



Universidad
de Cádiz

Escuela Superior
de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

**DISEÑO, DESARROLLO E
IMPLEMENTACIÓN DEL SOFTWARE DE
INTEGRACIÓN PARA CONTROL DE
MOVIMIENTO, VISIÓN ARTIFICIAL,
RECONOCIMIENTO DE VOZ, SENSADO
TÁCTIL Y EMG PARA PRÓTESIS DE MANO
ROBÓTICA PARA NIÑOS**

AUTOR: PRÁXEDES NEIRA GÓMEZ

Cádiz, septiembre 2025



Universidad
de Cádiz

Escuela Superior
de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

**DISEÑO, DESARROLLO E
IMPLEMENTACIÓN DEL SOFTWARE DE
INTEGRACIÓN PARA CONTROL DE
MOVIMIENTO, VISIÓN ARTIFICIAL,
RECONOCIMIENTO DE VOZ, SENSADO
TÁCTIL Y EMG PARA PRÓTESIS DE MANO
ROBÓTICA PARA NIÑOS**

TUTOR: BLAS SALVADOR DOMÍNGUEZ
CO-TUTOR: ARTURO MORGADO ESTÉVEZ
AUTOR: PRÁXEDES NEIRA GÓMEZ

Cádiz, septiembre 2025

Declaración personal de autoría

Práxedes Neira Gómez con DNI , estudiante del Grado en Ingeniería Informática en la Escuela Superior de Ingeniería de la Universidad de Cádiz, como autor de este documento académico titulado Diseño, desarrollo e implementación del software de integración para control de movimiento, visión artificial, reconocimiento de voz, sensado táctil y EMG para prótesis de mano robótica para niños y presentado como Trabajo Final de Grado

DECLARO QUE

Es un trabajo original, que no copio ni utilizo parte de obra alguna sin mencionar de forma clara y precisa su origen tanto en el cuerpo del texto como en su bibliografía y que no empleo datos de terceros sin la debida autorización, de acuerdo con la legislación vigente. Asimismo, declaro que soy plenamente consciente de que no respetar esta obligación podrá implicar la aplicación de sanciones académicas, sin perjuicio de otras actuaciones que pudieran iniciarse. En Puerto Real, a 01 de septiembre de 2025.

Fdo: Práxedes Neira Gómez

Agradecimientos

Gracias a mis padres y mi hermana, por ser mi apoyo incondicional durante toda mi etapa académica y mi vida. Porque gracias a ellos, me he convertido en la persona que soy hoy.

A todos los profesores que he tenido durante mi vida. A Emma, por brindarme su ayuda y apoyarme en uno de mis momentos más duros. A los profesores del I.E.S. Seritium, por hacerme ver la belleza de las ciencias y proporcionarme la cultura que tengo. A mis profesores de la ESI, por guiarme y formarme en mi carrera como ingeniero, en especial Juan Fran y Juan Carlos, por su paciencia y ayuda.

A los amigos que tenía y a los nuevos que he conocido en la Escuela, por hacerme reír constantemente, por la ayuda que nos hemos brindado y por hacer de esta etapa algo que nunca olvidaré.

A mis tutores, Blas y Arturo, por su ayuda, por enseñarme muchísimo de electrónica, y por hacer que este proceso no solo fuera apasionante, sino también muy divertido. A todos los compañeros del Laboratorio de Robótica Aplicada, especialmente a Miguel, por ser el compañero soñado para un trabajo, por todas las tardes que hemos tenido que pasar en la ESI y por estar ahí en los buenos y en los malos momentos.

Resumen

Este proyecto se centra en el diseño, desarrollo e implementación del software de integración que permita controlar los distintos elementos que forman una prótesis de mano robótica destinada a niños. Dichos elementos son: motores DC, que pueden manejar dedos; una cámara, que se encarga de la detección de objetos mediante visión artificial con OpenCV; un sistema de reconocimiento de voz; sensores capacitivos, que permiten detectar el tacto; y sensores electromiográficos (EMG), para detectar contracciones musculares en el antebrazo. Cada uno de estos subsistemas está gobernado por microcontroladores de la familia STM32 y colaboran entre ellos bajo la dirección de una Raspberry Pi, mediante comunicación I²C. Esta Raspberry Pi es, además, la encargada de la programación de los microcontroladores a través de OpenOCD. El sistema incluye también características lúdicas, como luces RGB y sonidos, para mejorar la experiencia del usuario infantil.

A lo largo de esta memoria se describen los pasos seguidos durante el estudio y el desarrollo del sistema, las distintas tecnologías consideradas, así como varias comparativas entre ellas. Asimismo, se muestran pruebas del funcionamiento final del sistema y las conclusiones obtenidas.

Palabras clave: Software de integración, prótesis robótica, mano robótica, niños, motores DC, visión artificial, OpenCV, reconocimiento de voz, sensores capacitivos, sensores EMG, red neuronal, microcontroladores, STM32, Raspberry Pi, I²C, OpenOCD, RGB.

Índice general

Índice de figuras	XI
-------------------	----

Índice de tablas	XV
------------------	----

I Prolegómeno	1
----------------------	----------

1. Introducción	3
------------------------	----------

1.1. Motivación	3
1.2. Alcance y Objetivos	4
1.3. Glosario de términos	6
1.4. Organización del documento	7

2. Antecedentes	9
------------------------	----------

2.1. Contexto	9
2.1.1. Introducción a las prótesis robóticas actuales	9
2.1.2. Evolución histórica de las prótesis robóticas	12
2.1.3. Prótesis robóticas de manos para niños	16
2.2. Estado de la técnica	18
2.2.1. Sensórica	18
2.2.2. Actuadores	22
2.2.3. Control	23
2.2.4. Algoritmos	24
2.2.5. Protocolos de comunicación	26
2.2.6. Programación y software en prótesis	27

3. Plan de gestión de proyecto	30
---------------------------------------	-----------

3.1. Metodología de desarrollo	30
3.1.1. Trabajos previos	33
3.2. Planificación del proyecto	39
3.3. Tecnologías	42
3.3.1. Hardware	43
3.3.2. Software	48
3.4. Costes	52
3.5. Riesgos	53
3.6. Tipos de pruebas a realizar	53

