tau_1}{\tau_2-\tau_1}+\left(\frac{\omega_d^2}{2\alpha}+\theta_0\right), \tag{4.65}$$

$$h_3\left(\tau\right)=-\frac{\omega_d^2}{2\alpha}\left(\frac{\tau-\tau_2}{\tau_f-\tau_2}-1\right)^2+\theta_f. \tag{4.66}$$

**Resolución de las ligaduras de posición**

En los experimentos realizados con los segmentos de trayectoria de las Ecuaciones 4.64, 4.65 y 4.66, no se ha obtenido resultados satisfactorios. Concretamente, el movimiento de las articulaciones no seguía un perfil de velocidad trapezoidal.

Por ello, se ha desechado las ligaduras de continuidad en velocidad en los instantes $\tau_1$ y $\tau_2$ y se ha mantenido sólo las ligaduras de continuidad en posición. Igualmente, se ha eliminado las ligaduras de las aceleraciones inicial y final. Para eliminar los grados de libertad adicionales que esto ocasiona, se ha impuesto las dos condiciones siguientes en la trayectoria (además de la continuidad en posición):

1. En el instante $\tau_1$ debe haberse recorrido el 25 % de la trayectoria, es decir,

$$h_1\left(t=1\right)=h_2\left(t=0\right)=\theta_1 \tag{4.67}$$

donde se ha usado la definición

$$\theta_1 \triangleq \theta_0+0.25\left(\theta_f-\theta_0\right) \tag{4.68}$$

2. En el instante $\tau_2$ debe haberse recorrido el 75 % de la trayectoria, es decir,

$$h_2\left(t=1\right)=h_3\left(\bar{t}=-1\right)=\theta_2 \tag{4.69}$$

donde se ha usado la definición

$$\theta_2 \triangleq \theta_0+0.75\left(\theta_f-\theta_0\right) \tag{4.70}$$

Después de introducir estas nuevas ligaduras, las ligaduras iniciales y finales permanecen iguales y siguen proporcionando los coeficientes

$$a_{11}=\theta_0, \tag{4.71}$$
$$a_{12}=0, \tag{4.72}$$
$$a_{31}=\theta_f, \tag{4.73}$$
$$a_{32}=0, \tag{4.74}$$

pero las ligaduras intermedias, que ahora se limitan a las ligaduras de las Ecuaciones 4.67 y 4.69, deben ser resueltas de nuevo.

De este modo, a partir de la Ecuación 4.67 obtenemos que

$$h_1\left(t=1\right)=h_2\left(t=0\right)=\theta_1 \Longleftrightarrow a_{13}+a_{12}+a_{11}=a_{21}=\theta_1. \tag{4.75}$$

Partiendo de la Ecuación 4.75 podemos conseguir entonces los coeficientes $a_{21}$ y $a_{13}$:

$$a_{21}=\theta_1, \tag{4.76}$$
$$a_{13}+\cancelto{0}{a_{12}}+\cancelto{\theta_0}{a_{11}}=\theta_1 \Longleftrightarrow a_{13}=\theta_1-\theta_0. \tag{4.77}$$

Asimismo, a partir de la Ecuación 4.69 tenemos que

$$h_2\left(t=1\right)=h_3\left(\bar{t}=-1\right)=\theta_2 \Longleftrightarrow a_{22}+a_{21}=a_{33}-a_{32}+a_{31}=\theta_2. \tag{4.78}$$

De forma similar, partiendo de la Ecuación 4.78 conseguimos los otros dos coeficientes restantes:

$$a_{22}+\cancelto{\theta_1}{a_{21}}=\theta_2 \Longleftrightarrow a_{22}=\theta_2-\theta_1, \tag{4.79}$$
$$a_{33}-\cancelto{0}{a_{32}}+\cancelto{\theta_f}{a_{31}}=\theta_2 \Longleftrightarrow a_{33}=\theta_2-\theta_f. \tag{4.80}$$

Por tanto, llegamos a las siguientes expresiones finales para los segmentos de la trayectoria:

$$h_1(t) = (\theta_1 - \theta_0)\, t^2 + \theta_0, \tag{4.81}$$
$$h_2(t) = (\theta_2 - \theta_1)\, t + \theta_1, \tag{4.82}$$
$$h_3\left(\bar{t}\right) = (\theta_2 - \theta_f)\, \bar{t}^2 + \theta_f. \tag{4.83}$$

Ahora sólo resta deshacer los cambios de variable de las Ecuaciones 4.22 y 4.24 (pág. 103), tras lo cual obtenemos:

$$h_1(\tau) = (\theta_1 - \theta_0) \left(\frac{\tau}{\tau_1}\right)^2 + \theta_0, \tag{4.84}$$
$$h_2(\tau) = (\theta_2 - \theta_1) \frac{\tau - \tau_1}{\tau_2 - \tau_1} + \theta_1, \tag{4.85}$$
$$h_3(\tau) = (\theta_2 - \theta_f) \left(\frac{\tau - \tau_2}{\tau_f - \tau_2} - 1\right)^2 + \theta_f. \tag{4.86}$$

Éstas son las expresiones finales de cada segmento de trayectoria que se usará en la implementación. En las pruebas realizadas se ha visto que el comportamiento de este método es mucho mejor que los demás desarrollados. Si bien bajo ciertas condiciones este método produce paradas bruscas (en ocasiones con ligeras oscilaciones) cuando el motor llega al final de la trayectoria, esto se puede controlar eligiendo apropiadamente la aceleración del sistema[39], tal como se comentará en la siguiente sección.

### Implementación de la planificación de trayectoria

Puesto que la planificación de trayectoria para producir un perfil de velocidad trapezoidal es otra tarea de control de motores, lo más lógico es incluirla en el módulo `motorctl`, donde también se realiza el control PID. Tal como se ha explicado anteriormente en la Sección 4.1.10 (pág. 100), el perfil de velocidad trapezoidal se genera muestreando la trayectoria planificada con un cierto periodo de muestreo $T$ y escribiendo la posición $h_k = h(\tau_0 + kT)$, obtenida en el instante de tiempo discreto $k$, en el registro de posición de consigna del bucle de control PID, lo cual provocará que el bucle PID produzca la señal de control de motores adecuada para corregir la posición y que el motor realice el movimiento para alcanzar la posición $h_k$. Puesto que el desplazamiento entre $h_{k-1}$ y $h_k$ es, en general, variable, es esta diferencia la que producirá los cambios en la velocidad y la aceleración necesarios para que el motor exhiba un comportamiento trapezoidal.

En este momento, la decisión de diseño más importante es la elección del periodo de muestreo $T$. En el sistema robótico PUMA de Unimation se usa este mismo enfoque con $T = 28\,\text{ms}$ [8]. Por este motivo, hemos usado este valor como referencia. Además, con los diferentes métodos de planificación de trayectoria desarrollados hemos probado periodos desde $T = 100\,\text{ms}$ hasta $T = 10\,\text{ms}$, pasando por valores como $T = 75\,\text{ms}$, $T = 50\,\text{ms}$ y $T = 25\,\text{ms}$, pero finalmente se ha visto que $T = 28\,\text{ms}$ es un valor idóneo para el método de planificación de trayectoria que finalmente se utilizará, tal como se comentó en la sección anterior.

[39] Éste es un valor que se modifica en la configuración del controlar.

Para realizar el muestreo de la trayectoria y comandar la posición al bucle PID con el fin de ejecutar un movimiento trapezoidal, se utilizará el timer 4 con periodo $T = 28\,\text{ms}$, lo cual requiere un valor de pre-escalado igual a 256. De esta manera, con $f = \frac{1}{T}$, el valor del registro `PR4` se puede calcular como

$$\text{PR4} = \left\lfloor \frac{f_{\text{CY}}}{f \cdot 256} \right\rfloor . \tag{4.87}$$

Por tanto, se deberá modificar la función `motorctl_setup` para configurar también el timer 4. Además, se desarrollará la función `generate_trapezoidal_profile` que, a su vez, invocará las subrutinas `generate_trapezoidal_profile_motor_x` para cada motor `x` que deba ejecutar un movimiento trapezoidal, y que devolverán el valor de $h_k$ para el motor correspondiente[40].
Para dar soporte a estas funciones, será necesaria una estructura de datos que almacene temporalmente la información necesaria para la generación del perfil mientras el movimiento está en ejecución. Esta estructura `motorctl_info_t` contendrá los campos mostrados en el Listado 4.19.

Listado 4.19: Definición de la estructura `motorctl_info_t` (fichero `motorctl/motorctl.c`)

```
72 typedef struct {
73     int       enabled    : 1;
74     float     wdes;
75     float     alpha;
76     int       theta0;
77     int       theta1;
78     int       theta2;
79     int       thetaf;
80     float     tau1;
81     float     tau2;
82     float     tauf;
83     float     tau;
84     int       position;
85 } motorctl_info_t;
```

De esta estructura deberá haber una variable para cada motor, por lo que se usará un array llamado `motorctl_info` para tal fin. Adicionalmente, cada una de las estructuras de este array deberá ser inicializada antes de ejecutar un movimiento trapezoidal, para lo cual se desarrollará la función `setup_trapezoidal_movement`, que será invocada por la función `generate_trapezoidal_profile` antes de ejecutar el movimiento.
Finalmente, la ISR `_T4Interrupt` invocará esta última función con el fin de obtener la siguiente posición de la trayectoria, $h_k$, y escribirá dicha posición en el registro de posición deseada, `motor_desired_pos`, de cada motor habilitado en modo trapezoidal y que deba ejecutar un movimiento en ese instante. Cuando un motor termine su movimiento, el campo

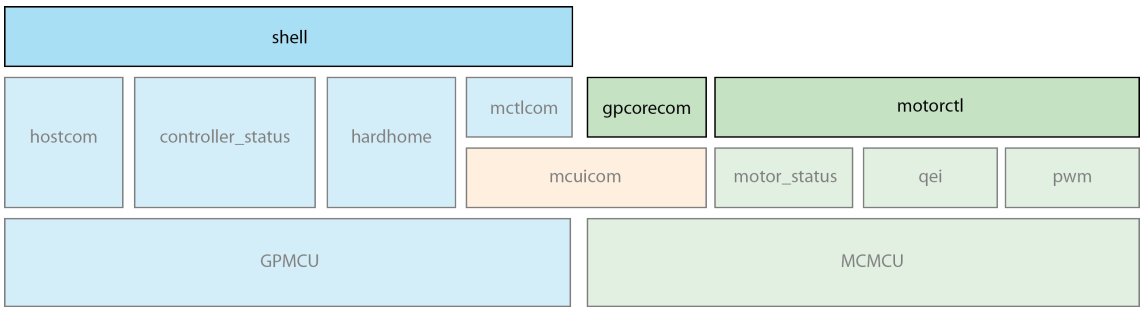
[40] Nótese que devuelven un solo valor, puesto que una invocación corresponde a una única iteración de la generación del perfil trapezoidal.

`enabled` de `motorctl_info_t` será deshabilitado, y la ISR dejará de actualizar la posición deseada de ese motor.

Para iniciar un movimiento trapezoidal desde el intérprete de comandos, se deberá modificar también el módulo `shell` con el fin de implementar el comando `MI` y actualizar los comandos `PD` y `PR`. Estos dos últimos comandos deberán modificarse para que no escriban la posición de destino directamente en `motor_desired_pos` —que es el registro de posición de consigna del bucle PID y, por tanto, inicia el movimiento automáticamente[41]—, sino primero en un registro intermedio llamado `motor_commanded_pos`. De esta forma, cuando se reciba el comando `MI`, se invocará una nueva función que se creará también en el módulo `motorctl`, la función `motorctl_move`. Esta función será la que iniciará el movimiento trapezoidal, para lo cual deberá habilitar el timer 4 —puesto que, hasta que no deba empezar el movimiento, estará deshabilitado— y copiar el valor de `motor_commanded_pos` al campo de la estructura `motorctl_info_t` que almacena la posición de destino del movimiento trapezoidal. Entonces, al habilitar el timer 4, comenzará el movimiento, y éste terminará cuando se llegue a la posición de destino y se haya deshabilitado el campo `enabled` de todos los motores, así como el timer 4.

Puesto que las tareas se reparten entre ambos microcontroladores, GPMCU y MCMCU, se requiere la adición de nuevos comandos al repertorio de instrucciones del MCMCU, de manera que el GPMCU pueda enviar las órdenes correspondientes al MCMCU. Por ello, las modificaciones descritas afectan a los módulos `motorctl`, `shell` y `gpcorecom`, tal como se muestra en la Figura 4.16.

Figura 4.16: Arquitectura de RhinoChip OS tras la implementación del perfil de velocidad trapezoidal




Tras realizar la implementación aquí descrita, se ha procedido entonces a las pruebas de este nuevo algoritmo. Como ya se ha comentado, este algoritmo es el que mejor se comporta de todas las variaciones que se ha probado, si bien en unas ocasiones puede presentar una velocidad muy baja, y en otras ocasiones paradas bruscas al final de la trayectoria.
En las pruebas se ha comprobado que estos fenómenos tienen lugar debido a un ajuste no óptimo de la *aceleración del sistema*, que es un parámetro de configuración del controlador. Concretamente, si la *aceleración del sistema* es muy baja, el movimiento será muy lento, mientras que, si es muy elevada, el motor realizará una parada brusca al final del movimiento. Por este motivo, se ha conducido una serie de pruebas para determinar experimentalmente el valor

[41] Esto se hizo así en un experimento anterior para facilitar la verificación del programa.

de aceleración adecuado para cada movimiento. Como resultado se ha obtenido que, para desplazamientos dentro de un cierto rango de pasos, la aceleración no debe superar un cierto valor límite, puesto que, a partir de ese valor, comienzan a producirse los fenómenos indeseados que se ha mencionado. Los resultados obtenidos se resumen en la Tabla 4.6.

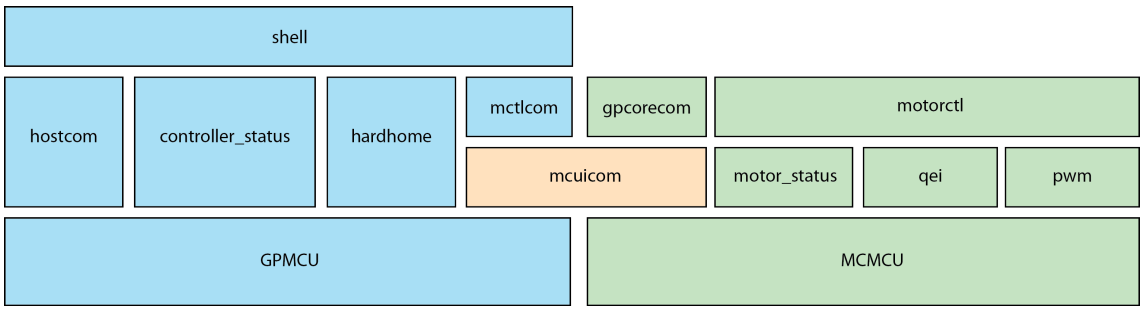
Tabla 4.6: Valores de la aceleración del sistema máximos para cada desplazamiento

| Desplazamiento | Aceleración máxima |
|---|---|
| 0–3000 pasos | 100 |
| 3000–4000 pasos | 75 |
| 4000–7000 pasos | 50 |
| 7000 pasos en adelante | 35 |

### 4.1.11 Resumen del diseño de RhinoChip

Tras todos los experimentos e iteraciones de diseño explicadas en las secciones anteriores, se ha llegado al diseño final del primer prototipo del controlador RhinoChip y su firmware RhinoChip OS, cuya arquitectura por capas en su estado final se muestra en la Figura 4.17. En

Figura 4.17: Arquitectura por capas de RhinoChip OS en su estado final




dicha figura se ilustra la división de tareas entre los dos microcontroladores, el GPMCU y el MCMCU. El GPMCU realiza primordialmente la interpretación de los comandos del PC host, y se comunica con el MCMCU para encomendarle la ejecución de los movimientos ordenados por el PC host. De esa forma, el MCMCU es el encargado de las tareas de control de motores, que pueden resumirse en (1) monitorizar la posición de los motores por medio de los codificadores incrementales, (2) ejecutar los movimientos ordenados siguiendo un perfil de velocidad trapezoidal, y (3) rectificar la posición de los motores para compensar cualquier perturbación externa y así mantener la posición deseada una vez se ha completado los movimientos ordenados.