II Desarrollo	55
----------------------	-----------

4. Análisis del Sistema	57
--------------------------------	-----------

4.1. Trabajos previos y cambios realizados	57
--	----

4.1.1.	Análisis y adaptación del trabajo sobre control de motores, comunicación con Raspberry y arquitectura	57
4.1.2.	Análisis y adaptación del trabajo sobre reconocimiento de voz	59
4.1.3.	Análisis y adaptación del trabajo sobre reconocimiento de objetos mediante visión artificial y feedback de tacto	61
4.1.4.	Análisis y adaptación del trabajo sobre sensores EMG	64
4.2.	Funcionamiento de cada subsistema	66
4.2.1.	Subsistema de control de motores	66
4.2.2.	Subsistema de reconocimiento de voz	67
4.2.3.	Subsistema de sensado de tacto	69
4.2.4.	Subsistema de reconocimiento de objetos	70
4.2.5.	Subsistema de sensado EMG y retroalimentación de tacto	70
4.3.	Funcionamiento del sistema central	73
5.	Diseño del Sistema	76
5.1.	Arquitectura del Sistema	76
5.1.1.	Comunicación de subsistemas	77
5.2.	Diseño de la Interfaz de Usuario	81
5.3.	Innovación y mejora	82
5.3.1.	Agarre adaptativo	82
5.3.2.	Programación con OpenOCD	83
5.3.3.	Depuración de cada subsistema mediante UART virtual en Raspberry	84
5.3.4.	Mejora en el código de reconocimiento de voz y modo de bajo consumo	84
5.3.5.	Mejora en la red neuronal del subsistema de sensado EMG y adición de motores vibradores para realimentación del tacto	85
5.3.6.	Mejora y rediseño en el código de sensado de tacto	86
5.3.7.	Mejora en el uso de OpenCV para reconocimiento de objetos	88
6.	Implementación del Sistema	91
6.1.	Entorno de Construcción	91
6.2.	Código Fuente	91
6.2.1.	Organización del código fuente	91
6.2.2.	Implementación del subsistema de control de motores	93
6.2.3.	Implementación del subsistema de sensado de tacto	94
6.2.4.	Implementación del subsistema de reconocimiento de voz	95
6.2.5.	Implementación del subsistema de reconocimiento de objetos	96
6.2.6.	Implementación del subsistema de sensado EMG y retroalimentación de tacto	96
6.2.7.	Implementación de los subsistemas de iluminación RGB y salida de audio	104
6.2.8.	Implementación del sistema central	105
7.	Integración del Sistema	109
7.1.	Integración Hardware	109
7.2.	Integración Software	109
7.2.1.	Integración del subsistema de control de motores	110

7.2.2.	Integración del subsistema de sensado de tacto	112
7.2.3.	Integración del subsistema de reconocimiento de voz	113
7.2.4.	Integración del subsistema de sensado EMG y realimentación del tacto	113
7.2.5.	Integración del subsistema de reconocimiento de objetos	114
7.2.6.	Integración de los subsistemas de iluminación RGB y salida de audio	115
8.	Pruebas del Sistema	117
8.1.	Estrategia	117
8.2.	Pruebas de Funcionamiento	117
8.2.1.	Pruebas de la placa de pruebas del microcontrolador STM32L496RGT6117	
8.2.2.	Pruebas para comprender el funcionamiento del AD7147	118
8.3.	Pruebas Unitarias	120
8.3.1.	Pruebas del subsistema de control de motores	121
8.3.2.	Pruebas del subsistema de sensado de tacto	122
8.3.3.	Pruebas del subsistema de reconocimiento de voz	124
8.3.4.	Pruebas del subsistema de sensado EMG y realimentación del tacto	126
8.3.5.	Pruebas del subsistema de reconocimiento de objetos	127
8.3.6.	Pruebas de los subsistemas de iluminación RGB y salida de audio	128
8.4.	Pruebas de Integración	128
8.4.1.	Integración del subsistema de control de motores	129
8.4.2.	Integración del subsistema de sensado de tacto	130
8.4.3.	Integración del subsistema de reconocimiento de voz	132
8.4.4.	Integración del subsistema de sensado EMG	132
8.4.5.	Integración del subsistema de reconocimiento de objetos	134
8.4.6.	Integración de los subsistemas de iluminación RGB y salida de audio	135
8.4.7.	Integración completa	135
8.5.	Discusión de los resultados de las pruebas	140
III	Epílogo	142
9.	Conclusiones	144
9.1.	Discusión y objetivos alcanzados	144
9.2.	Lecciones aprendidas	146
9.3.	Trabajo futuro	147
	Bibliografía	150
IV	Anexos	157
A.	Manual del desarrollador	159
A.1.	Introducción	159
A.2.	Preparación del entorno de trabajo	159
A.2.1.	Hardware	159

A.2.2. Software para la Raspberry	159
A.2.3. Software relacionado con los microcontroladores STM	169
B. Manual de usuario	174
B.1. Introducción	174
B.2. Instalación	174
B.2.1. Requisitos	174
B.2.2. Procedimiento	175
B.3. Uso del sistema	176
B.3.1. Modo Voz	176
B.3.2. Modo Cámara	176
B.3.3. Modo Mano	177
B.3.4. Tacto	177
C. Esquemáticos	178
C.1. PCB principal	179
C.1.1. Esquemático 1 PCB principal	179
C.1.2. Esquemático 2 PCB principal	180
C.1.3. Board Top PCB principal	181
C.1.4. Board Bottom PCB principal	182
C.1.5. Board Capa 1 PCB principal	183
C.1.6. Board Capa 2 PCB principal	184
C.2. PCB IC sensores capacitivos	186
C.2.1. Esquemático PCB sensores capacitivos	186
C.2.2. Board Top PCB capacitivos	187
C.2.3. Board Bottom PCB capacitivos	188
C.3. PCB botones y LCD	190
C.3.1. Esquemático PCB botones y LCD	190
C.3.2. Board PCB botones y LCD	191
C.4. PCB motores	193
C.4.1. Esquemático PCB motores	193
C.4.2. Board Top PCB motores	194
C.4.3. Board Bottom PCB motores	195