Para la implementación de estas tareas se ha necesitado el uso de diversos timers del MCMCU con el fin de realizar el muestreo de señales o generar señales de salida periódicas. La información de configuración de los timers utilizados se resume en la Tabla 4.7 (pág. 113).

Asimismo, para la lectura de señales de entrada y la generación de señales de control digitales ha sido necesario utilizar diversos pines de E/S de ambos microcontroladores. Estas señales

Tabla 4.7: Temporizadores utilizados

| Microcontrolador | Temporizador | Tarea | Frecuencia | Periodo |
|---|---|---|---|---|
| MCMCU | Timer 1 | PWM | 5 kHz | 0.2 ms |
| MCMCU | Timer 2 | QEI | 10 kHz | 0.1 ms |
| MCMCU | Timer 3 | PID | 5 kHz | 0.2 ms |
| MCMCU | Timer 4 | Perfil de velocidad | 37 Hz | 28 ms |

son todas para el control de motores, pero, puesto que son muchas señales y el número de pines de E/S de los microcontroladores es bastante limitado, no ha sido suficiente con utilizar los pines del MCMCU, sino que ha sido necesario usar también algunos pines del GPMCU para implementar el *hard home*. Los pines del GPMCU utilizados se detallan en la Tabla 4.8, y los

Tabla 4.8: Asignación de pines del GPMCU

| Pin | Nombre de la señal | Tipo | Descripción |
|---|---|---|---|
| RB0 | LMT_MA | Entrada digital | Señal del microswitch del motor A |
| RB1 | LMT_MB | Entrada digital | Señal del microswitch del motor B |
| RB2 | LMT_MC | Entrada digital | Señal del microswitch del motor C |
| RB3 | LMT_MD | Entrada digital | Señal del microswitch del motor D |
| RB4 | LMT_ME | Entrada digital | Señal del microswitch del motor E |
| RB5 | LMT_MF | Entrada digital | Señal del microswitch del motor F |

del MCMCU en la Tabla 4.9.

Tabla 4.9: Asignación de pines del MCMCU

| Pin | Nombre de la señal | Tipo | Descripción |
|---|---|---|---|
| RB0 | QEA_MA | Entrada digital | Señal $A$ del motor A |
| RB1 | QEB_MA | Entrada digital | Señal $B$ del motor A |
| RB2 | QEA_MB | Entrada digital | Señal $A$ del motor B |
| RB3 | QEB_MB | Entrada digital | Señal $B$ del motor B |
| RB4 | QEA_MC | Entrada digital | Señal $A$ del motor C |
| RB5 | QEB_MC | Entrada digital | Señal $B$ del motor C |
| RB6 | DIR1 | Salida digital | Selección de sentido de giro del canal 1 (motor A) |
| RB7 | DIR2 | Salida digital | Selección de sentido de giro del canal 2 (motor B) |
| RB8 | DIR3 | Salida digital | Selección de sentido de giro del canal 3 (motor C) |
| RC13 | DIR4 | Salida digital | Selección de sentido de giro del canal 4 (motor D) |
| RC14 | DIR5 | Salida digital | Selección de sentido de giro del canal 5 (motor E) |
| RD2 | QEA_MD | Entrada digital | Señal $A$ del motor D |
| RD3 | QEB_MD | Entrada digital | Señal $B$ del motor D |
| RE0 | PWM1 | Salida de PWM | Señal de PWM del canal 1 (motor A) |
| RE1 | PWM2 | Salida digital | Señal de PWM del canal 2 (motor B) |

| Pin | Nombre de la señal | Tipo | Descripción |
|---|---|---|---|
| RE2 | PWM3 | Salida de PWM | Señal de PWM del canal 3 (motor C) |
| RE3 | PWM4 | Salida digital | Señal de PWM del canal 4 (motor D) |
| RE4 | PWM5 | Salida de PWM | Señal de PWM del canal 5 (motor E) |
| RE5 | PWM6 | Salida digital | Señal de PWM del canal 6 (motor F) |
| RE8 | DIR6 | Salida digital | Selección de sentido de giro del canal 6 (motor F) |
| RF0 | QEA_ME | Entrada digital | Señal $A$ del motor E |
| RF1 | QEB_ME | Entrada digital | Señal $B$ del motor E |
| RF4 | QEA_MF | Entrada digital | Señal $A$ del motor F |
| RF5 | QEB_MF | Entrada digital | Señal $B$ del motor F |

Finalmente, para dar soporte a la comunicación entre el PC host y el GPMCU, así como entre el GPMCU y el MCMCU, se ha utilizado las dos UARTs del GPMCU (UART 1 y UART 2), y la UART 1 del MCMCU, con los parámetros de configuración detallados en la Tabla 4.10.

Tabla 4.10: Configuración de las UARTs del GPMCU

| Parámetro | UART 1[42] | UART 2 |
|---|---|---|
| Tasa de transferencia | 9600 baudios | 9600 baudios |
| Bits de datos | 8 | 8 |
| Bits de parada | 1 | 1 |
| Paridad | sin paridad | sin paridad |

## 4.2 Diseño e implementación de la aplicación MarkCommander

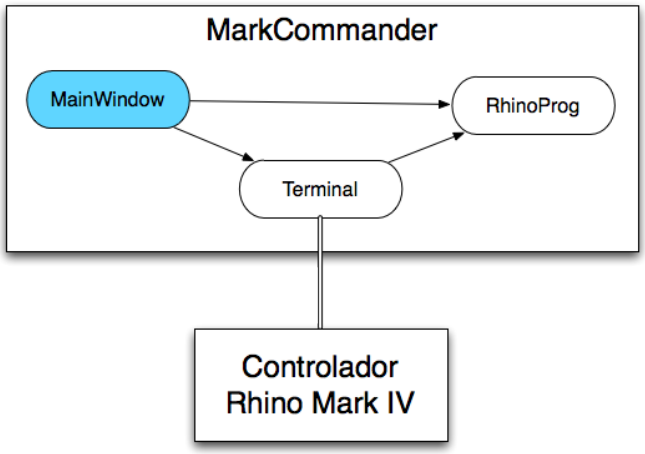
### 4.2.1 Iteración 1: Terminal

La aplicación MarkCommander se desarrollará usando el framework Qt. En este entorno, una aplicación gráfica acostumbra a tener una ventana principal de la clase `MainWindow`. Esta ventana principal será la que actúe como *front-end*, es decir, será la interfaz entre el usuario y el *back-end*. El *back-end* de MarkCommander se fundamentará en la clase `Terminal`, que modelará un terminal de línea de comandos y se encargará de abstraer al usuario de la comunicación con el controlador Mark IV. Además, con vistas a posibilitar el envío y recepción de

[42]La configuración de la UART 1 del GPMCU es igual a la configuración de la UART 1 del MCMCU, puesto que se usan para comunicar ambos microcontroladores el uno con el otro.

programas del controlador, se diseñará la clase `RhinoProg`, que representará un fichero de programa. Esta arquitectura se ilustra en la Figura 4.18.

Figura 4.18: Arquitectura software de MarkCommander

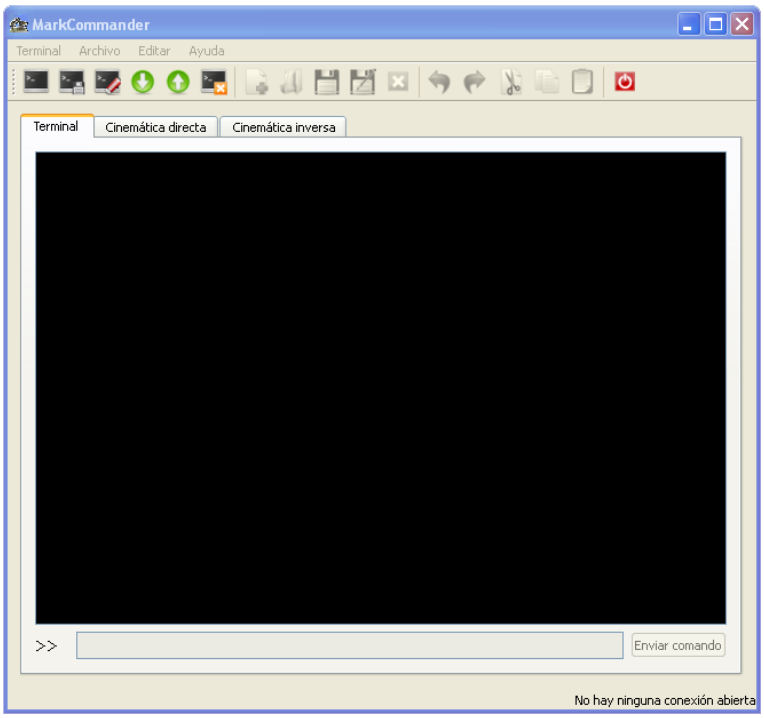



La clase `Terminal`[43] abstrae la comunicación entre el PC y el controlador Mark IV a través del puerto serie, proporcionando las operaciones de comunicación elementales: enviar comando, leer respuesta del comando, recibir programa y transmitir programa. Cuando el usuario abre una conexión con un controlador Mark IV a través de algún puerto serie del sistema, MarkCommander crea un objeto de tipo `Terminal` para conectar con el controlador seleccionado. Entonces `Terminal` maneja la comunicación y, cada vez que el usuario ordena al *front-end* que realice alguna acción, el *front-end* traduce la acción en alguna de las operaciones que permite realizar el `Terminal`.

Para dar soporte a la funcionalidad del terminal, la aplicación dispone de otros módulos auxiliares. El módulo `rhinolang`[44] porporciona definiciones de macros del formato de los comandos y los ficheros (separadores y caracteres especiales) y un mecanismo para comprobar si un comando produce una respuesta por parte del controlador, ya que es necesario saber si un comando produce respuesta, para esperar por dicha respuesta y leerla del buffer del puerto serie. Por su parte, el módulo `errorcodes`[45] proporciona definiciones para manejar los códigos de error del controlador Mark IV y traducir los códigos numéricos a cadenas de texto comprensibles por una persona. Por último, el módulo `exception`[46] proporciona una clase `Exception` para facilitar y homogeneizar el control de excepciones a lo largo de la aplicación.

[43] La clase `Terminal` se encuentra definida en los ficheros `terminal.h` y `terminal.cpp`.
[44] Véase los ficheros `rhinolang.h` y `rhinolang.cpp`.
[45] Véase los ficheros `errorcodes.h` y `errorcodes.cpp`.
[46] Véase los ficheros `exception.h` y `exception.cpp`.

Figura 4.19: Ventana principal de MarkCommander con la vista de `Terminal` activa




### 4.2.2 Iteración 2: Transmisión y recepción de programas

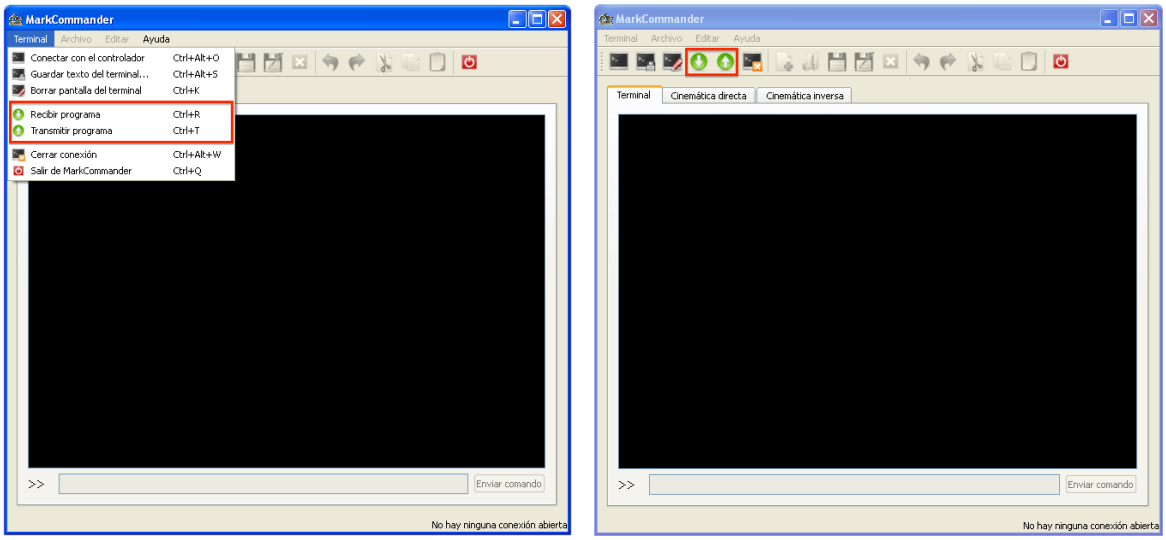
Una vez implementada la funcionalidad de comunicación del terminal, lo cual ya permite enviar comandos al controlador y recibir la respuesta, el siguiente paso es diseñar e implementar la transmisión y recepción de programas del controlador.

Para tal fin, se ha diseñado la clase `RhinoProg`[47], que abstrae el manejo de programas del controlador. Los objetos de tipo `RhinoProg` almacenan el código (comandos) del programa, y proporcionan operaciones para modificar el programa y una operación para guardar el programa en un fichero en disco. Cada vez que se recibe un programa, el *back-end* (`Terminal`) devuelve al *front-end* (`MainWindow`) un objeto `RhinoProg` con el programa recibido, y el *front-end* continuará con la acción especificada por el usuario. Al transmitir un programa, el front-end proporcionará el programa leído de disco en un objeto `RhinoProg` al *back-end*, que se encargará de enviar dicho programa al controlador Mark IV.

Además, en esta iteración se ha debido modificar la interfaz de la ventana principal para incluir e implementar las opciones de menú y los iconos de la barra de herramientas correspondientes a la transmisión y recepción de programas, tal como se ve en la Figura 4.20 (pág. 117).

[47] La clase `RhinoProg` se encuentra definida en `mainwindow.h` y `mainwindow.cpp`.

Figura 4.20: Funcionalidad de envío y recepción de progamas




(a) Opciones en el menú *Archivo*

(b) Botones de la barra de herramientas

### 4.2.3 Iteración 3: Configuración del controlador

Dado que hay ocasiones en las que se desea cambiar la configuración del controlador, es razonable proporcionar esta funcionalidad desde la interfaz gráfica, para que el usuario no tenga que interactuar por medio de comandos con el controlador, puesto que el uso de los comandos de configuración es particularmente complejo y requiere realizar cálculos binarios y a nivel de bit. Por ello, en esta iteración se ha añadido a la ventana principal una opción de menú que despliega una ventana de diálogo desde la que configurar el controlador de forma más fácil e intuitiva (Figura 4.21, pág. 118).

### 4.2.4 Iteración 4: Referencia de comandos y códigos de error

Otra idea para mejorar la usabilidad del terminal y que no es muy costosa de realizar es incluir una referencia del repertorio de instrucciones del controlador y de los códigos de error. Para llevar esto a cabo, bastará con redactar los documentos en HTML y añadir las opciones pertinentes al menú *Ayuda* de la ventana principal, de manera que se pueda abrir las ventanas de ayuda en línea correspondientes (Figura 4.22, pág. 118).

### 4.2.5 Iteración 5: Historial de comandos

Ahora que la funcionalidad fundamental del terminal está completa, pueden surgir diversas ideas para mejorar el manejo del terminal.

La mayor parte de las *shells* de sistemas operativos guardan un historial de los comandos anteriores de manera que, si se pulsa la tecla *Arriba* del teclado, se puede visualizar el último

Figura 4.21: Diálogo de configuración del controlador

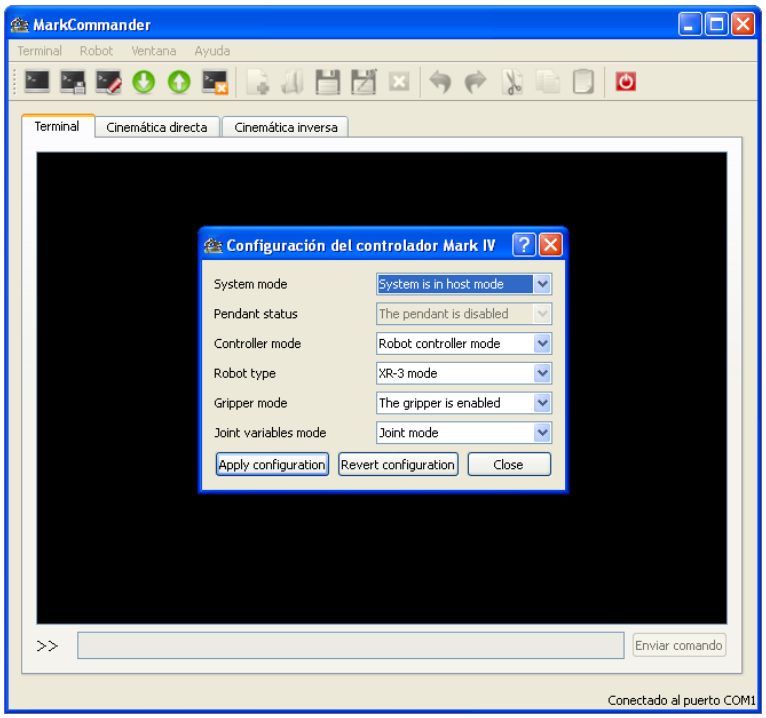
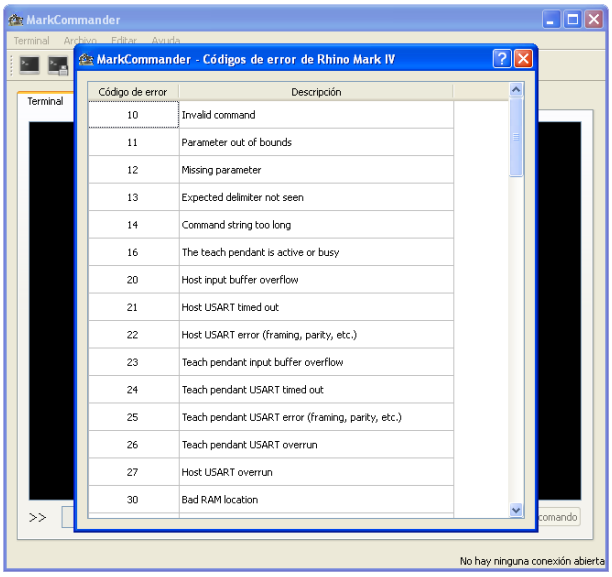
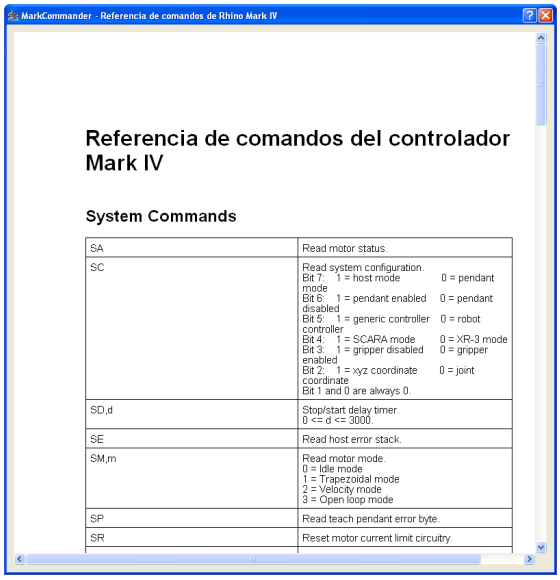



Figura 4.22: Ayuda en línea de MarkCommander




(a) Referencia de códigos de error




(b) Referencia de comandos

comando ejecutado. Si se pulsa dicha tecla sucesivas veces, se puede navegar por todo el historial de comandos de la sesión actual o incluso de sesiones anteriores, y con la tecla *Abajo* se puede hacer lo mismo en sentido opuesto.

Puesto que el terminal de MarkCommander no es más que una *shell* basada en una interfaz gráfica de escritorio, podría resultar muy útil para el usuario proporcionar una funcionalidad similar. El problema es que ningún *widget* del framework Qt proporciona semejante funcionalidad. Por ello, será necesario crear un *widget* basado en `QLineEdit` —el widget que está en uso actualmente para la línea de introducción de comandos— al que se le deberá añadir dicha funcionalidad.

Para tal fin, se ha creado el módulo `cmdlineedit`[48], que define una clase `CmdLineEdit` que hereda de `QLineEdit` y añade al widget `QLineEdit` de Qt la capacidad de mantener un historial de los comandos enviados durante la sesión, para así poder navegar por dicho historial y repetir comandos recientes sin necesidad de escribirlos de nuevo.

### 4.2.6 Iteración 6: Cinemática directa

Una vez diseñada e implementada la funcionalidad del terminal de MarkCommander, otra funcionalidad de gran peso es la resolución de las cinemáticas directa e inversa. Puesto que, por motivos prácticos, este trabajo se restringe al manipulador XR-4 de Rhino Robotics, en esta iteración se implementará sólo la cinemática directa de dicho manipulador.

Para tal fin se plantea el desarrollo de una clase `RhinoXR4` que modele la cinemática del manipulador y que proporcione los métodos necesarios para calcular la cinemática directa. Esta clase se encuentra definida en el módulo `rhino_xr4`, que está formado por los ficheros `rhino_xr4.h` y `rhino_xr4.cpp`. Además, se deberá modificar la interfaz gráfica para soportar el cálculo de la cinemática directa, de lo cual resulta la vista de la Figura 4.23(a) (pág. 120).

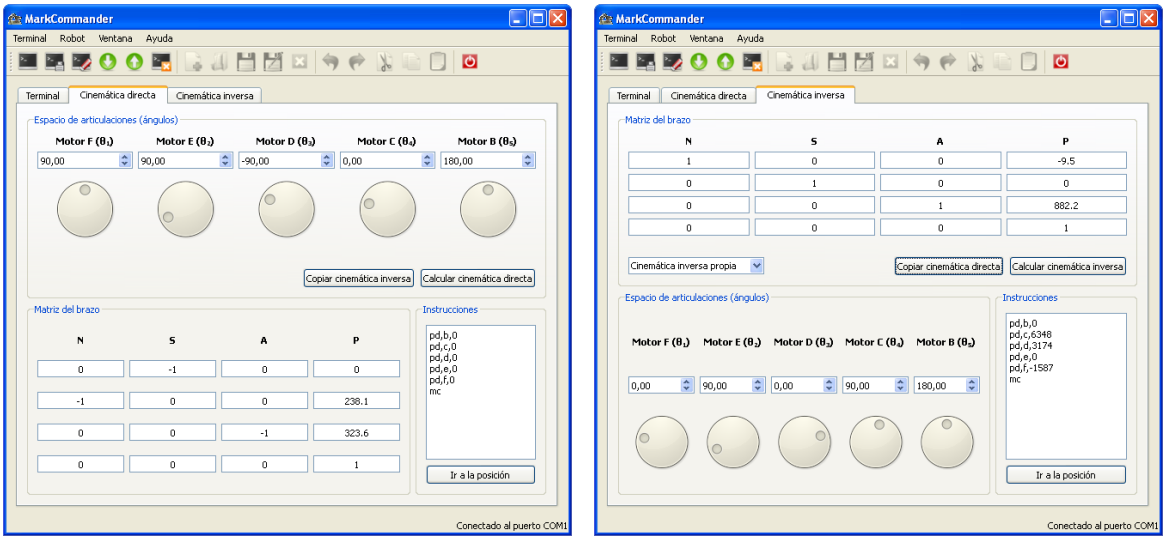
### 4.2.7 Iteración 7: Cinemática inversa

De manera similar a como se realizó en la iteración anterior, en esta nueva iteración se deberá implementar la cinemática inversa del manipulador XR-4, para así completar la funcionalidad fundamental de la interfaz gráfica.

Para ello, podemos basarnos en el diseño realizado en la iteración anterior para la resolución de la cinemática directa y ampliar la clase `RhinoXR4` con las funciones necesarias para el cálculo de la cinemática inversa tal como se describe en el Apéndice B (pág. 135). El resultado es la vista que se muestra en la Figura 4.23(b) (pág. 120).

[48] Véase los ficheros `cmdlineedit.h` y `cmdlineedit.cpp`.

Figura 4.23: Vistas de la interfaz para la resolución de las cinemática directa e inversa




(a) Pestaña para la resolución de la cinemática directa (b) Pestaña para la resolución de la cinemática inversa

### 4.2.8 Resumen del diseño de MarkCommander

En la Tabla 4.11 se presenta un resumen de todos los módulos de la arquitectura software de MarkCommander, indicando la función que realiza cada módulo, y a qué componente de la aplicación pertenece (si al *Terminal*, a la resolución de la *Cinemática directa*, o a la resolución de la *Cinemática inversa*).

Tabla 4.11: Módulos de la arquitectura software de MarkCommander

| Módulo | Componente | Funcionalidad |
|---|---|---|
| `mainwindow` | Terminal | Ventana principal de la aplicación. Maneja los eventos de la interfaz gráfica y se encarga de la interacción con el usuario. Define una clase para representar programas del controlador y resuelve la cinemática del manipulador haciendo uso del modelo del robot XR-4 definido en `rhino_xr4`. |
| `terminal` | Terminal | Abstrae la comunicación con el controlador mediante el puerto serie, proporcionando un terminal de línea de comandos con funciones para el envío de comandos, la recepción de respuestas, y la transmisión y recepción de programas del controlador. |
| `rhinolang` | Terminal | Proporciona las definiciones del lenguaje de comandos del controlador necesarias para el envío de comandos con el formato correcto. |

| Módulo | Componente | Funcionalidad |
|---|---|---|
| `rhinoprog` | Terminal | Abstrae el manejo de programas del controlador proporcionando una clase que permite leer y escribir programas en ficheros de texto. |
| `errorcodes` | Terminal | Proporciona las definiciones de los códigos de error del controlador y asocia el código numérico de error con un mensaje descriptivo. |
| `exception` | Terminal | Proporciona un modelo de excepciones para facilitar el manejo de condiciones de error del controlador. |
| `cmdlineedit` | Terminal | Extiende el widget `QLineEdit` de Qt para que guarde un historial de los comandos enviados durante la sesión. |
| `matrix` | Cinemática directa, Cinemática inversa | Define estructuras de datos y operaciones con matrices para la resolución de la cinemática. |
| `rhino_xr4` | Cinemática directa, Cinemática inversa | Define un modelo del manipulador XR-4 que permite resolver su cinemática directa e inversa. |

# Referencias


[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2 edition, 2007.

[2] E. S. TEZ. Interfacing Bi-Phase Incremental Encoders. *IEEE Transactions on Industrial Electronics*, 33(3):337–338, August 1986.

[3] Bernard Hébert, Michel Brûlé, and Louise-A. Dessaint. A high efficiency interface for a biphase incremental encoder with error detection (servomotor control). *IEEE Transactions on Industrial Electronics*, 40(1):155–156, 1993.

[4] Agustín Cruz Contreras, Edgar A. Portilla Flores, and Ramón Silva Ortigoza. Multiplicador Electrónico para Encoder Incremental.

[5] Katsuhiko Ogata. *Ingeniería de Control Moderna*. Pearson Educación, 4 edition, 2003.

[6] Arthur G. O. Mutambara. *Design and Analysis of Control Systems*. CRC Press, 1999.

[7] Stephen Bowling. *Application Note 696: PIC18CXXX/16CXXX DC Servomotor Application*.

[8] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robótica: Control, Detección, Visión e Inteligencia*. McGraw-Hill, 1 edition, 1988.

[9] dsPIC30F Family Reference Manual. http://is.gd/CQ0eLy, .

[10] dsPIC30F Data Sheet General Purpose and Sensor Family. http://is.gd/fwnaUM, .

[11] dsPIC30F Data Sheet Motor Control and Power Conversion Family. http://is.gd/aaN4xR, .

[12] dsPIC30F3014/4013 Data Sheet. http://is.gd/r1kgHl, .

[13] dsPIC30F4011/4012 Data Sheet. http://is.gd/hBsQPY, .

[14] *The TTL Data Book for Design Engineers*, volume 1. Texas Instruments, 1984.

[15] *TeleVideo 9220 Video Display Terminal Operator's Manual*. TeleVideo Systems, Inc., 1985.

[16] Gustavo Adolfo Alarcón Gacitúa, Daniel Adrián Fernando Lajo Carpio, and Marcelo Quispe Ccachuco. Diseño de un Controlador PID con Interfaz Gráfica de Control para un Sistema de Equilibrio Bi-hélice. *11th Latin American and Caribbean Conference for Engineering and Technology*, 2013.

# 5 Conclusiones y trabajo futuro

## 5.1 Valoración y conclusiones

La robótica en general y la robótica industrial en particular se han convertido a día de hoy en el presente y el futuro de la sociedad industrializada. La robótica industrial es ya algo sin lo que la industria no conseguiría alcanzar los niveles de producción y los bajos costes posibles en la actualidad gracias a la automatización blanda, con lo que supone un motor para el avance de la industria y el crecimiento de las empresas. Por su parte, la robótica en general es un área de investigación muy importante que busca activamente la creación de nuevas soluciones para mejorar la calidad de vida de las personas. Si bien la robótica en general abarca más tipos de robots que la robótica industrial, restringida en gran parte a los manipuladores robóticos, los conocimientos de la robótica industrial son fundamentales en el diseño de otros robots no industriales. Ejemplo de ello son robots de servicio tipo rover[1] o robots humanoides[2], que suelen requerir manipuladores robóticos en su diseño, así como el uso de diversos sensores y técnicas de visión artificial también empleados en la robótica industrial.

Por estos motivos, la enseñanza de la robótica en los centros de educación superior es esencial para formar futuros profesionales que puedan desempeñar una labor de desarrollo e investigación en estas áreas, ya que la industria continuará demandando profesionales preparados. Para que esta formación sea lo más completa posible, lo ideal es incluir un programa de enseñanza práctica de la robótica utilizando robots educativos. De entre los robots educativos disponibles en el mercado, los sistemas robóticos de Rhino Robotics siguen gozando de gran aceptación, a pesar de su antigüedad. A su favor juegan especialmente su alta programabilidad —ya que son pocos los robots educativos pensados para la enseñanza de la robótica industrial que permiten el desarrollo de programas con comandos de texto— y el fácil manejo del sistema por medio de la paleta de programación.

[1] Véase, por ejemplo, el concepto de rover de la Agencia Espacial Europea: http://youtu.be/9qkC4keOAbA.

[2] Por ejemplo, Robonaut: http://youtu.be/vfDXzkFHnz0, http://youtu.be/2YYr9wp9hSM, REEM-C: http://youtu.be/4HZIDpNSKyc, Justin: http://youtu.be/R6pPwP3s7s4, http://youtu.be/9x_c4Dwdegc.

Dado que, como se ha mencionado, los sistemas robóticos de Rhino Robotics son bastante antiguos —datan de finales de los años 80—, difícilmente se podrán conseguir en el mercado, ya sea que se busque sistemas nuevos o repuestos para reparar sistemas averiados. Asimismo, el software que se distribuía con los sistemas está totalmente desfasado.
Con el fin de solventar estos problemas, se ha diseñado y desarrollado un prototipo de un nuevo sistema hardware/software para reemplazar los sistemas antiguos. Por medio de este desarrollo se ha sentado las bases para un controlador robótico retrocompatible con el controlador Mark IV, y se ha creado una utilidad software moderna para su manejo. Los siguientes pasos en el desarrollo de este trabajo serían, por tanto, continuar con las mejoras del prototipo de controlador y, eventualmente, diseñar brazos robóticos compatibles con los robots Rhino, para así reemplazar los sistemas antiguos.
Además, para garantizar la disponibilidad a largo plazo de este nuevo sistema, lo ideal sería liberar las especificaciones y el código bajo licencias libres, para fomentar el crecimiento de una comunidad de desarrolladores y usuarios que den soporte y alarguen la vida del sistema, de manera que no se estanque o quede desfasado en unos pocos años, como ha sucedido con los sistemas de Rhino Robotics y como sucederá, en general, con cualquier sistema comercial a largo plazo. Asimismo, el aprovechamiento de nuevas tecnologías, como la impresión 3D, podría favorecer aun más la aceptación del sistema, debido a que permitiría la reducción de costes y una mayor adaptabilidad del sistema a las necesidades individuales de cada usuario.