Índice de figuras

2.1. Mano robótica BeBionic	11
2.2. Mano robótica i-Limb	11
2.3. Primera prótesis conocida	13
2.4. Mano neumática. Posiblemente el primer componente protésico motorizado	14
2.5. Componente de mano eléctrica. Quizás el primer mecanismo eléctrico de mano	14
2.6. Mano eléctrica usada por Reiter en el desarrollo de la primera prótesis mioeléctrica (Circa 1943)	15
2.7. Dos vistas de la mano de Vaduz	15
2.8. Mano de Bottommley	15
2.9. Mano de Sven	16
2.10. <i>Hero ARM</i> de <i>Open Bionics</i>	17
2.11. <i>Phoenix Hand V2</i> de e-NABLE	17
2.12. <i>System Electric Hand 2000</i> de Ottobock	17
2.13. Sensor resistivo	20
2.14. Sensor capacitivo	20
2.15. Raspberry Pi Camera Module 3	21
3.1. Arduino UNO	30
3.2. Placas de desarrollo STM	31
3.3. PCB STM32-L496RGT6	32
3.4. PCB Capacitivos y Motores	33
3.5. Módulo MAX98357 de Adafruit	34
3.6. LPKF para fabricación de PCBs	34
3.7. Arquitectura hardware propuesta en trabajo previo	35
3.8. Diseño del Modelo CNN en el <i>Deep Network Designer</i> de MATLAB. . .	36
3.9. Matriz de confusión para los datos de validación con el modelo CNN. .	36
3.10. PCB antigua para el trabajo de reconocimiento de voz	37
3.11. PCB para driver de motores vibradores	38
3.12. PCB flexible para motores vibradores	38
3.13. Brazaletes con PCB que recogen las señales EMG y PCB principal que las procesa	38
3.14. Flujo de la comunicación I ² C entre el microcontrolador principal y los auxiliares para el trabajo de sensado EMG	39
3.15. Arquitectura de la red neuronal del microcontrolador principal	40
3.16. Matriz de confusión del entrenamiento de 4 clases con sensores EMG .	41
3.17. Prótesis antigua	41
3.18. Diagrama de Gantt de las fases del proyecto	43
3.19. Raspberry Pi Zero 2W	45
3.20. Modelo de cámara Raspberry utilizado (V2)	45
3.21. Motor DC de cada dedo	46
3.22. Driver de motores DC TLE94112ES	47
3.23. Conversor capacitivo-digital AD7147	47

3.24. IDE de STM	49
3.25. Logos de los lenguajes C y C++	49
3.26. Logo de Python	50
3.27. Logo de TensorFlow	50
3.28. Logo de OpenCV	51
4.1. Diagrama de clases para el control de motores usado en trabajo previo y reutilizado en el actual	58
4.2. Diagrama de clases implementado en la Raspberry para el manejo de cada subsistema en nuestro trabajo actual	59
4.3. Rendimiento del modelo de reconocimiento de voz en distintos microcontroladores STM32	62
4.4. Diagrama de clases para el control de sensores capacitivos desarrollado para el trabajo actual	64
4.5. <i>Dashboard</i> del flujo creado en STM32CubeMonitor	65
4.6. Clases de entrenamiento para subsistema EMG	66
4.7. Diagrama de flujo del subsistema de control de motores	67
4.8. Diagrama de flujo del subsistema de reconocimiento de voz	68
4.9. Diagrama de flujo del subsistema de sensado de tacto	69
4.10. Diagrama de flujo de la función llamada desde el programa principal para el reconocimiento de objetos	70
4.11. Diagrama de flujo del microcontrolador principal del subsistema EMG	71
4.12. Diagrama de flujo de cada microcontrolador auxiliar del subsistema EMG	72
4.13. Diagrama de flujo del sistema central	74
5.1. Arquitectura del sistema, donde se observan los 7 subsistemas: control de motores, sensado de tacto, reconocimiento de objetos, iluminación RGB, salida de audio, reconocimiento de voz y sensado EMG y realimentación de tacto. También aparece la forma de comunicación entre elementos, como el bus I ² C, los buses SPI y los buses I2S. Además, se puede apreciar cómo el subsistema de sensado EMG y realimentación de tacto usa un bus I ² C (I ² C2) para la comunicación con el sistema central, y otro bus I ² C (I ² C1) para la comunicación con sus microcontroladores auxiliares.	77
5.2. Formato de comunicación I ² C entre el microcontrolador de motores y la Raspberry	78
5.3. Formato de comunicación I ² C entre el microcontrolador de sensores de tacto y la Raspberry	79
5.4. Formato de comunicación I ² C entre el microcontrolador de reconocimiento de voz y la Raspberry	80
5.5. Formato de comunicación I ² C de la Raspberry al microcontrolador principal de sensado EMG	80
5.6. Formato de comunicación I ² C del microcontrolador principal de sensado EMG a la Raspberry	81
5.7. Interfaz disponible para el usuario de la prótesis. Contamos con 3 botones para cambiar de modo y una pantalla LCD que podrá ser usada en el futuro para visualización de datos o de manera lúdica.	82
5.8. Modo de bajo consumo <i>shutdown mode</i> de STM	85

5.9. Gráfica de precisión del entrenamiento y validación de la red neuronal del subsistema de sensado EMG	87
5.10. Gráfica de pérdida del entrenamiento y validación de la red neuronal del subsistema de sensado EMG	87
5.11. Matriz de confusión del entrenamiento del subsistema de sensado EMG	88
6.1. Función <i>update_status()</i> del subsistema de reconocimiento de objetos .	97
6.2. Función <i>getObjects()</i> del subsistema de reconocimiento de objetos . . .	97
6.3. <i>Resample</i> en el entrenamiento de la red neuronal para el subsistema EMG	98
6.4. Escalado en el entrenamiento de la red neuronal para el subsistema EMG	98
6.5. Características IEMG, MAV, SSI, RMS y VAR	100
6.6. Características MYOP, WL, DAMV, M2 y DVARV	101
6.7. Características DASDV, MAX, MIN y WAMP	102
6.8. Características IASD y IATD	103
6.9. Características IEAV, IALV y IE	104
6.10. Escalado realizado en el microcontrolador principal del subsistema EMG para los datos del amplificador 1	104
6.11. Función <i>update_devices()</i> del sistema central	106
6.12. Función <i>decide()</i> del sistema central	107
7.1. PCB integrada con forma de palma	110
7.2. Integración de todos los subsistemas con la PCB de la palma	111
7.3. Algunos tipos de agarre configurados	112
8.1. Programador ST-Link	118
8.2. Pruebas con distintas pesas para medir las variaciones de capacidad . .	120
8.3. Montaje para las pruebas unitarias del subsistema de control de motores	121
8.4. Impresión por pantalla de la prueba de motores	122
8.5. Montaje para las pruebas unitarias del subsistema de sensado de tacto	123
8.6. Salida de la función de prueba <i>testI2cMessage()</i>	123
8.7. Salida de la función de prueba <i>testHeightsInterrupt()</i>	124
8.8. Salida de la función de prueba <i>testI2cSensorControl()</i>	124
8.9. Montaje para las pruebas unitarias del subsistema de reconocimiento de voz. El micrófono está soldado a una PCB de pruebas fabricada por nosotros. El modelo de Raspberry usado es Zero W y no Zero 2W por ser con el que se contaba en ese momento.	125
8.10. Prueba del reconocimiento de 3 números: uno (0x11), dos (0x19) y tres (0x1d)	125
8.11. Montaje para las pruebas unitarias del subsistema de sensado EMG . .	126
8.12. Predicción errónea del sensado EMG	126
8.13. Predicción de la misma muestra durante el entrenamiento y en el microcontrolador	127
8.14. Montaje para las pruebas del subsistema de reconocimiento de objetos .	127
8.15. Prueba del subsistema de reconocimiento de objetos	128
8.16. Pruebas del aro RGB con distintos colores	129
8.17. Montaje para las pruebas de integración del subsistema de control de motores	130

8.18. Salida por terminal de Raspberry de la prueba de integración del subsistema de control de motores	131
8.19. Montaje para las pruebas de integración del subsistema de sensado de tacto	131
8.20. Salida por terminal de Raspberry de la prueba de integración del subsistema de sensado de tacto	132
8.21. Salida por terminal de microcontrolador de la prueba de integración del subsistema de sensado de tacto	132
8.22. Salida por terminal de Raspberry de la prueba de integración del subsistema de sensado de tacto	133
8.23. Salida por terminal de Raspberry de la prueba de integración del subsistema de sensado EMG	134
8.24. Prueba de integración del subsistema de reconocimiento de objetos . . .	134
8.25. Integración de todos los subsistemas con la PCB de la palma	136
8.26. Prueba del funcionamiento del Modo Voz integrado en el programa final	137
8.27. Prueba del funcionamiento del Modo Cámara integrado en el programa final. Reconocimiento de persona y movimiento asociado	138
8.28. Prueba de activación de la ISR correspondiente a tocar un sensor de tacto y parada del motor asociado	139
8.29. Prueba de la funcionalidad de agarre adaptativo con Modo Voz	139
A.1. Raspberry Pi Imager	160
A.2. Salida del <i>script</i> de instalación para manejar el altavoz	162
A.3. Programa de ejemplo para leer la UART de la Raspberry	166
A.4. Salida obtenida al programar correctamente un microcontrolador con OpenOCD	169
A.5. STM32CubeIDE añadiendo una nueva red neuronal con STM32Cube.AI	172
B.1. Botonera con 3 botones, cada uno para activar un modo	176

Índice de tablas

3.1. Tabla comparativa de diferentes familias de microcontroladores.	44
3.2. Riesgos del proyecto	54
4.1. Resultados del entrenamiento de los modelos de reconocimiento de voz evaluados en el proyecto previo.	60

Parte I

Prolegómeno

1. Introducción

A continuación, se describe la motivación del presente proyecto y su alcance. También se incluye un glosario de términos y la organización del resto de la presente documentación.

1.1. Motivación

El campo de prótesis robóticas de manos para adultos ha ido evolucionando con el tiempo hasta obtener, a día de hoy, bastantes modelos bien desarrollados. No obstante, esto cambia si nos fijamos en el caso de los niños. La funcionalidad y el número de articulaciones suele ser más reducido, así como la cantidad de modelos existentes. Tanto las dimensiones reducidas, como el riesgo de quedar obsoletas rápidamente por el crecimiento del usuario y el alto costo suponen un reto y hacen que exista esta falta de variedad y desarrollo [1]. Es por ello que en el laboratorio de Robótica Aplicada de la Universidad de Cádiz se ha trabajado en una prótesis de este tipo que aborde los citados problemas. Como resultado del trabajo previo, contamos con ciertas funcionalidades no integradas completamente en la actualidad, como el movimiento de los dedos mediante motores, visión artificial que es capaz de reconocer objetos, un sistema que aporta retroalimentación sobre el tacto mediante sensores capacitivos y micromotores, reconocimiento por voz y sensores EMG que permiten reconocer impulsos eléctricos en el antebrazo de una persona que tiene la mano amputada.

Sin embargo, al ser trabajo realizado por distintos estudiantes para distintos proyectos, son sistemas independientes que no han sido adaptados para trabajar en conjunto y proporcionar la funcionalidad de una mano real. Esto hace que exista una falta de comunicación entre los sistemas, como entre los sensores capacitivos y los motores, lo que provoca que no haya adaptabilidad en el movimiento de los dedos ni se paren cuando se aprieta demasiado. Por ejemplo, si se quiere coger un objeto cilíndrico como una taza, se podría tener un movimiento de los dedos a una posición predeterminada para coger esa taza de ese tamaño, pero si quieres agarrar un objeto cilíndrico de menor radio como una botella, la posición previa no es válida y, para el caso de una botella de radio mayor, se aprieta sin control.

De este modo, en el presente Trabajo de Fin de Grado se propone desarrollar una funcionalidad conjunta que integre el movimiento de motores, la visión artificial, el reconocimiento por voz, el sensado de EMG, así como el reconocimiento de tacto y su retroalimentación mediante vibración. Todo esto implica un rediseño, mejora y adaptación de cada uno de los citados sistemas, no solo para colaborar entre ellos, sino también debido a la nueva electrónica, ya que esta integración no será solamente de componentes software, sino también hardware, porque al igual que los proyectos tenían softwares independientes, también tenían hardware igualmente independiente. Esta nueva electrónica estará formada por nuevos microcontroladores de la familia

STM32, nuevas PCB y un nuevo diseño mecánico que permita integrar todo este hardware en una estructura única, aunque estas partes serán desarrolladas por compañeros especializados en estas áreas.

1.2. Alcance y Objetivos

El objetivo principal de este Trabajo de Fin de Grado consiste en la integración y adaptación de todas las partes software unitarias existentes de la prótesis de mano robótica en un único sistema, de tal manera que trabajen colaborativamente para conseguir emular el comportamiento de una mano real, así como otras funciones extras que permitan el uso de la mano de una manera lúdica para un niño que carece de ella.

Para lograr el objetivo principal se proponen los siguientes objetivos específicos:

1. Estudiar y analizar el hardware, software y arquitecturas existentes de la prótesis de mano robótica.
2. Diseñar e implementar una arquitectura adecuada capaz de gobernar todos los sistemas de una manera óptima.
3. Estudiar la programación de distintos microcontroladores STM32 mediante su IDE.
4. Testear y desarrollar la comunicación entre los microcontroladores STM32 y la Raspberry Pi (sistema central).
5. Programar los distintos microcontroladores desde la Raspberry Pi usando OpenOCD.
6. Adaptar el software para la lectura y configuración de los sensores capacitivos (tacto) de los dedos de la prótesis.
7. Integrar y adaptar la programación para el control de motores y sensores que permiten mover cada dedo de la mano.
8. Integrar el software para el reconocimiento de voz de la mano robótica.
9. Integrar la programación del sistema de visión artificial empleando la Raspberry Pi.
10. Integrar el software de control de los LEDs RGB y del sistema de salida de audio.
11. Comunicar la Raspberry Pi con el sistema EMG para reconocer movimientos de una mano amputada y enviar la retroalimentación del tacto.
12. Desarrollar el software de integración final para el funcionamiento de la prótesis de la mano robótica. Usar las funcionalidades de reconocimiento por voz y visión artificial para el uso lúdico de la prótesis mediante su movimiento y retroalimentación del tacto.