A pesar de que el presente trabajo se ha ocupado sólo del desarrollo de un primer prototipo, este prototipo ya proporciona las funcionalidades básicas para su uso en gran parte de los ejercicios docentes de iniciación a la robótica industrial. Concretamente, el prototipo desarrollado admite órdenes enviadas desde un PC por medio del puerto serie y que permiten controlar los movimientos del robot en el espacio de las articulaciones, para lo cual este prototipo implementa el control realimentado de la posición y movimientos con perfil de velocidad trapezoidal. Además, la utilidad software desarrollada permite la resolución de las cinemáticas directa e inversa del brazo robótico XR-4.

En el aspecto técnico, se ha escogido implementar el prototipo usando para ello microcontroladores de la familia dsPIC de Microchip, inicialmente porque era el único material disponible, pero también por interés académico y profesional en dicha plataforma, y porque, tras compararlos con otras plataformas, se ha visto que representan la solución más equilibrada para los requisitos de la aplicación en cuestión, a pesar del elevado coste de las herramientas de desarrollo.
Adicionalmente, ha sido necesario diseñar una placa para el control de motores adaptada a las necesidades de la aplicación, puesto que, entre la oferta de placas de control de motores para las plataformas de prototipado presentes en el mercado, no existe ninguna placa con las características necesarias para la aplicación. Este diseño de hardware se ha basado en el driver de motores L298, que proporciona dos puentes en H que pueden ser controlados mediantes sus respectivas señales de PWM y selección de sentido de giro del motor, y se ha planteado la inclusión de un circuito de protección contra sobreintensidad, así como el aislamiento óptico de los microcontroladores, para así proteger el sistema ante fallos eléctricos.

Desde el punto de vista académico, el diseño de este sistema aporta una experiencia y conocimientos considerable en las áreas de

1. diseño de sistemas embebidos e integración hardware/software,
2. arquitectura y programación de microcontroladores,
3. compiladores e intérpretes,
4. estudio cinemático de manipuladores robóticos,
5. sistemas de control,
6. técnicas de control de motores (y la electrónica correspondiente),
7. diseño digital, y
8. ensamblado de electrónica.

Por ello, la publicación de las especificaciones, los ficheros de diseño y el código fuente proporciona otra ventaja adicional a las ya comentadas, en este caso académica, y es que posibilitaría no sólo el aprendizaje y ejercitación de estos conocimientos por parte del diseñador y desarrollador, sino también del usuario final que desea comprender cómo funciona el sistema o que desea aprender de las áreas de conocimiento mencionadas por medio del estudio de las soluciones adoptadas para la resolución de los problemas de diseño que una aplicación de estas características plantea. Esto posibilita el uso del sistema no sólo como herramienta para la enseñanza práctica de la programación y el control de robots industriales, sino también como ejemplo de caso de uso de diferentes técnicas de ingeniería de control, electrónica e informática. Finalmente, una última posibilidad para su uso en la enseñanza y el aprendizaje es tomar este sistema como base para otros desarrollos por medio de los cuales poner en práctica técnicas de otras áreas de conocimiento, lo cual nos plantea algunas opciones de trabajo futuro.

## 5.2 Trabajo futuro

Dado que en el presente trabajo se ha desarrollado sólo el primer prototipo del sistema, quedan por desarrollar algunas funcionalidades presentes en el sistema antiguo (controlador Mark IV) y también existen diversas opciones de ampliación más allá de las funcionalidades fundamentales. Además, algunas de estas posibilidades de ampliación involucran actividades de otras áreas de conocimiento, lo cual posibilita la realización de otros proyectos y el uso del sistema desarrollado como base para la enseñanza y el aprendizaje de estas otras áreas.

En primer lugar surgen algunas propuestas de ampliación y mejora del controlador RhinoChip, tanto en hardware como en software. Estas propuestas incluyen:

- Optimizar las estructuras de datos (fundamentalmente buffers) e implementar una pila de errores.
- Completar el repertorio de instrucciones del controlador.

- Implementar más modos de movimiento (movimiento coordinado, movimiento cartesiano).
- Realizar un estudio dinámico del sistema robótico y calcular las ganancias óptimas para el bucle de control PID.
- Implementar un módulo de E/S para el controlador.
- Realizar el diseño del sistema final con una placa de circuito impreso y una fuente de alimentación, para integrarlo todo dentro de una carcasa y poder hacer un montaje del sistema en producción.

Además, también la utilidad software MarkCommander admite la ampliación de sus funcionalidades con tareas como:

- Añadir soporte para el robot Rhino SCARA y, en general, para otros robots.
- Añadir un editor de programas con resaltado de sintaxis.
- Implementar una interfaz de control del robot en tiempo real (similar a lo que se haría por medio de una paleta de programación, pero desde el software para PC).

Finalmente, la mejora del sistema puede conducir al surgimiento de otros proyectos, algunos de ellos en otras disciplinas. Por ejemplo:

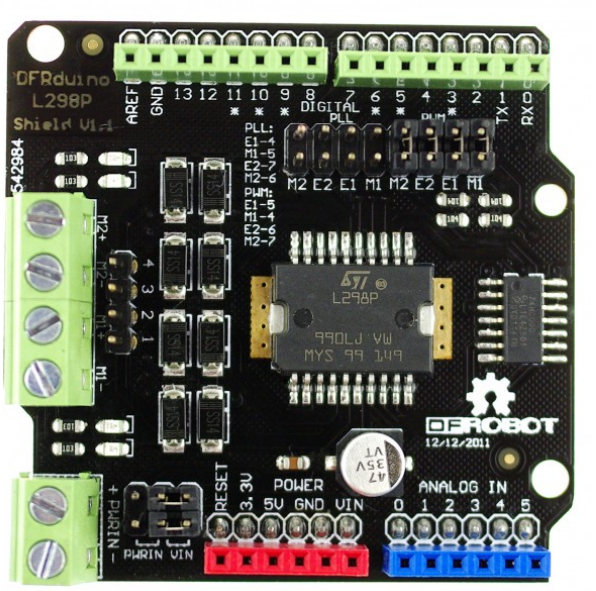
- Diseñar e implementar una paleta de programación (como dispositivo físico, no como software para PC).
- Diseñar un lenguaje de programación de alto nivel para el robot.
- Diseñar y desarrollar un simulador programable para PC, para así simular el funcionamiento de los robots antes de programarlos.
- Diseñar nuevos brazos robóticos, ya sea inspirados en los brazos de Rhino Robotics o nuevos diseños.

# A

# Diseño de una placa para el control de motores

El enfoque más común para el control de motores de corriente continua, y el planteado en este proyecto, es el uso de modulación por ancho de pulso (PWM) y un puente en H.
Al inicio del proyecto, se planteó utilizar la placa Arduino Motor Shield de DFRduino con el chip L298P[1] (Figura A.1), que proporciona todo lo necesario para controlar dos motores de corriente continua mediante PWM. Sin embargo, esta placa tiene una carencia fundamental, y es que no proporciona circuitería de protección contra sobreintensidad. Esto implica que, si el motor se bloquea, la intensidad que circula por el circuito de control de motores aumentará drásticamente, pudiendo quemar el circuito y dejarlo inservible. De hecho, esto sucedió en una ocasión durante el desarrollo del proyecto.

Figura A.1: Arduino Motor Shield de DFRduino




En su funcionamiento normal, la placa de control de motores de DFRduino alimentaba el

[1] http://is.gd/pQDR3Z

motor y éste giraba en el sentido especificado, consumiendo del orden de $0.5\,\mathrm{A}$, según indicaba el medidor de intensidad de la fuente de alimentación del montaje experimental realizado en el laboratorio. Sin embargo, en una ocasión, durante ciertas pruebas, el motor se bloqueó por un fallo mecánico, dejó de girar, y la intensidad de corriente que circulaba por el circuito superó los $1.5\,\mathrm{A}$ durante un tiempo prolongado. La intensidad máxima soportada por el chip L298P en el que se basa la placa de DFRduino es de $3\,\mathrm{A}$ en pico y $2\,\mathrm{A}$ en funcionamiento continuo, de manera que el chip terminó quemándose y la placa quedó inutilizable.

Reemplazar la placa no es una opción aceptable, por dos motivos:

1. Si bien se dispone de más unidades de la misma placa, al no disponer de un protector de sobreintensidad, el mismo incidente podría volver a suceder, lo cual repercutiría negativamente en la calidad y el coste del sistema.
2. En el área de la robótica industrial no se puede permitir la existencia del riesgo de accidentes, ya sea para el robot, debido al coste del sistema, o para su entorno y cualquier operario que pudiera trabajar en él. Además, según cómo sea el entorno de trabajo, las colisiones del brazo robótico con objetos de su entorno pueden ser frecuentes. Por todo esto, un mecanismo para detectar colisiones[2] y proteger el sistema robótico es fundamental en el diseño del sistema.

Por tanto, la única opción era diseñar una placa de control de motores propia. Se decidió basar el diseño de la nueva placa en el diseño de la placa DFRduino, añadiendo las siguientes características:

1. **Protección contra sobreintensidad:** Aprovechando las características del chip L298[3], se dotará a la placa de un circuito de detección de sobreintensidad, de manera que una señal de alarma se active cuando alguno de los motores se bloquee o, en general, se detecte que la intensidad de corriente que alimenta un motor supera un umbral de seguridad preestablecido.
2. **Optoaislamiento:** Para incrementar la seguridad de la electrónica del controlador y, en concreto, de los microcontroladores, se plantea aislar las señales de control mediante optoacopladores. De esta manera, si ocurriese algún fallo eléctrico que inutilizase la placa de motores, se evitaría que este fallo afectase al resto de la electrónica del controlador.
3. **Alimentación externa:** Puesto que se plantea aislar la electrónica de los microcontroladores de la electrónica de control de motores, ello requiere también que la alimentación de la lógica de control de motores se tome de una fuente de alimentación independiente (hasta ahora se tomaban los $5\,\mathrm{V}$ y la tierra de la placa de microcontroladores). Por ello, se plantea que, si la tensión de alimentación de los motores es suficientemente elevada, se obtenga la alimentación de la lógica a partir de esta fuente mediante un regulador de tensión o, en caso de que la tensión de alimentación no sea suficiente, que se pueda conectar una fuente de alimentación externa.

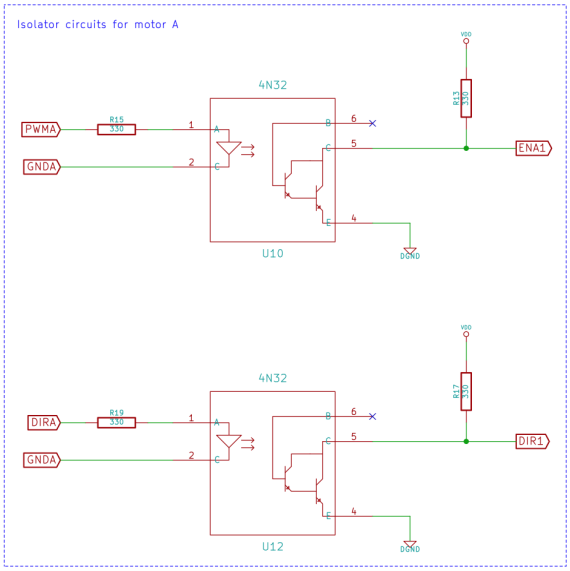
[2] Una colisión producirá que por el motor circule una intensidad de corriente elevada, por lo que la colisión se puede detectar mediante un circuito detector de sobreintensidad.

[3] Este chip en todas sus variantes permite añadir dos resistencias externas para la medición de la intensidad, una para cada motor.

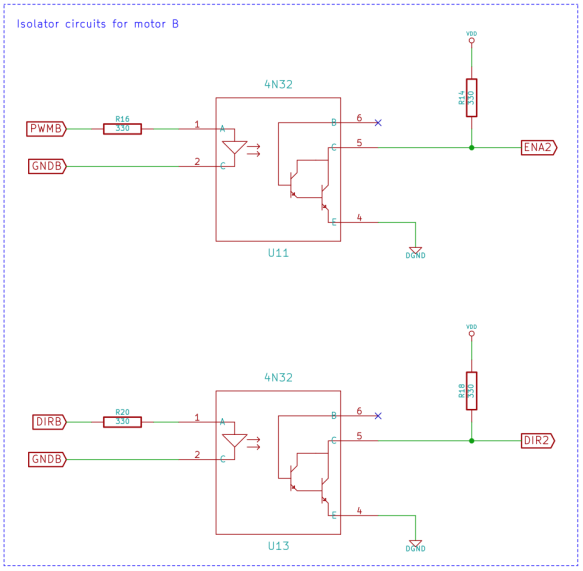
## A.1 Señales de control y diseño del circuito optoaislador

Se plantea como objetivo que la placa diseñada sea capaz de controlar dos motores mediante una señal de PWM y una señal de dirección (selección del sentido de giro). Además, puesto que dichas señales deben estar optoacopladas, deberá proporcionarse una entrada de tierra para cada motor. Finalmente, por versatilidad, se propone añadir una señal de salida que indique si se ha producido una parada de emergencia del motor debida a un evento de sobreintensidad. Puesto que esta última señal será de salida, se requerirán dos entradas de $V_{DD}$ y tierra, respectivamente. Los circuitos optoaisladores se muestran en la Figura A.2.

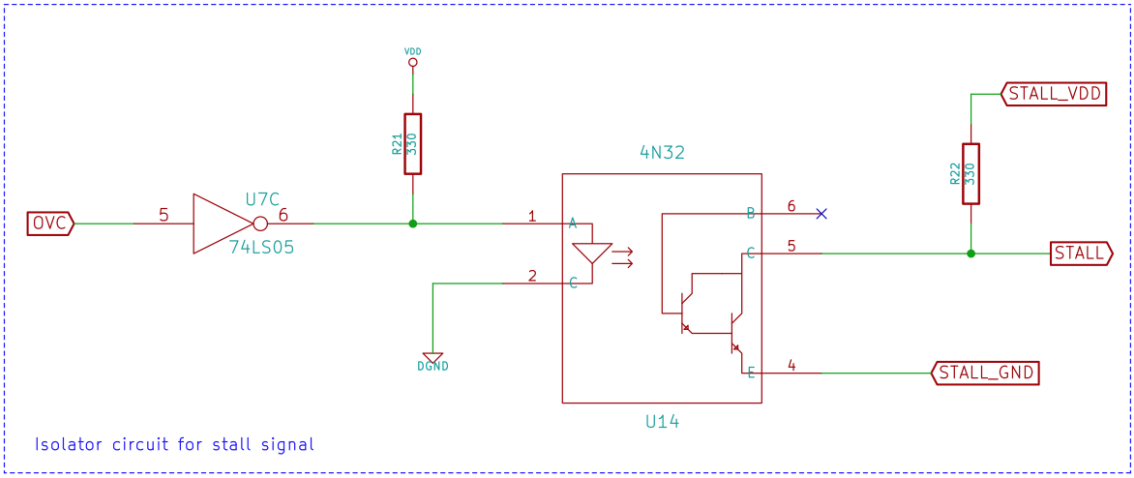
Figura A.2: Circuitos optoaisladores




(a) Optoaisladores para las señales del motor A




(b) Optoaisladores para las señales del motor B

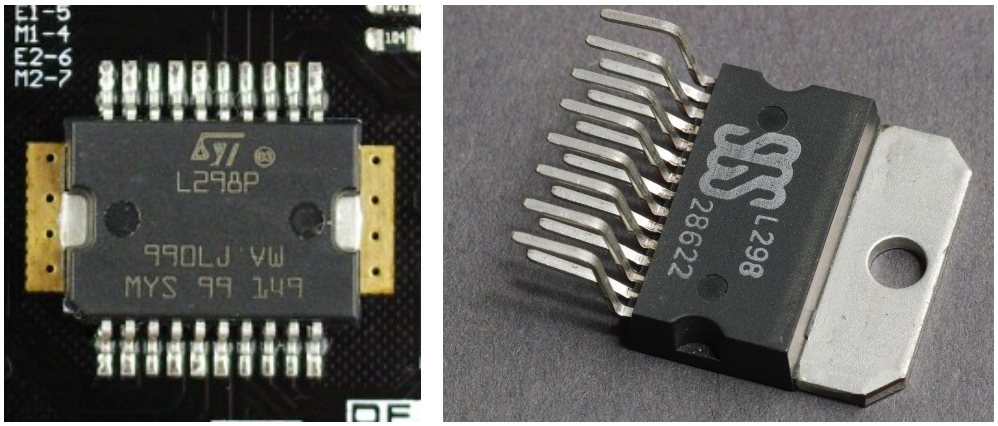



(c) Optoaislador de la señal de parada

## A.2 Mejora del circuito de control de motores con protección contra sobreintensidad

Uno de los cambios introducidos al diseño original de la placa DFRduino es la utilización del chip L298N (variante THT[4] vertical) en vez del L298P (variante SMD[5]) o el L298HN (variante THT horizontal), ya que tiene una mayor superficie de disipación con una placa metálica que permite la instalación de un disipador más grande, lo cual ayudará a refrigerar el chip, evitando que se caliente tanto como el L298P sin disipador que está montado en la placa DFRduino.

Figura A.3: Circuito integrado L298




(a) L298P (variante SMD) (b) L298N (variante THT)

Para construir un circuito detector de sobreintensidad se aprovechará las salidas *SENSA* y *SENSB* del chip, que permiten conectar una *resistencia de sensado* para medir la intensidad que circula por el circuito. La resistencia debe ser baja, y se recomienda de unos $0.5\,\Omega$. Con esta *resistencia de sensado*, deseamos limitar la corriente que circula por el circuito a $2\,\mathrm{A}$[6]. En estas circunstancias, la caída de tensión a extremos de la resistencia será

$$V = IR = (2\,\mathrm{A}) \times (0.5\,\Omega) = 1\,\mathrm{V}. \tag{A.1}$$

Por tanto, podemos utilizar un comparador con un voltaje de referencia de $1\,\mathrm{V}$ para detectar cuándo la intensidad supera los $2\,\mathrm{A}$ y, en tal caso, generar una señal de alarma que permita tomar medidas para proteger el circuito. Para generar dicho voltaje de referencia utilizaremos un divisor de tensión con dos resistencias de $4\,\mathrm{k}\Omega$ y $1\,\mathrm{k}\Omega$, respectivamente, y una tensión de alimentación de $5\,\mathrm{V}$. Entonces introduciremos ambos voltajes en un comparador LM319 que, a

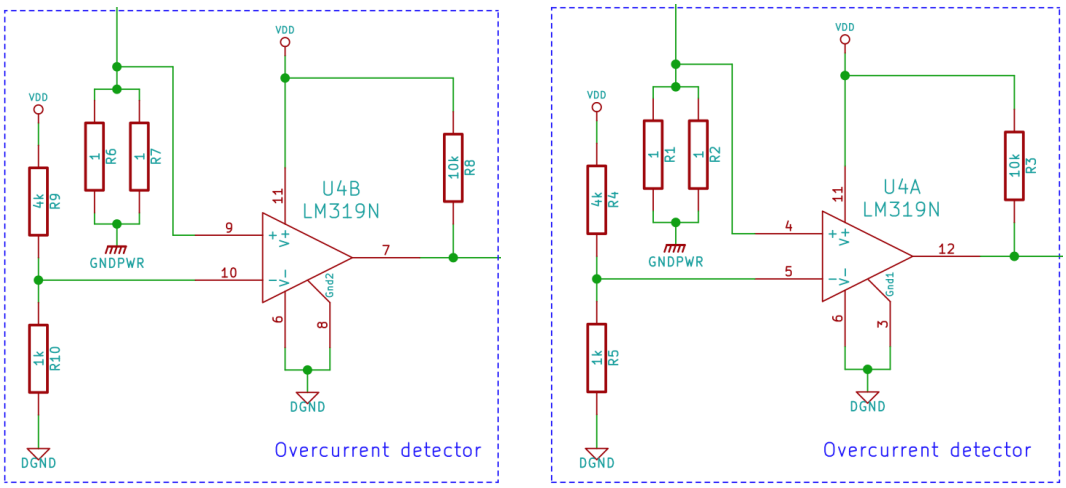
[4]THT, siglas de *Through-Hole Technology*, es una tecnología de montaje de circuitos integrados que consiste en que los chips tienen unos pines que se insertan en unos agujeros en la placa de circuito impreso y se sueldan.

[5]SMD, siglas de *Surface Mount Device*, corresponde a una tecnología de montaje de circuitos integrados que consiste en que los chips se disponen sobre la placa y se sueldan sin necesidad de agujeros; de ahí el nombre "montaje superficial".

[6]Puesto que la potencia disipada será $P = RI^2 = (0.5\,\Omega) \times (2\,\mathrm{A})^2 = 2\,\mathrm{W}$, se deberá usar resistencias que soporten hasta $2\,\mathrm{W}$. Como no disponemos de resistencias tan bajas capaces de soportar tanta potencia, usaremos dos resistencias de $1\,\Omega$ en paralelo.

su salida, producirá una señal activa a nivel alto cuando se detecte sobreintensidad (Figura A.4).

Figura A.4: Circuitos detectores de sobreintensidad

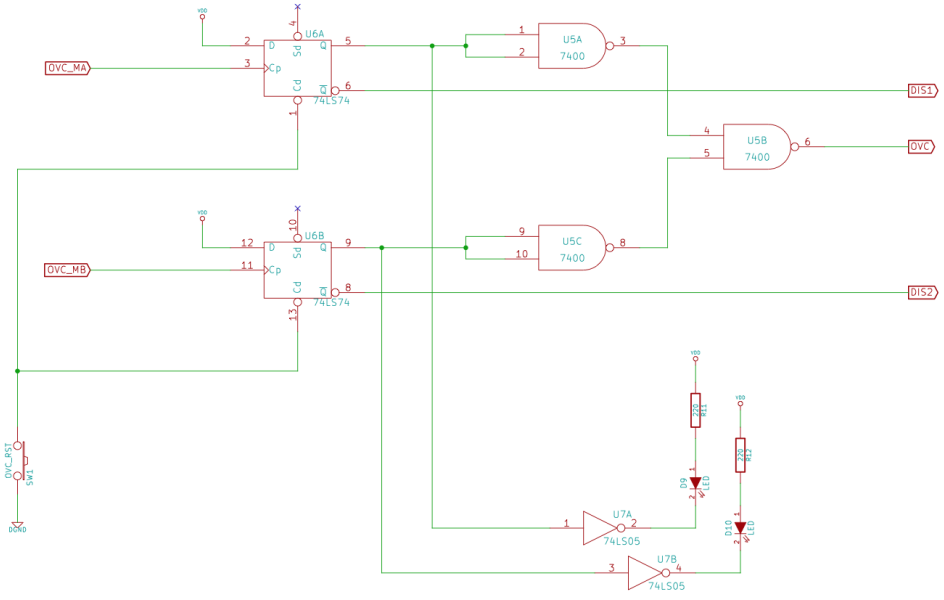



(a) Detector de sobreintensidad para el motor A

(b) Detector de sobreintensidad para el motor B

La salida de los detectores de sobreintensidad va entonces a unos flip-flops y luego se combinan (mediante una operación OR) para detectar si alguno de los dos motores se ha bloqueado. Esta señal será la que se lleve al optoacoplador de la Figura A.2(c) (pág. 129). Además, una de las salidas del flip-flop inhibirá la señal de PWM del motor bloqueado y se encenderá un indicador luminoso por cada motor bloqueado. Para poder habilitar nuevamente el motor bloqueado, se incluirá un botón de reinicio que reiniciará el estado de los flip-flops, habilitando de nuevo la entrada de PWM que activa los motores. Este circuito se presenta en la Figura A.5.

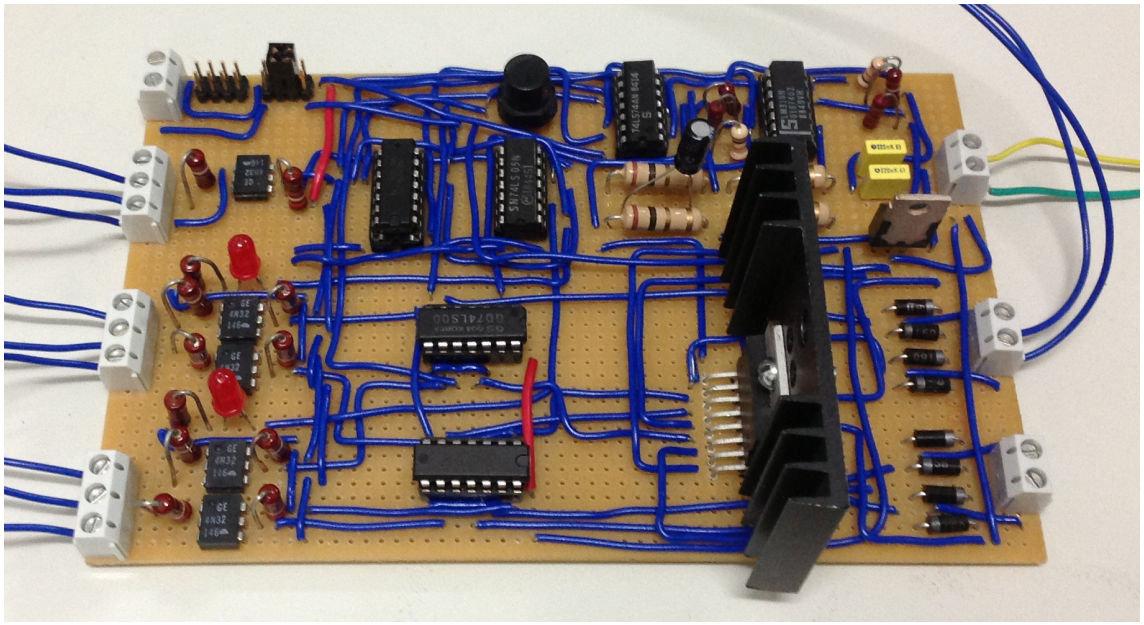
Figura A.5: Circuito protector de sobreintensidad



## A.3 Montaje y verificación del prototipo

El montaje se ha realizado fundamentalmente en una *perfboard*, si bien algunos componentes han sido probados primeramente en una protoboard, antes de ser soldados a la placa. En la Figura A.6 se puede ver el aspecto del prototipo de la placa después de la verificación y modificación de la revisión 1.3 para dar lugar a la 1.4. Esto es así porque en la revisión 1.3, que es la primera versión implementada del diseño, se encontraron ciertos fallos.

Figura A.6: Prototipo de la placa de control de motores (revisión 1.4)



En primer lugar, en la fase de diseño de la placa se olvidó negar las señales de PWM y dirección procedentes de los optoacopladores, o modificar la lógica que las procesa antes de introducirlas en el L298, por lo que las señales de PWM y dirección son ahora activas a nivel bajo. Esto realmente no supone un gran problema, ya que se puede solucionar por software, y así es como se ha hecho.

En cambio, el otro problema que se ha encontrado es que el circuito de protección de la revisión 1.3 no funciona adecuadamente. En concreto, el circuito protector deshabilita los motores en situaciones en las que no debería, como, por ejemplo, cuando se pone en marcha o se para los motores. Tras diagnosticar el problema, se ha llegado a la conclusión de que esto ocurre porque el circuito detecta sobreintensidad cuando se produce algún pico de intensidad instantáneo, que es lo que sucede cuando se pone en marcha o se para el motor.
Este problema es más complicado de resolver que el anterior, puesto que requiere cierto filtrado de las señales. En una primera aproximación, se ha instalado un condensador electrolítico en paralelo con cada *resistencia de sensado* para filtrar los picos de intensidad. Durante un tiempo esta aproximación funcionó en todas o casi todas las pruebas, pero, llegado un punto, dejó de funcionar correctamente, por lo que no parece ser una solución robusta a largo plazo. Las otras dos opciones son o bien utilizar un filtro RC para integrar la señal y filtrar los picos de intensi-

dad totalmente, o bien prescindir de los flip-flops y así conseguir que los motores se desactiven sólo mientras continúe produciéndose el evento de sobreintensidad. A pesar de ello, ninguno de estos dos enfoques se ha probado, ya que cualquiera de ellos requeriría más pruebas, y desoldar y volver a soldar componentes, lo cual alargaría la duración del proyecto más de lo establecido. En cualquier caso, siempre es posible desactivar el circuito protector si se extrae el chip 7474 que contiene los flip-flops. Esto permitirá que se siga pudiendo utilizar la placa, sólo que no dispondrá de protección contra sobreintensidad. Ésta ha sido finalmente la solución escogida, para poder continuar el desarrollo del proyecto sin retrasos, y se ha dejado la resolución de estas cuestiones para futuras revisiones de la placa. Sin embargo, para no dañar la placa mientras continúan los experimentos, ésta deberá ser utilizada con una fuente de alimentación que disponga de limitador de intensidad, para así prevenir que la circuitería se pueda quemar debido a que el circuito protector de la placa se encuentra desactivado.

# Referencias


[1] *The TTL Data Book for Design Engineers*, volume 1. Texas Instruments, 1984.

[2] *Application Note 1332: Current Sensing Circuits Concepts and Fundamentals*.

[3] *Application Note: Using the L6506 for Current Control*. ST Microelectronics, .

[4] Lelio Soares and Victor Casanova Alcalde. An Educational Robotic Workstation based on the Rhino XR4 robot. *Proceedings. Frontiers in Education. 36th Annual Conference*, pages 7–12, 2006.

[5] Victor Casanova Alcalde and Lelio Soares. A laboratory platform for teaching computer vision control of robotic systems. *2008 38th Annual Frontiers in Education Conference*.

[6] *SB120-SB160 Schottky Barrier Chip*. Diodes Incorporated.

[7] *L298 Dual Full-bridge Driver*. ST Microelectronics, .

[8] *L6506 Current Controller for Stepping Motors*. ST Microelectronics, .

[9] *LM319 Dual Comparator*. Fairchild Semiconductor, .

[10] *LM78XX / LM78XXA 3-Terminal 1 A Positive Voltage Regulator*. Fairchild Semiconductor, .

[11] Carla Silva Rocha Aguiar. Projeto e Implementação de uma nova Arquitetura de Controle para o Robô Rhino XR-4. Master's thesis, Universidade de Brasília, 2004.

[12] *SB120 thru SB160 Schottky Barrier Rectifier*. Vishay General Semiconductor.

[13] Stephen Bowling. *Application Note 696: PIC18CXXX/16CXXX DC Servomotor Application*.

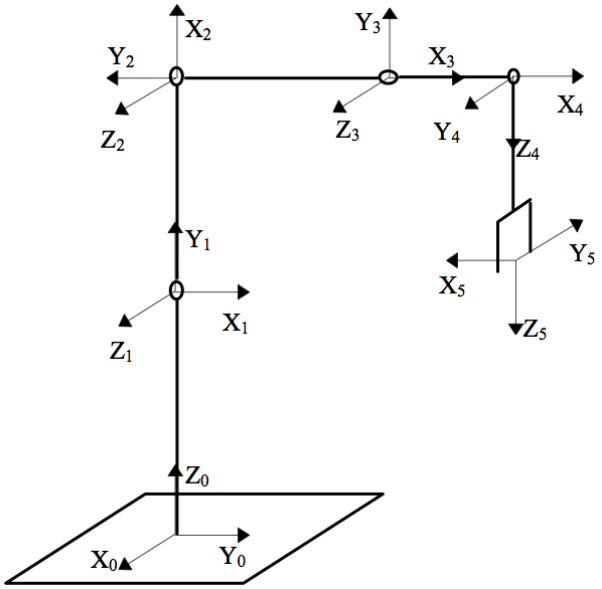
# B Cinemática del manipulador XR-4

## B.1 Cinemática directa

### B.1.1 Resolución de la cinemática directa

El esquema del manipulador robótico con la asignación de sistemas de coordenadas para las articulaciones se puede ver en la figura Figura B.1. A partir de esta figura se extrae los

Figura B.1: Asignación de sistemas de coordenadas para la resolución de la cinemática del robot Rhino XR-4




parámetros de la Tabla B.1 (pág. 136).