El alcance de este proyecto, diseñado para cumplir con todos los objetivos anteriores, es el siguiente:

1. Estudio de todos los trabajos que se han realizado previamente sobre la prótesis de mano robótica. Este estudio incluye la funcionalidad desarrollada y su arquitectura tanto hardware, como software.
2. Diseño e implementación de una arquitectura software adecuada capaz de gobernar todos los sistemas de una manera óptima, empleando la Raspberry Pi como elemento de control central.
3. Estudio de la familia de microcontroladores STM32 y los IDEs disponibles. Para poder desarrollar el nuevo software usando estos microcontroladores, será necesario aprender a programarlos. Esta programación se podrá realizar usando el IDE de Arduino con bibliotecas oficiales que permitan usar código Arduino o el propio IDE de STM.
4. Pruebas y desarrollo de la comunicación entre los STM32 y una Raspberry Pi (elemento central), y entre los STM32 y otros circuitos integrados. Estas comunicaciones se llevarán a cabo mediante los protocolos I²C y SPI.
5. Programar cada uno de los microcontroladores empleados desde la Raspberry Pi usando OpenOCD para así poder programar y acceder a todos los dispositivos desde un único elemento central.
6. Adaptación del software de lectura y configuración de los sensores capacitivos a la nueva arquitectura para así poder simular el tacto y parar los motores cuando se ejerza cierta presión.
7. Integración y adaptación del software que es capaz de mover y leer la posición de los motores usando un STM32 para funcionar según la información que le proporcione la Raspberry Pi.
8. Integración y adaptación del software de reconocimiento de voz usando un STM32 para que sea capaz de distinguir entre comandos para informar a la Raspberry Pi y que ésta mueva la mano en consecuencia.
9. Integración del software de reconocimiento de imágenes para poder mover la mano según los objetos que detecte.
10. Integración del software de control de los LEDs RGB y del sistema de salida de audio para el uso lúdico del niño al usar la prótesis.
11. Comunicación de la Raspberry Pi con un sistema de sensado de EMG para mover los dedos según los impulsos eléctricos del antebrazo y proporcionar la retroalimentación del tacto a los micromotores que se encuentran en este sistema.
12. Desarrollo del software en la Raspberry Pi que sea capaz de comunicarse con cada uno de los sistemas anteriores para que puedan trabajar de manera colaborativa en la movilidad y parada de la mano para poder conseguir un movimiento adaptativo en lugar de predefinido. Además, las funciones de reconocimiento de voz e imágenes permitirán un uso lúdico de la prótesis, fuera del uso normal de la mano con los sensores EMG.

1.3. Glosario de términos

En esta sección se explican los conceptos exclusivos al desarrollo del proyecto que pueden no ser triviales de cara a la comprensión del lector.

- **ADC (Analog to Digital Converter)**: Conversor analógico-digital. Convierte señales analógicas en valores digitales.
- **EMG (Electromyography)**: Técnica que mide la actividad eléctrica de los músculos.
- **Visión Artificial**: Campo de la inteligencia artificial que se enfoca en el desarrollo de sistemas que pueden interpretar y entender el mundo visual como los seres humanos.
- **OpenCV**: Biblioteca para visión por computador utilizada en robótica.
- **Raspberry Pi Zero 2W**: Ordenador embebido de bajo coste con CPU ARM de cuatro núcleos.
- **OpenOCD (Open On-Chip Debugger)**: Herramienta de software libre que se utiliza para programar y depurar microcontroladores y sistemas embebidos.
- **STM32**: Familia de microcontroladores de 32 bits de alto rendimiento, fabricados por STMicroelectronics.
- **I²C (Inter-Integrated Circuit)**: Protocolo de comunicación serie mediante bus compartido entre microcontroladores o entre microcontroladores y periféricos, como sensores, pantallas, etc.
- **SPI (Serial Peripheral Interface)**: Protocolo de comunicación serie punto a punto. Sirve para comunicar microcontroladores entre ellos o con periféricos.
- **MCU (MicroController Unit)**: Circuito integrado programable diseñado para controlar dispositivos electrónicos en sistemas embebidos.
- **Sistema embebido**: Sistema de cómputo integrado dentro de un dispositivo mayor, como un robot.
- **DMA (Direct Memory Access)**: Mecanismo que permite a ciertos periféricos transferir datos directamente a la memoria (y viceversa) sin intervención de la CPU o MCU.
- **Red neuronal**: Modelo computacional inspirado en el cerebro humano, compuesto por capas de nodos (neuronas artificiales) que procesan y aprenden patrones a partir de datos.
- **IDE (Integrated Development Environment)**: Aplicación que integra herramientas esenciales para programar, como editor de código, compilador, depurador y gestión de proyectos, todo en una sola interfaz.
- **MEMS (Micro-Electro-Mechanical Systems)**: Dispositivos miniaturizados que combinan componentes eléctricos y mecánicos en un solo chip. Se usan en sensores como acelerómetros, giroscopios y micrófonos.

- **I2S (Inter-IC Sound):** Protocolo de comunicación serial diseñado para transmitir audio digital entre componentes, como microcontroladores, DACs o codecs.
- **PCB (Printed Circuit Board):** Placa que sirve de soporte físico y conexión eléctrica para componentes electrónicos, usando pistas de cobre impresas.

1.4. Organización del documento

En este apartado se proporcionará una visión general de la estructura de este documento, describiendo el contenido de cada uno de los capítulos.

1. Introducción

En este capítulo se describe la motivación del proyecto y su alcance, incluyendo un glosario de términos empleados durante el mismo y la organización de la memoria.

2. Antecedentes

En este capítulo se da un contexto de las prótesis robóticas, primero a nivel general y luego hablando concretamente de las prótesis de niños, haciendo también un repaso de la evolución de las mismas. Asimismo, se estudian las posibles opciones tanto software como hardware que se tenían para desarrollar el proyecto.