La matriz de transformación del sistema de coordenadas $\{S_i\}$ al sistema de coordenadas

Tabla B.1: Parámetros cinemáticos del robot Rhino XR-4

| Articulación $i$ | $a_i$ | $\alpha_i$ | $d_i$ | $\theta_i$ | HOME |
|---|---|---|---|---|---|
| 1 | 0 | 90º | 260 | ${\theta_1}^*$ | ${\theta_1}^* = 90º$ |
| 2 | 228.6 | 0º | 0 | ${\theta_2}^*$ | ${\theta_2}^* = 90º$ |
| 3 | 228.6 | 0º | 0 | ${\theta_3}^*$ | ${\theta_3}^* = -90º$ |
| 4 | 9.5 | 90º | 0 | ${\theta_4}^*$ | ${\theta_4}^* = 0º$ |
| 5 | 0 | 0º | 165 | ${\theta_5}^*$ | ${\theta_5}^* = 180º$ |

$\{S_{i-1}\}$ es de la forma

$$\mathbf{A}_{i-1}^{i} = \begin{bmatrix} \mathrm{c}\left(\theta_i\right) & -\mathrm{s}\left(\theta_i\right)\mathrm{c}\left(\alpha_i\right) & \mathrm{s}\left(\theta_i\right)\mathrm{s}\left(\alpha_i\right) & a_i\mathrm{c}\left(\theta_i\right) \\ \mathrm{s}\left(\theta_i\right) & \mathrm{c}\left(\theta_i\right)\mathrm{c}\left(\alpha_i\right) & -\mathrm{c}\left(\theta_i\right)\mathrm{s}\left(\alpha_i\right) & a_i\mathrm{s}\left(\theta_i\right) \\ 0 & \mathrm{s}\left(\alpha_i\right) & \mathrm{c}\left(\alpha_i\right) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

donde $\mathrm{c}\left(\alpha\right) = \cos\left(\alpha\right)$ y $\mathrm{s}\left(\alpha\right) = \mathrm{sen}\left(\alpha\right)$.

Utilizando los datos de la Tabla B.1, llegamos a las siguientes matrices:

$$\mathbf{A}_0^1 = \begin{bmatrix} \mathrm{c}_1 & 0 & \mathrm{s}_1 & 0 \\ \mathrm{s}_1 & 0 & -\mathrm{c}_1 & 0 \\ 0 & 1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{c}_1 & 0 & \mathrm{s}_1 & 0 \\ \mathrm{s}_1 & 0 & -\mathrm{c}_1 & 0 \\ 0 & 1 & 0 & 260 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{A}_1^2 = \begin{bmatrix} \mathrm{c}_2 & -\mathrm{s}_2 & 0 & a_2\mathrm{c}_2 \\ \mathrm{s}_2 & \mathrm{c}_2 & 0 & a_2\mathrm{s}_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{c}_2 & -\mathrm{s}_2 & 0 & 228.6 \cdot \mathrm{c}_2 \\ \mathrm{s}_2 & \mathrm{c}_2 & 0 & 228.6 \cdot \mathrm{s}_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{A}_2^3 = \begin{bmatrix} \mathrm{c}_3 & -\mathrm{s}_3 & 0 & a_3\mathrm{c}_3 \\ \mathrm{s}_3 & \mathrm{c}_3 & 0 & a_3\mathrm{s}_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{c}_3 & -\mathrm{s}_3 & 0 & 228.6 \cdot \mathrm{c}_3 \\ \mathrm{s}_3 & \mathrm{c}_3 & 0 & 228.6 \cdot \mathrm{s}_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{A}_3^4 = \begin{bmatrix} \mathrm{c}_4 & 0 & \mathrm{s}_4 & a_4\mathrm{c}_4 \\ \mathrm{s}_4 & 0 & -\mathrm{c}_4 & a_4\mathrm{s}_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{c}_4 & 0 & \mathrm{s}_4 & 9.5 \cdot \mathrm{c}_4 \\ \mathrm{s}_4 & 0 & -\mathrm{c}_4 & 9.5 \cdot \mathrm{s}_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{A}_4^5 = \begin{bmatrix} \mathrm{c}_5 & -\mathrm{s}_5 & 0 & 0 \\ \mathrm{s}_5 & \mathrm{c}_5 & 0 & 0 \\ 0 & 0 & 1 & d_5 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{c}_5 & -\mathrm{s}_5 & 0 & 0 \\ \mathrm{s}_5 & \mathrm{c}_5 & 0 & 0 \\ 0 & 0 & 1 & 165 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

donde $c_i = \mathrm{c}\left(\theta_i\right)$ y $s_i = \mathrm{s}\left(\theta_i\right)$.

La matriz del brazo será entonces

$$\mathbf{T}_0^5 = \mathbf{A}_0^1\mathbf{A}_1^2\mathbf{A}_2^3\mathbf{A}_3^4\mathbf{A}_4^5 = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{B.1}$$

donde

$$\begin{aligned}
r_{1,1} &= \mathrm{s}_1\mathrm{s}_5 + \mathrm{c}_5\left[\mathrm{c}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{c}_3 - \mathrm{c}_1\mathrm{s}_2\mathrm{s}_3\right) - \mathrm{s}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_1\mathrm{c}_3\mathrm{s}_2\right)\right], \\
r_{1,2} &= \mathrm{c}_5\mathrm{s}_1 - \mathrm{s}_5\left[\mathrm{c}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{c}_3 - \mathrm{c}_1\mathrm{s}_2\mathrm{s}_3\right) - \mathrm{s}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_1\mathrm{c}_3\mathrm{s}_2\right)\right], \\
r_{1,3} &= \mathrm{c}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_1\mathrm{c}_3\mathrm{s}_2\right) + \mathrm{s}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{c}_3 - \mathrm{c}_1\mathrm{s}_2\mathrm{s}_3\right), \\
r_{2,1} &= \mathrm{c}_5\left[\mathrm{c}_4\left(\mathrm{c}_2\mathrm{c}_3\mathrm{s}_1 - \mathrm{s}_1\mathrm{s}_2\mathrm{s}_3\right) - \mathrm{s}_4\left(\mathrm{c}_2\mathrm{s}_1\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_1\mathrm{s}_2\right)\right] - \mathrm{c}_1\mathrm{s}_5, \\
r_{2,2} &= -\mathrm{c}_1\mathrm{c}_5 - \mathrm{s}_5\left[\mathrm{c}_4\left(\mathrm{c}_2\mathrm{c}_3\mathrm{s}_1 - \mathrm{s}_1\mathrm{s}_2\mathrm{s}_3\right) - \mathrm{s}_4\left(\mathrm{c}_2\mathrm{s}_1\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_1\mathrm{s}_2\right)\right], \\
r_{2,3} &= \mathrm{c}_4\left(\mathrm{c}_2\mathrm{s}_1\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_1\mathrm{s}_2\right) + \mathrm{s}_4\left(\mathrm{c}_2\mathrm{c}_3\mathrm{s}_1 - \mathrm{s}_1\mathrm{s}_2\mathrm{s}_3\right), \\
r_{3,1} &= \mathrm{c}_5\left[\mathrm{c}_4\left(\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_2\right) + \mathrm{s}_4\left(\mathrm{c}_2\mathrm{c}_3 - \mathrm{s}_2\mathrm{s}_3\right)\right], \\
r_{3,2} &= -\mathrm{s}_5\left[\mathrm{c}_4\left(\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_2\right) + \mathrm{s}_4\left(\mathrm{c}_2\mathrm{c}_3 - \mathrm{s}_2\mathrm{s}_3\right)\right], \\
r_{3,3} &= \mathrm{s}_4\left(\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_2\right) - \mathrm{c}_4\left(\mathrm{c}_2\mathrm{c}_3 - \mathrm{s}_2\mathrm{s}_3\right), \\
t_1 &= d_5\left[\mathrm{c}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_1\mathrm{c}_3\mathrm{s}_2\right) + \mathrm{s}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{c}_3 - \mathrm{c}_1\mathrm{s}_2\mathrm{s}_3\right)\right] \\
&\quad + a_2\mathrm{c}_1\mathrm{c}_2 + a_4\mathrm{c}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{c}_3 - \mathrm{c}_1\mathrm{s}_2\mathrm{s}_3\right) - a_4\mathrm{s}_4\left(\mathrm{c}_1\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_1\mathrm{c}_3\mathrm{s}_2\right) \\
&\quad - a_3\mathrm{c}_1\mathrm{s}_2\mathrm{s}_3 + a_3\mathrm{c}_1\mathrm{c}_2\mathrm{c}_3, \\
t_2 &= d_5\left[\mathrm{c}_4\left(\mathrm{c}_2\mathrm{s}_1\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_1\mathrm{s}_2\right) + \mathrm{s}_4\left(\mathrm{c}_2\mathrm{c}_3\mathrm{s}_1 - \mathrm{s}_1\mathrm{s}_2\mathrm{s}_3\right)\right] \\
&\quad + a_2\mathrm{c}_2\mathrm{s}_1 + a_4\mathrm{c}_4\left(\mathrm{c}_2\mathrm{c}_3\mathrm{s}_1 - \mathrm{s}_1\mathrm{s}_2\mathrm{s}_3\right) - a_4\mathrm{s}_4\left(\mathrm{c}_2\mathrm{s}_1\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_1\mathrm{s}_2\right) \\
&\quad - a_3\mathrm{s}_1\mathrm{s}_2\mathrm{s}_3 + a_3\mathrm{c}_2\mathrm{c}_3\mathrm{s}_1, \\
t_3 &= d_1 + a_2\mathrm{s}_2 - d_5\left[\mathrm{c}_4\left(\mathrm{c}_2\mathrm{c}_3 - \mathrm{s}_2\mathrm{s}_3\right) - \mathrm{s}_4\left(\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_2\right)\right] \\
&\quad + a_3\mathrm{c}_2\mathrm{s}_3 + a_3\mathrm{c}_3\mathrm{s}_2 + a_4\mathrm{c}_4\left(\mathrm{c}_2\mathrm{s}_3 + \mathrm{c}_3\mathrm{s}_2\right) + a_4\mathrm{s}_4\left(\mathrm{c}_2\mathrm{c}_3 - \mathrm{s}_2\mathrm{s}_3\right)
\end{aligned}$$

Mediante las identidades trigonométricas

$$\begin{aligned}
\operatorname{sen}\left(\alpha \pm \beta\right) &= \operatorname{sen}\left(\alpha\right)\cos\left(\beta\right) \pm \cos\left(\alpha\right)\operatorname{sen}\left(\beta\right), \\
\cos\left(\alpha \pm \beta\right) &= \cos\left(\alpha\right)\cos\left(\beta\right) \mp \operatorname{sen}\left(\alpha\right)\operatorname{sen}\left(\beta\right),
\end{aligned}$$

podemos obtener una expresión mucho más simplificada de los elementos de la matriz[1]:

$$\mathbf{T}_0^5 = \begin{bmatrix} \mathrm{s}_1\mathrm{s}_5 + \mathrm{c}_5\mathrm{c}_1\mathrm{c}_{2+3+4} & \mathrm{c}_5\mathrm{s}_1 - \mathrm{s}_5\mathrm{c}_1\mathrm{c}_{2+3+4} & \mathrm{c}_1\mathrm{s}_{2+3+4} & t_1 \\ \mathrm{c}_5\mathrm{s}_1\mathrm{c}_{2+3+4} - \mathrm{c}_1\mathrm{s}_5 & -\mathrm{c}_1\mathrm{c}_5 - \mathrm{s}_5\mathrm{s}_1\mathrm{c}_{2+3+4} & \mathrm{s}_1\mathrm{s}_{2+3+4} & t_2 \\ \mathrm{c}_5\mathrm{s}_{2+3+4} & -\mathrm{s}_5\mathrm{s}_{2+3+4} & -\mathrm{c}_{2+3+4} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{B.2}$$

donde

$$\begin{aligned}
t_1 &= \mathrm{c}_1\left(d_5\mathrm{s}_{2+3+4} + a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} + a_4\mathrm{c}_{2+3+4}\right), \\
t_2 &= \mathrm{s}_1\left(d_5\mathrm{s}_{2+3+4} + a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} + a_4\mathrm{c}_{2+3+4}\right), \\
t_3 &= d_1 - d_5\mathrm{c}_{2+3+4} + a_2\mathrm{s}_2 + a_3\mathrm{s}_{2+3} + a_4\mathrm{s}_{2+3+4}.
\end{aligned}$$

[1]Se ha simplificado mediante dichas identidades a mano, y el resultado se ha comprobado con el software Wolfram Mathematica 7.0.

### B.1.2 Comprobación de la cinemática directa para la posición 1

Para comprobar la cinemática directa, vamos a obtener la matriz del brazo para la posición elegida como posición de HOME (Figura B.1, pág. 135), que es la posición en la que los valores de las variables de articulación son

$$\begin{aligned}
\theta_1{}^* &= 90^{\text{o}}, \\
\theta_2{}^* &= 90^{\text{o}}, \\
\theta_3{}^* &= -90^{\text{o}}, \\
\theta_4{}^* &= 0^{\text{o}}, \\
\theta_5{}^* &= 180^{\text{o}}.
\end{aligned}$$

Si obtenemos la matriz del brazo directamente, llegamos a la siguiente matriz:

$$\mathbf{T}_{\text{HOME}} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & a_3 + a_4 \\ 0 & 0 & -1 & d_1 + a_2 - d_5 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 238.1 \\ 0 & 0 & -1 & 323.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ahora calculamos la matriz del brazo a partir de la matriz $\mathbf{T}_0^5$ sin simplificar (Ecuación B.1, pág. 137), sustituyendo los valores de las variables de articulación para la posición de HOME, obteniendo así la matriz

$$\mathbf{T}_{0,\text{HOME}}^5 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 238.1 \\ 0 & 0 & -1 & 323.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

En el caso de sustituir los valores en la matriz $\mathbf{T}_0^5$ simplificada (Ecuación B.2, pág. 137), resulta la matriz

$$\mathbf{T}_{0,\text{HOME}}^5 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 238.1 \\ 0 & 0 & -1 & 323.6 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

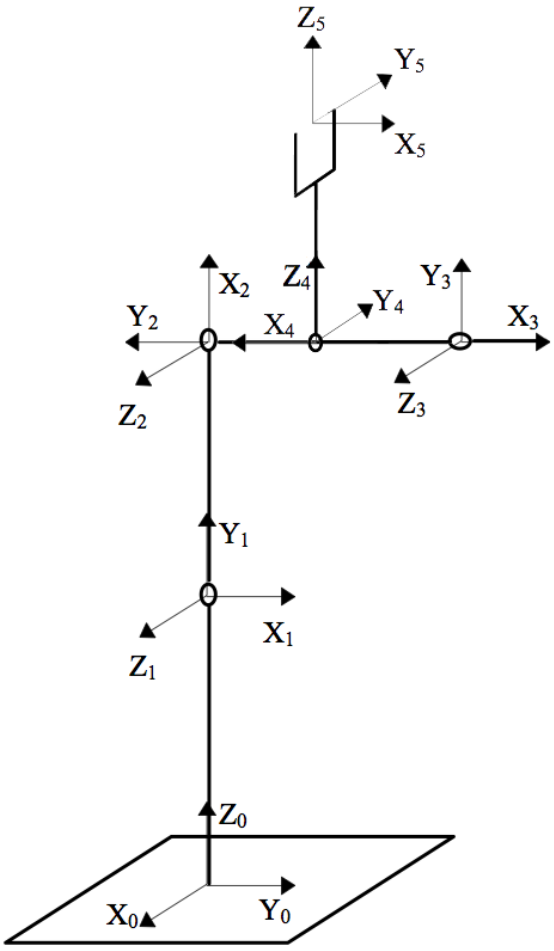
que coincide con la matriz anterior, por lo que se asume que el desarrollo de las matrices es correcto.

### B.1.3 Comprobación de la cinemática directa para la posición 2

Para comprobar la cinemática directa, vamos a obtener la matriz del brazo para la posición 2 (Figura B.2, pág. 139), en la que los valores de las variables de articulación son

$$\begin{aligned}
\theta_1{}^* &= 90^{\text{o}}, \\
\theta_2{}^* &= 90^{\text{o}}, \\
\theta_3{}^* &= -90^{\text{o}}, \\
\theta_4{}^* &= 180^{\text{o}}, \\
\theta_5{}^* &= 180^{\text{o}}.
\end{aligned}$$

Figura B.2: Esquema del robot XR-4 en la posición 2




Si obtenemos la matriz del brazo directamente, llegamos a la siguiente matriz:

$$\mathbf{T}_{\text{POS2}} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & a_3 - a_4 \\ 0 & 0 & 1 & d_1 + a_2 + d_5 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 219.1 \\ 0 & 0 & 1 & 653.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ahora calculamos la matriz del brazo a partir de la matriz $\mathbf{T}_0^5$ sin simplificar (Ecuación B.1, pág. 137), sustituyendo los valores de las variables de articulación para la posición 2, obteniendo así la matriz

$$\mathbf{T}_{0,\text{POS2}}^5 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 219.1 \\ 0 & 0 & 1 & 653.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

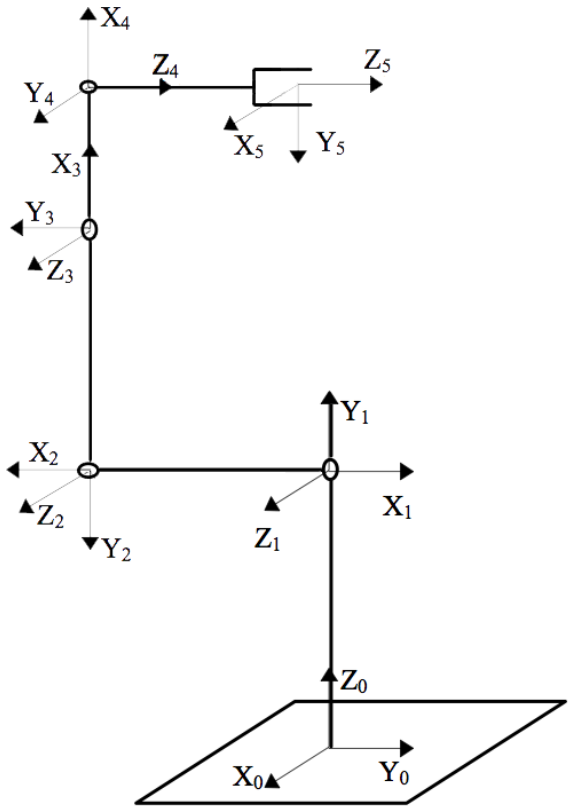
En el caso de sustituir los valores en la matriz $\mathbf{T}_0^5$ simplificada (Ecuación B.2, pág. 137), resulta la matriz

$$\mathbf{T}_{0,\text{POS2}}^5 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 219.1 \\ 0 & 0 & 1 & 653.6 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

que coincide con la matriz anterior, por lo que se asume que el desarrollo de las matrices es correcto.

### B.1.4 Comprobación de la cinemática directa para la posición 3

Figura B.3: Esquema del robot XR-4 en la posición 3




Para comprobar la cinemática directa, vamos a obtener la matriz del brazo para la posición 3 (Figura B.3), en la que los valores de las variables de articulación son

$$
\begin{aligned}
\theta_1{}^* &= 90^o, \\
\theta_2{}^* &= 180^o, \\
\theta_3{}^* &= -90^o, \\
\theta_4{}^* &= 0^o, \\
\theta_5{}^* &= 90^o.
\end{aligned}
$$

Si obtenemos la matriz del brazo directamente, llegamos a la siguiente matriz:

$$
\mathbf{T}_{\text{POS3}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & d5 - a_2 \\ 0 & -1 & 0 & d_1 + a_3 + a_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -63.6 \\ 0 & -1 & 0 & 498.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
$$

Ahora calculamos la matriz del brazo a partir de la matriz $\mathbf{T}_0^5$ sin simplificar (Ecuación B.1, pág. 137), sustituyendo los valores de las variables de articulación para la posición 3, obteniendo así la matriz

$$\mathbf{T}_{0,\text{POS3}}^5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -63.6 \\ 0 & -1 & 0 & 498.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

En el caso de sustituir los valores en la matriz $\mathbf{T}_0^5$ simplificada (Ecuación B.2, pág. 137), resulta la matriz

$$\mathbf{T}_{0,\text{POS3}}^5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -63.6 \\ 0 & -1 & 0 & 498.1 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

que coincide con la matriz anterior, por lo que se asume que el desarrollo de las matrices es correcto.

## B.2 Cinemática inversa

### B.2.1 Resolución de la cinemática inversa

La matriz del brazo es

$$\mathbf{T}_0^5 = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{s}_1\mathrm{s}_5 + \mathrm{c}_5\mathrm{c}_1\mathrm{c}_{2+3+4} & \mathrm{c}_5\mathrm{s}_1 - \mathrm{s}_5\mathrm{c}_1\mathrm{c}_{2+3+4} & \mathrm{c}_1\mathrm{s}_{2+3+4} & t_1 \\ \mathrm{c}_5\mathrm{s}_1\mathrm{c}_{2+3+4} - \mathrm{c}_1\mathrm{s}_5 & -\mathrm{c}_1\mathrm{c}_5 - \mathrm{s}_5\mathrm{s}_1\mathrm{c}_{2+3+4} & \mathrm{s}_1\mathrm{s}_{2+3+4} & t_2 \\ \mathrm{c}_5\mathrm{s}_{2+3+4} & -\mathrm{s}_5\mathrm{s}_{2+3+4} & -\mathrm{c}_{2+3+4} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

donde

$$\begin{aligned} t_1 &= \mathrm{c}_1\left(d_5\mathrm{s}_{2+3+4} + a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} + a_4\mathrm{c}_{2+3+4}\right), \\ t_2 &= \mathrm{s}_1\left(d_5\mathrm{s}_{2+3+4} + a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} + a_4\mathrm{c}_{2+3+4}\right), \\ t_3 &= d_1 - d_5\mathrm{c}_{2+3+4} + a_2\mathrm{s}_2 + a_3\mathrm{s}_{2+3} + a_4\mathrm{s}_{2+3+4}. \end{aligned}$$

Para ${\theta_1}^*$, podríamos fijarnos en la relación entre $r_{2,3}$ y $r_{1,3}$, pero, cuando $\mathrm{s}_{2+3+4} = 0$, llevaría a una indeterminación, por lo que hay que buscar otra forma. Por ello, podemos partir de la relación entre $t_2$ y $t_1$:

$$\frac{t_2}{t_1} = \frac{\mathrm{s}_1}{\mathrm{c}_1} \implies {\theta_1}^* = \operatorname{arc\,tg}\left(\frac{t_2}{t_1}\right).$$

En el caso de ${\theta_5}^*$, podríamos fijarnos en la relación entre $r_{3,2}$ y $r_{3,1}$, pero, cuando $\mathrm{s}_{2+3+4} = 0$, igualmente llevaría a una indeterminación, por lo que hay que buscar otra forma. Observando un poco las dos primeras columnas vemos que

$$\begin{aligned} \mathrm{s}_1 r_{1,1} - \mathrm{c}_1 r_{2,1} &= \mathrm{s}_1^2\mathrm{s}_5 + \mathrm{s}_1\mathrm{c}_1\mathrm{c}_5\mathrm{c}_{2+3+4} - \mathrm{s}_1\mathrm{c}_1\mathrm{c}_5\mathrm{c}_{2+3+4} + \mathrm{c}_1^2\mathrm{s}_5 = \left(\mathrm{s}_1^2 + \mathrm{c}_1^2\right)\mathrm{s}_5, \\ \mathrm{s}_1 r_{1,2} - \mathrm{c}_1 r_{2,2} &= \mathrm{s}_1^2\mathrm{c}_5 - \mathrm{s}_1\mathrm{c}_1\mathrm{s}_5\mathrm{c}_{2+3+4} + \mathrm{c}_1^2\mathrm{c}_5 + \mathrm{s}_1\mathrm{c}_1\mathrm{c}_{2+3+4} = \left(\mathrm{s}_1^2 + \mathrm{c}_1^2\right)\mathrm{c}_5. \end{aligned}$$

De aquí obtenemos que

$$\frac{\mathrm{s}_1 r_{1,1} - \mathrm{c}_1 r_{2,1}}{\mathrm{s}_1 r_{1,2} - \mathrm{c}_1 r_{2,2}} = \frac{\mathrm{s}_5}{\mathrm{c}_5} \implies {\theta_5}^* = \operatorname{arc\,tg}\left(\frac{\mathrm{s}_1 r_{1,1} - \mathrm{c}_1 r_{2,1}}{\mathrm{s}_1 r_{1,2} - \mathrm{c}_1 r_{2,2}}\right).$$

Además, de la matriz del brazo también obtenemos $\alpha = {\theta_2}^* + {\theta_3}^* + {\theta_4}^*$:

$$-r_{3,3} = \mathrm{c}_{2+3+4} \implies {\theta_2}^* + {\theta_3}^* + {\theta_4}^* = \alpha = \arccos\left(-r_{3,3}\right).$$

Esto nos servirá para el cálculo de los ángulos ${\theta_2}^*$, ${\theta_3}^*$ y ${\theta_4}^*$.

El cálculo de los ángulos ${\theta_2}^*$, ${\theta_3}^*$ y ${\theta_4}^*$ es algo más complicado. Para empezar, podemos tomar

$$\begin{aligned} t_2 &= \mathrm{s}_1\left(d_5\mathrm{s}_{2+3+4} + a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} + a_4\mathrm{c}_{2+3+4}\right), \\ t_3 &= d_1 - d_5\mathrm{c}_{2+3+4} + a_2\mathrm{s}_2 + a_3\mathrm{s}_{2+3} + a_4\mathrm{s}_{2+3+4} \end{aligned}$$

para obtener las ecuaciones

$$\begin{aligned} A &= a_2\mathrm{s}_2 + a_3\mathrm{s}_{2+3} = t_3 - d_1 + d_5\mathrm{c}_{2+3+4} - a_4\mathrm{s}_{2+3+4}, \\ B &= a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} = \frac{t_2}{\mathrm{s}_1} - d_5\mathrm{s}_{2+3+4} - a_4\mathrm{c}_{2+3+4}. \end{aligned}$$

Sin embargo, nótese que habrá que comprobar si $\mathrm{s}_1 = 0$ y, en caso afirmativo, $B$ deberá calcularse a partir de $t_1$ de la forma

$$B = a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} = \frac{t_1}{\mathrm{c}_1} - d_5\mathrm{s}_{2+3+4} - a_4\mathrm{c}_{2+3+4}.$$

Ahora elevamos $A$ y $B$ al cuadrado:

$$\begin{aligned} \left(a_2\mathrm{s}_2 + a_3\mathrm{s}_{2+3}\right)^2 &= a_2^2\mathrm{s}_2^2 + 2a_2a_3\mathrm{s}_2\mathrm{s}_{2+3} + a_3^2\mathrm{s}_{2+3}^2, \\ \left(a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3}\right)^2 &= a_2^2\mathrm{c}_2^2 + 2a_2a_3\mathrm{c}_2\mathrm{c}_{2+3} + a_3^2\mathrm{c}_{2+3}^2. \end{aligned}$$

Si sumamos $A^2$ y $B^2$, tendremos que

$$\begin{aligned} A^2 + B^2 &= a_2^2 + 2a_2a_3\mathrm{c}_{2+3-2} + a_3^2 = \\ &= a_2^2 + 2a_2a_3\mathrm{c}_3 + a_3^2 = \\ &= a_2^2 + 2a_2a_3\mathrm{c}_3 + a_3^2. \end{aligned}$$

y podremos calcular ${\theta_3}^*$ como

$$A^2 + B^2 = a_2^2 + 2a_2a_3\mathrm{c}_3 + a_3^2 \implies {\theta_3}^* = \pm\arccos\left(\frac{A^2 + B^2 - a_2^2 - a_3^2}{2a_2a_3}\right).$$

Puesto que nos interesa la solución "elbow-up" (con el codo hacia arriba), en el algoritmo de cálculo de las variables de articulación deberemos tomar siempre la solución negativa de ${\theta_3}^*$.

Ya tenemos una expresión para calcular ${\theta_3}^*$. Ahora resta calcular ${\theta_2}^*$ y ${\theta_4}^*$. A partir de $A$ y $B$ podemos calcular ${\theta_2}^*$ si expandimos las expresiones del seno y del coseno utilizando identidades trigonométricas:

$$\begin{aligned} A &= a_2\mathrm{s}_2 + a_3\mathrm{s}_{2+3} = a_2\mathrm{s}_2 + a_3\left(\mathrm{s}_2\mathrm{c}_3 + \mathrm{c}_2\mathrm{s}_3\right) = \left(a_2 + a_3\mathrm{c}_3\right)\mathrm{s}_2 + \left(a_3\mathrm{s}_3\right)\mathrm{c}_2, \\ B &= a_2\mathrm{c}_2 + a_3\mathrm{c}_{2+3} = a_2\mathrm{s}_2 + a_3\left(\mathrm{c}_2\mathrm{c}_3 - \mathrm{s}_2\mathrm{s}_3\right) = \left(a_2 + a_3\mathrm{c}_3\right)\mathrm{c}_2 - \left(a_3\mathrm{s}_3\right)\mathrm{s}_2. \end{aligned}$$

De aquí obtenemos el sistema de ecuaciones

$$\begin{cases} (a_2 + a_3\mathrm{c}_3)\,\mathrm{s}_2 + (a_3\mathrm{s}_3)\,\mathrm{c}_2 = A \\ (-a_3\mathrm{s}_3)\,\mathrm{s}_2 + (a_2 + a_3\mathrm{c}_3)\,\mathrm{c}_2 = B \end{cases},$$

que podemos resolver por el método de Cramer. Para ello, consideremos la matriz

$$\mathbf{M} = \begin{bmatrix} a_2 + a_3\mathrm{c}_3 & a_3\mathrm{s}_3 \\ -a_3\mathrm{s}_3 & a_2 + a_3\mathrm{c}_3 \end{bmatrix},$$

tal que

$$|\mathbf{M}| = (a_2 + a_3\mathrm{c}_3)^2 + (a_3\mathrm{s}_3)^2 = a_2^2 + 2a_2a_3\mathrm{c}_3 + \mathrm{c}_3^2 + (a_3\mathrm{s}_3)^2\,.$$

Entonces obtenemos que

$$\mathrm{s}_2 = \frac{\begin{vmatrix} A & a_3\mathrm{s}_3 \\ B & a_2 + a_3\mathrm{c}_3 \end{vmatrix}}{|\mathbf{M}|} = \frac{A\,(a_2 + a_3\mathrm{c}_3) - Ba_3\mathrm{s}_3}{|\mathbf{M}|},$$

$$\mathrm{c}_2 = \frac{\begin{vmatrix} a_2 + a_3\mathrm{c}_3 & A \\ -a_3\mathrm{s}_3 & B \end{vmatrix}}{|\mathbf{M}|} = \frac{B\,(a_2 + a_3\mathrm{c}_3) + Aa_3\mathrm{s}_3}{|\mathbf{M}|}.$$