3. Plan de Gestión del Proyecto

En este capítulo se habla de lo relacionado a la gestión del proyecto: metodología de desarrollo, planificación del proyecto, tecnologías, recursos y coste, riesgos y tipos de pruebas realizadas para el aseguramiento de la calidad.

4. Desarrollo

Esta es la parte central del documento, que abarca el análisis del sistema, su diseño, implementación, integración y las pruebas con su discusión. Aquí se detallan las decisiones, el funcionamiento del sistema tanto a nivel individual, como colaborativo, su arquitectura y las pruebas realizadas para validar su funcionamiento.

5. Epílogo

Esta es la parte final de la memoria, donde se habla de las conclusiones obtenidas, detallando los objetivos alcanzados, las lecciones aprendidas durante el desarrollo del proyecto y el trabajo futuro que se propone.

6. Anexos

En los Anexos, se incluyen los esquemáticos de cada uno de los elementos hardware que conforman la prótesis; el manual del desarrollador, con los pasos para llevar a cabo la réplica del proyecto y así poder extender o modificar el código fuente; y el manual de usuario, que aporta las instrucciones de uso de la prótesis y sus funciones por parte del usuario final de la misma.

2. Antecedentes

En este capítulo se pretende dar un primer acercamiento a las prótesis, tanto clásicas como robóticas, pasando por su evolución y explicando la situación actual de las mismas, principalmente de las prótesis de mano robótica para niños. Asimismo, se describirán las alternativas tecnológicas que existen en cuanto a sensórica, actuadores, control, algoritmos de reconocimiento, etc., para posteriormente elegir la que más nos convenga de cada una de ellas.

2.1. Contexto

En la actualidad, gran cantidad de personas sufren malformaciones o amputaciones de miembros superiores o inferiores del cuerpo. Esto puede ser producido por accidentes, enfermedades o problemas durante el embarazo. Las consecuencias son complicadas, ya que no solo se ve reducida la movilidad del afectado, sino también su calidad de vida, debido a que muchas de las acciones que podemos ver como cotidianas y normales, para ellos son realmente complejas.

Como solución a estos problemas surgieron las prótesis de extremidades. No obstante, la OMS estima que, actualmente, solo una de cada diez personas que necesitan ayudas técnicas prioritarias, incluidas las prótesis y ortesis, tienen acceso a ellas, debido a su alto costo y a la falta de conocimiento, disponibilidad, personal capacitado, políticas y financiación. Si de esas ayudas técnicas solo nos quedamos con las prótesis y ortesis, este número baja. Todo esto sabiendo que alrededor de 1 de cada 200 personas (un 0,5 % de la población mundial) las necesita [2].

Además, es conveniente saber que las manos que existen en el mercado son aparatos que no suelen diferenciar entre adultos y niños más allá del tamaño. Pero los niños no solo necesitan una prótesis robótica funcional que les permita desarrollar sus funciones básicas, sino que debe estar adaptada a las necesidades de la infancia. Por ello, quisimos desarrollar una mano que, además de la funcionalidad que se espera de ella, tuviera ciertos añadidos que hicieran que a los más pequeños les gustase tener una mano robótica. De este modo, conseguiríamos que se adaptasen a ella lo más rápido posible y podríamos evitar que se sintiesen acomplejados.

2.1.1. Introducción a las prótesis robóticas actuales

Las prótesis de mano han protagonizado una de las transformaciones más significativas en la unión entre medicina e ingeniería. Han evolucionado desde ser herramientas pasivas y meramente funcionales a convertirse en sofisticadas extensiones biónicas del cuerpo humano. El objetivo actual ya no es solo sustituir una extremidad, sino restaurar la funcionalidad motora y sensorial de la forma más natural e intuitiva posible. En esta sección se explora el panorama actual de las prótesis de mano robóticas.

Sistemas de control avanzados

La interfaz entre el usuario y la prótesis es el pilar fundamental de su funcionalidad. Aunque el control mioeléctrico (EMG), que decodifica las señales eléctricas de los músculos del muñón, sigue siendo el estándar, la tecnología ha evolucionado para ofrecer un control más sofisticado.

- **Reconocimiento de patrones:** En lugar de usar señales musculares simples para abrir o cerrar la mano, los sistemas modernos emplean algoritmos de aprendizaje automático (*machine learning*). Estos algoritmos se entrenan para reconocer patrones complejos de contracción muscular, permitiendo al usuario acceder a un repertorio de agarres (pinza fina, agarre de potencia, etc.) de manera mucho más intuitiva. La investigación reciente se centra en arquitecturas de redes neuronales cada vez más robustas para mejorar la precisión en tareas cotidianas [3].
- **Reinervación Muscular Específica (TMR):** Considerado uno de los mayores avances quirúrgicos en este campo, el procedimiento de TMR (*Targeted Muscle Reinnervation*) redirige los nervios que originalmente controlaban la mano y los dedos hacia músculos residuales en el muñón o en el pecho. Al contraer estos músculos reinervados, el usuario genera señales EMG que se corresponden directamente con los movimientos de la mano amputada, logrando un control simultáneo y más natural de las articulaciones de la prótesis [4].
- **Interfaces neuronales y sensores implantados:** La frontera de la investigación se sitúa en la comunicación directa con el sistema nervioso. Los sensores mioeléctricos implantables (IMES) se colocan directamente en el músculo para captar señales con mayor fidelidad y menos ruido que los sensores de superficie. A un nivel más avanzado, las interfaces neuronales que se conectan directamente a los nervios periféricos o incluso al cerebro (Interfaces Cerebro-Computadora o BCI) prometen un futuro donde el control de la prótesis sea indistinguible del de una extremidad biológica [5].

Mecánica y actuación

La destreza de una prótesis viene determinada por su diseño mecánico. La mano humana, con más de 20 grados de libertad, es un prodigio de la ingeniería biológica difícil de replicar.

- **Grados de libertad (DoF) y actuadores:** Las prótesis comerciales varían desde 1 o 2 DoF (apertura/cierre y rotación de muñeca) hasta más de 10 DoF en manos multiarticuladas como las de Bebionic (Figura 2.1) o i-Limb (Figura 2.2), que permiten el control individual de los dedos. El reto es integrar los motores, generalmente de corriente continua (DC), y las transmisiones en un formato compacto y ligero.
- **Subactuación (*Underactuation*):** Para simplificar el control y reducir el peso, muchos diseños avanzados emplean mecanismos de subactuación. Con este enfoque, un solo motor puede controlar múltiples articulaciones (por ejemplo, las falanges de un dedo). Esto permite que los dedos se adapten pasivamente a la

forma de los objetos, logrando un agarre estable y seguro con un control mucho más simple [6] [7].