Dado que conocemos el seno y el coseno del ángulo, es posible entonces calcular el ángulo por medio de la arcotangente:

$${\theta_2}^* = \mathrm{arc\,tg}\left(\frac{\mathrm{s}_2}{\mathrm{c}_1}\right) = \mathrm{arc\,tg}\left(\frac{A\,(a_2 + a_3\mathrm{c}_3) - Ba_3\mathrm{s}_3}{B\,(a_2 + a_3\mathrm{c}_3) + Aa_3\mathrm{s}_3}\right).$$

Por último, dado que conocemos ${\theta_2}^*$, ${\theta_3}^*$ y $\alpha = {\theta_2}^* + {\theta_3}^* + {\theta_4}^*$, podemos calcular ${\theta_4}^*$ fácilmente como

$${\theta_4}^* = \alpha - {\theta_2}^* - {\theta_3}^*.$$

En resumen, si conocemos la matriz del brazo

$$\mathbf{T}_0^5 = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

podemos calcular las variables de articulación mediante las siguientes expresiones:

$$\begin{aligned} {\theta_1}^* &= \mathrm{arc\,tg}\left(\frac{t_2}{t_1}\right), \\ {\theta_5}^* &= \mathrm{arc\,tg}\left(\frac{\mathrm{s}_1 r_{1,1} - \mathrm{c}_1 r_{2,1}}{\mathrm{s}_1 r_{1,2} - \mathrm{c}_1 r_{2,2}}\right), \\ {\theta_3}^* &= \pm\mathrm{arc\,cos}\left(\frac{A^2 + B^2 - a_2^2 - a_3^2}{2a_2a_3}\right), \\ {\theta_2}^* &= \mathrm{arc\,tg}\left(\frac{A\,(a_2 + a_3\mathrm{c}_3) - Ba_3\mathrm{s}_3}{B\,(a_2 + a_3\mathrm{c}_3) + Aa_3\mathrm{s}_3}\right), \\ {\theta_4}^* &= \alpha - {\theta_2}^* - {\theta_3}^*, \end{aligned}$$

donde

$$A = t_3 - d_1 - d_5 r_{3,3} - a_4 \mathrm{s}\left(\alpha\right),$$
$$B = \begin{cases} \dfrac{t_1}{\mathrm{c}\left({\theta_1}^*\right)} - d_5 \mathrm{s}\left(\alpha\right) + a_4 r_{3,3}, & \text{si } \mathrm{s}_1 = 0 \\ \dfrac{t_2}{\mathrm{s}\left({\theta_1}^*\right)} - d_5 \mathrm{s}\left(\alpha\right) + a_4 r_{3,3}, & \text{en otro caso} \end{cases},$$

siendo $\alpha = {\theta_2}^* + {\theta_3}^* + {\theta_4}^* = \arccos\left(-r_{3,3}\right)$.
En el cálculo de ${\theta_3}^*$ vemos que la cinemática inversa puede tener múltiples soluciones. Puesto que nos interesa la solución "elbow-up" (con el codo hacia arriba), en el algoritmo de cálculo de las variables de articulación deberemos tomar siempre la solución negativa de ${\theta_3}^*$.

### B.2.2 Comprobación de la cinemática inversa para la posición 1

La matriz del brazo en la posición de HOME es

$$\mathbf{T}^5_{0,\mathrm{HOME}} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 238.1 \\ 0 & 0 & -1 & 323.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Resolviendo la cinemática inversa, obtenemos los resultados de la Tabla B.2.

Tabla B.2: Resultados de la cinemática inversa del XR-4 para la posición de HOME

| Variable de articulación | Solución de la cinemática inversa | Solución correcta |
|---|---|---|
| ${\theta_1}^*$ | $90^o$ | $90^o$ |
| ${\theta_2}^*$ | $90^o$ | $90^o$ |
| ${\theta_3}^*$ | $-90^o$ | $-90^o$ |
| ${\theta_4}^*$ | $0^o$ | $0^o$ |
| ${\theta_5}^*$ | $180^o$ | $180^o$ |

### B.2.3 Comprobación de la cinemática inversa para la posición 2

La matriz del brazo en la posición 2 es

$$\mathbf{T}^5_{0,\mathrm{POS2}} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 219.1 \\ 0 & 0 & 1 & 653.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Tabla B.3: Resultados de la cinemática inversa del XR-4 para la posición 2

| Variable de articulación | Solución de la cinemática inversa | Solución correcta |
|---|---|---|
| ${\theta_1}^*$ | $90^o$ | $90^o$ |
| ${\theta_2}^*$ | $90^o$ | $90^o$ |
| ${\theta_3}^*$ | $-90^o$ | $-90^o$ |
| ${\theta_4}^*$ | $180^o$ | $180^o$ |
| ${\theta_5}^*$ | $-180^o$ | $180^o$ |

Resolviendo la cinemática inversa, obtenemos los resultados de la Tabla B.3. En dicha tabla se puede observar que todos los ángulos coinciden, excepto ${\theta_5}^*$, aunque $180^o$ y $-180^o$ representan el mismo ángulo. Además, éste es el ángulo de "roll" de la pinza, por lo que no se producirá ningún choque contra un tope mecánico. Sin embargo, habría que revisar por qué el ángulo es diferente, ya que lo que sí podría pasar es que se obtenga un ángulo incorrecto que produzca que la pinza esté orientada de tal forma que no pueda agarrar el objeto que debe agarrar.

### B.2.4 Comprobación de la cinemática inversa para la posición 3

La matriz del brazo en la posición 3 es

$$\mathbf{T}^5_{0,\text{POS3}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -63.6 \\ 0 & -1 & 0 & 498.1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.3}$$

Resolviendo la cinemática inversa, obtenemos los resultados de la Tabla B.4. En este caso

Tabla B.4: Resultados de la cinemática inversa del XR-4 para la posición 3

| Variable de articulación | Solución de la cinemática inversa | Solución correcta |
|---|---|---|
| ${\theta_1}^*$ | $-90^o$ | $90^o$ |
| ${\theta_2}^*$ | $89.801727^o$ | $180^o$ |
| ${\theta_3}^*$ | $-85.033767^o$ | $-90^o$ |
| ${\theta_4}^*$ | $-94.767961^o$ | $0^o$ |
| ${\theta_5}^*$ | $-90^o$ | $90^o$ |

vemos que ningún ángulo coincide, ya sea en valor o en signo. Sin embargo, al sustituir la solución de la cinemática inversa en la matriz del brazo obtenida de la cinemática directa, se obtiene la misma matriz $\mathbf{T}^5_0$ que la de la Ecuación B.3, con lo cual la solución obtenida de la cinemática inversa parece ser correcta.

# Referencias

[1] *Rhino Mark IV Owner's Manual*.

[2] Robert Joseph Schilling. *Fundamentals of Robotics: Analysis and Control*. Prentice Hall, 1990.

[3] Antonio Barrientos, Luis Felipe Peñín, Carlos Balaguer, and Rafael Aracil. *Fundamentos de Robótica*. McGraw-Hill, 1997.

# C

# Comandos para la intercomunicación entre GPMCU y MCMCU

Dado que en la arquitectura software propuesta para el firmware RhinoChip OS (Sección 3.4.2, pág. 44) contempla la división de las tareas de cómputo entre dos microcontroladores, uno de propósito general y otro de control de motores, se hace necesaria la comunicación entre ambos, tal como se ha expuesto en la Sección 4.1.7 (pág. 90).

Para ello, se ha diseñado un repertorio de instrucciones que permiten al GPMCU enviar órdenes al MCMCU para realizar diversas tareas de configuración, monitorización y ejecución de movimientos. Los comandos implementados en la primera versión del prototipo que ha sido objeto de este proyecto se listan en la Tabla C.1.

Tabla C.1: Comandos para la intercomunicación entre GPMCU y MCMCU

| Comando | Significado |
|---|---|
| RA | Lee el codificador del motor A |
| RB | Lee el codificador del motor B |
| RC | Lee el codificador del motor C |
| RD | Lee el codificador del motor D |
| RE | Lee el codificador del motor E |
| RF | Lee el codificador del motor F |
| SA | Para el motor A (establece el ciclo de trabajo a cero) |
| SB | Para el motor B (establece el ciclo de trabajo a cero) |
| SC | Para el motor C (establece el ciclo de trabajo a cero) |
| SD | Para el motor D (establece el ciclo de trabajo a cero) |
| SE | Para el motor E (establece el ciclo de trabajo a cero) |
| SF | Para el motor F (establece el ciclo de trabajo a cero) |
| SS | Para para todos los motores (establece ciclo de trabajo a cero) |
| AA | Establece la posición de destino absoluta para el motor A en el espacio de las articulaciones |

| Comando | Significado |
|---|---|
| AB | Establece la posición de destino absoluta para el motor B en el espacio de las articulaciones |
| AC | Establece la posición de destino absoluta para el motor C en el espacio de las articulaciones |
| AD | Establece la posición de destino absoluta para el motor D en el espacio de las articulaciones |
| AE | Establece la posición de destino absoluta para el motor E en el espacio de las articulaciones |
| AF | Establece la posición de destino absoluta para el motor F en el espacio de las articulaciones |
| BA | Establece la posición de destino relativa para el motor A en el espacio de las articulaciones |
| BB | Establece la posición de destino relativa para el motor B en el espacio de las articulaciones |
| BC | Establece la posición de destino relativa para el motor C en el espacio de las articulaciones |
| BD | Establece la posición de destino relativa para el motor D en el espacio de las articulaciones |
| BE | Establece la posición de destino relativa para el motor E en el espacio de las articulaciones |
| BF | Establece la posición de destino relativa para el motor F en el espacio de las articulaciones |
| CX | Establece la posición de destino absoluta en el espacio cartesiano (coordinate X) |
| CY | Establece la posición de destino absoluta en el espacio cartesiano (coordinate Y) |
| CZ | Establece la posición de destino absoluta en el espacio cartesiano (coordinate Z) |
| DX | Establece la posición de destino relativa en el espacio cartesiano (coordinate X) |
| DY | Establece la posición de destino relativa en el espacio cartesiano (coordinate Y) |
| DZ | Establece la posición de destino relativa en el espacio cartesiano (coordinate Z) |
| MI | Inicia un movimiento independiente (para cuando se alcanza la posición de destino) |
| MC | Inicia un movimiento coordinado coordinated (para cuando se alcanza la posición de destino) |
| MP | Mueve los motores de acuerdo con los registros de PWM y dirección (no para hasta que no se ponga el ciclo de trabajo a cero) |
| NA,m | Establece el modo del motor A a 0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto |

| Comando | Significado |
|---|---|
| NB,m | Establece el modo del motor B a 0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto |
| NC,m | Establece el modo del motor C a 0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto |
| ND,m | Establece el modo del motor D a 0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto |
| NE,m | Establece el modo del motor E a 0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto |
| NF,m | Establece el modo del motor F a 0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto |
| PA,d | Establece el nivel de PWM y la dirección para el motor A |
| PB,d | Establece el nivel de PWM y la dirección para el motor B |
| PC,d | Establece el nivel de PWM y la dirección para el motor C |
| PD,d | Establece el nivel de PWM y la dirección para el motor D |
| PE,d | Establece el nivel de PWM y la dirección para el motor E |
| PF,d | Establece el nivel de PWM y la dirección para el motor F |
| QA | Lee el modo del motor A (0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto) |
| QB | Lee el modo del motor B (0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto) |
| QC | Lee el modo del motor C (0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto) |
| QD | Lee el modo del motor D (0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto) |
| QE | Lee el modo del motor E (0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto) |
| QF | Lee el modo del motor F (0 = inactivo, 1 = trapezoidal, 2 = velocidad, 3 = bucle abierto) |
| RP | Restaura la configuración de PWM desde la EEPROM |
| EA | Habilita el control PID para el motor A |
| EB | Habilita el control PID para el motor B |
| EC | Habilita el control PID para el motor C |
| ED | Habilita el control PID para el motor D |
| EE | Habilita el control PID para el motor E |
| EF | Habilita el control PID para el motor F |
| DA | Deshabilita el control PID para el motor A |
| DB | Deshabilita el control PID para el motor B |
| DC | Deshabilita el control PID para el motor C |
| DD | Deshabilita el control PID para el motor D |
| DE | Deshabilita el control PID para el motor E |
| DF | Deshabilita el control PID para el motor F |
| KA | Borra el registro de posición del motor A (establece su valor a cero) |
| KB | Borra el registro de posición del motor B (establece su valor a cero) |

| Comando | Significado |
|---|---|
| KC | Borra el registro de posición del motor C (establece su valor a cero) |
| KD | Borra el registro de posición del motor D (establece su valor a cero) |
| KE | Borra el registro de posición del motor E (establece su valor a cero) |
| KF | Borra el registro de posición del motor F (establece su valor a cero) |
| IA,d | Incrementa el registro de posición deseada del motor A en $d$ pasos |
| IB,d | Incrementa el registro de posición deseada del motor B en $d$ pasos |
| IC,d | Incrementa el registro de posición deseada del motor C en $d$ pasos |
| ID,d | Incrementa el registro de posición deseada del motor D en $d$ pasos |
| IE,d | Incrementa el registro de posición deseada del motor E en $d$ pasos |
| IF,d | Incrementa el registro de posición deseada del motor F en $d$ pasos |
| GA,d | Establece el registro de posición deseada del motor A |
| GB,d | Establece el registro de posición deseada del motor B |
| GC,d | Establece el registro de posición deseada del motor C |
| GD,d | Establece el registro de posición deseada del motor D |
| GE,d | Establece el registro de posición deseada del motor E |
| GF,d | Establece el registro de posición deseada del motor F |
| XA | Devuelve 1 si el motor A está ejecutando un movimiento trapezoidal |
| XB | Devuelve 1 si el motor B está ejecutando un movimiento trapezoidal |
| XC | Devuelve 1 si el motor C está ejecutando un movimiento trapezoidal |
| XD | Devuelve 1 si el motor D está ejecutando un movimiento trapezoidal |
| XE | Devuelve 1 si el motor E está ejecutando un movimiento trapezoidal |
| XF | Devuelve 1 si el motor F está ejecutando un movimiento trapezoidal |
| XX | Devuelve 1 si algún motor está ejecutando un movimiento trapezoidal |
| AR | Lee la aceleración del sistema |
| AS,d | Establece la aceleración del sistema |
| RV | Lee la velocidad del sistema |
| SV,d | Establece la velocidad del sistema |
| CA | Lee el nivel de PWM y la dirección del motor A |
| CB | Lee el nivel de PWM y la dirección del motor B |
| CC | Lee el nivel de PWM y la dirección del motor C |
| CD | Lee el nivel de PWM y la dirección del motor D |
| CE | Lee el nivel de PWM y la dirección del motor E |
| CF | Lee el nivel de PWM y la dirección del motor F |
| FA | Lee la posición comandada del motor A |
| FB | Lee la posición comandada del motor B |
| FC | Lee la posición comandada del motor C |
| FD | Lee la posición comandada del motor D |
| FE | Lee la posición comandada del motor E |
| FF | Lee la posición comandada del motor F |
| YA | Lee la velocidad deseada del motor A |
| YB | Lee la velocidad deseada del motor B |
| YC | Lee la velocidad deseada del motor C |

| Comando | Significado |
|---|---|
| YD | Lee la velocidad deseada del motor D |
| YE | Lee la velocidad deseada del motor E |
| YF | Lee la velocidad deseada del motor F |
| ZA,d | Establece la velocidad deseada motor A |
| ZB,d | Establece la velocidad deseada motor B |
| ZC,d | Establece la velocidad deseada motor C |
| ZD,d | Establece la velocidad deseada motor D |
| ZE,d | Establece la velocidad deseada motor E |
| ZF,d | Establece la velocidad deseada motor F |
| UP,m,d | Establece la ganancia proporcional del motor $m$ |
| UI,m,d | Establece la ganancia integral del motor $m$ |
| UD,m,d | Establece la ganancia derivativa del motor $m$ |
| WP,m | Lee la ganancia proporcional del motor $m$ |
| WI,m | Lee la ganancia integral del motor $m$ |
| WD,m | Lee la ganancia derivativa del motor $m$ |