- **Materiales y fabricación:** La impresión 3D ha revolucionado especialmente el campo de las prótesis pediátricas, permitiendo la creación de dispositivos personalizados, ligeros y de bajo coste que pueden reemplazarse fácilmente a medida que el niño crece [8]. En las prótesis de gama alta, se utilizan materiales como la fibra de carbono y aleaciones de titanio para maximizar la resistencia y minimizar el peso.



Figura 2.1: Mano robótica BeBionic



Figura 2.2: Mano robótica i-Limb

Retroalimentación sensorial

La capacidad de sentir con la prótesis es crucial para un control preciso y para la sensación de que la prótesis es parte del propio cuerpo.

- **Retroalimentación táctil:** Sensores de presión en los dedos pueden detectar el contacto y la fuerza ejercida sobre un objeto. Esta información se transmite al usuario mediante métodos no invasivos, como vibraciones (retroalimentación vibrotáctil) o presión sobre la piel (retroalimentación mecánica) [9].
- **Restauración sensorial directa:** El enfoque más avanzado, como se mencionó anteriormente, es la estimulación eléctrica de los nervios residuales en el muñón. Esto puede evocar sensaciones de tacto y presión que el usuario percibe como provenientes directamente de su mano “fantasma” [10].

Factores prácticos

El éxito de una prótesis no solo depende de su tecnología, sino también de factores prácticos y humanos.

- **Energía y autonomía:** La duración de la batería es una de las principales preocupaciones de los usuarios. La mayoría de las prótesis actuales ofrecen autonomía para un día de uso, como es el caso de la mano BeBionic, pero la gestión de la energía sigue siendo un desafío de diseño crítico.
- **Peso y durabilidad:** Existe un compromiso constante entre añadir más funcionalidad (y por tanto más motores y peso) y la comodidad del usuario. Una prótesis pesada puede causar fatiga y ser abandonada. Además, debe ser lo suficientemente robusta para soportar los rigores del uso diario.
- **Psicología y estética (Cosmesis):** La aceptación de la prótesis por parte del usuario está íntimamente ligada a su apariencia. Mientras que algunos usuarios prefieren un guante cosmético que imita una mano real, otros adoptan con orgullo la estética robótica y futurista de sus dispositivos. La posibilidad de personalizar el diseño es cada vez más importante.

2.1.2. Evolución histórica de las prótesis robóticas

Las primeras prótesis que se conocen son dos dedos de los pies provenientes del antiguo Egipto, en concreto, antes del 600 a.C.(véase Figura 2.3). A día de hoy, no se sabe con seguridad si tenían un propósito funcional o meramente estético, aunque hay ciertos estudios que indican que podrían haber llegado a ayudar a las personas a las cuales se les implantaron. [11].

Con el tiempo, empezaron a mejorar tanto los materiales con los que se fabricaban como su funcionamiento. Las anteriormente mencionadas egipcias estaban fabricadas con madera y piel. Luego, en tiempos de las civilizaciones griega y romana, las prótesis de extremidades inferiores se realizaban normalmente de placas de bronce con un núcleo de madera y correas de cuero [12]. Estos materiales fueron los que se siguieron usando



Figura 2.3: Primera prótesis conocida

hasta el siglo XX, aunque poco a poco, las prótesis iban siendo más funcionales. Entre el siglo XV y el XIX, en Francia y Suiza se fabricaron extremidades que tenían más capacidad móvil usando cables, engranajes, palancas y resortes. No obstante, los ajustes a estos dispositivos debían realizarse manualmente. Por ejemplo, para cerrar una mano artificial se necesitaba usar un tenedor a modo de palanca y la mano que tuviese libre para manejar dicha palanca. Posteriormente, se empezaron a construir más prótesis funcionales al usar plásticos y otros materiales artificiales de manera que cada vez iban siendo más livianas, duraderas y se ajustaban mejor a las extremidades [13].

Aun así, la verdadera mejora llegó cuando empezaron a motorizarse, siendo las prótesis de manos las más afectadas, al ser las extremidades que más tipos de movimientos tienen y unas de las más útiles en el día a día. La primera de la que se tiene conocimiento es una mano neumática patentada en Alemania en 1915 (véanse Figuras 2.4 y 2.5). La utilización de extremidades motorizadas probablemente no se llevó a cabo de manera significativa entre las dos guerras mundiales, pero extremidades impulsadas por CO₂ fueron utilizadas por Weil ya en 1948. Luego, hubo un avance significativo al aparecer las prótesis mioeléctricas, que son aquellas controladas por contracciones musculares, siendo la primera desarrollada durante los años 40 por Reinhold Reiter (Figura 2.6). Esta no fue muy conocida y el control mioeléctrico no se redescubrió hasta los años 50.

Durante el período de la posguerra, se desarrolló la mano eléctrica de Vaduz (Figura 2.7). Esta mano empleaba una bolsa neumática que detectaba presión muscular mediante presión neumática, lo cual activaba un servomecanismo para cerrarla. Por esto, debe ser considerada uno de los antecesores más importantes de las manos eléctricas. Esta mano fue seguida por una mano inglesa desarrollada alrededor de 1965 por Bottomley, la cual fue la primera mano mioeléctrica que mostraba un control proporcional de la fuerza (Figura 2.8). Sin embargo, el problema de todas ellas era que eran dispositivos de una única función y no adaptativos. De este modo, también en 1965, se empezó a trabajar en la mano de SVEN, que arregló estos problemas, siendo una mano eléctrica adaptativa y con múltiples funciones (Figura 2.9). Además, la ventaja de la energía eléctrica frente al gas comprimido es que se podía almacenar de manera

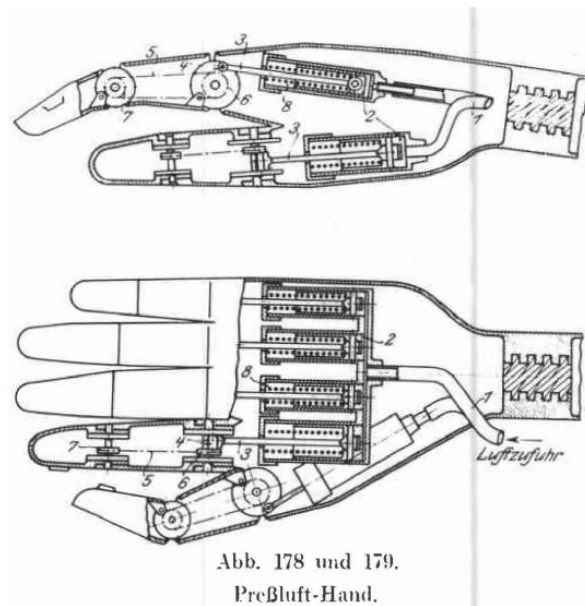


Figura 2.4: Mano neumática. Posiblemente el primer componente protésico motorizado

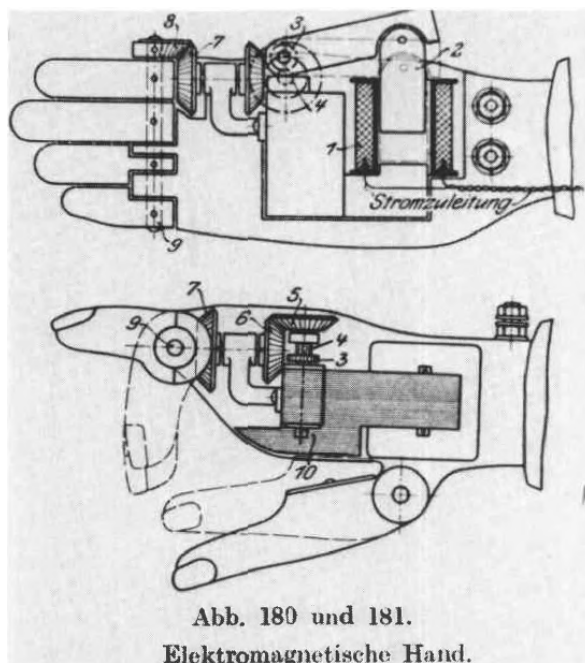


Figura 2.5: Componente de mano eléctrica. Quizás el primer mecanismo eléctrico de mano

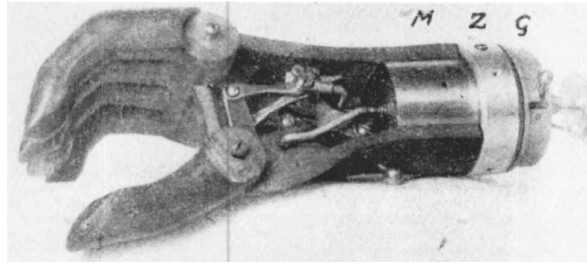


Figura 2.6: Mano eléctrica usada por Reiter en el desarrollo de la primera prótesis mioeléctrica (Circa 1943)

más barata y segura. Y no solo eso, sino que los circuitos electrónicos permitían más posibilidades de control, como movimiento de la muñeca, del codo, etc. Por otro lado, en las manos eléctricas y mioeléctricas, toda la parte electrónica y las baterías iban fuera de las prótesis, así como los cables, por lo que eran bastante incómodas y se producían interferencias [14].

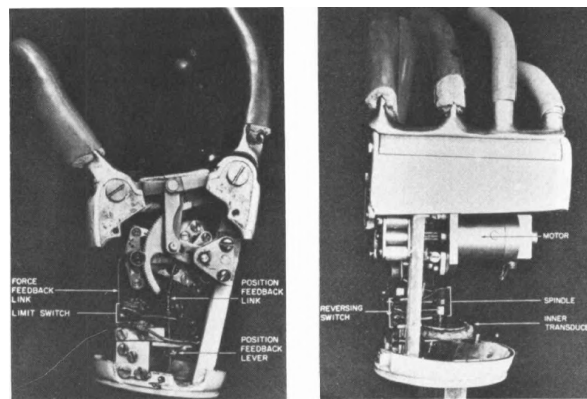


Figura 2.7: Dos vistas de la mano de Vaduz

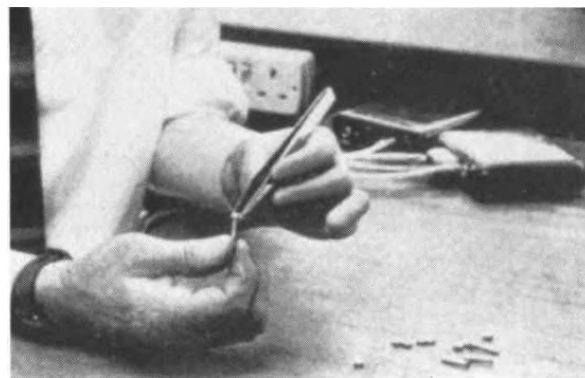


Figura 2.8: Mano de Bottommley

Aun así, continuó la investigación de las manos electrónicas y estas fueron evolucionando. Primero reduciendo el tamaño de los componentes electrónicos y baterías y, por tanto, el peso de las prótesis, lo que aumentó su manejo y comodidad. Posteriormente, incluyendo el campo de la informática al comenzar a programar estas manos de tal modo que pudieran realizar más variedad de movimientos y de manera más controlada a los deseos del amputado, sin que requiriesen mucha carga mental para él.

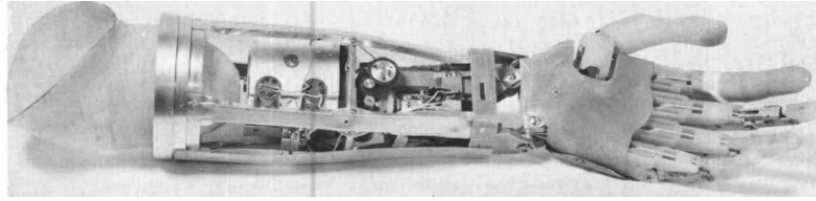


Figura 2.9: Mano de Sven

Todas estas mejoras y, en especial, la comodidad y facilidad de manejo han permitido que sea más sencillo el desarrollo de prótesis de manos robóticas que se puedan integrar de manera más fácil en la vida cotidiana del afectado.

2.1.3. Prótesis robóticas de manos para niños

El panorama actual de las prótesis de mano robóticas para niños se encuentra en un punto de inflexión, impulsado en gran medida por la democratización de la tecnología de impresión 3D. Esta revolución ha abordado con éxito uno de los mayores desafíos históricos: el prohibitivo coste y la necesidad de reemplazar los dispositivos a medida que el niño crece. Gracias a comunidades globales de código abierto y a productos comerciales innovadores, ahora es posible crear prótesis ligeras, asequibles y, fundamentalmente, personalizadas. Este nivel de personalización va más allá del simple ajuste anatómico; permite a los niños elegir colores y diseños, transformando su prótesis en un accesorio de empoderamiento y orgullo, lo cual es un factor clave para su aceptación psicológica y social.

Para ilustrar este ecosistema, tres ejemplos destacan por sus diferentes filosofías. La *Hero Arm* de *Open Bionics* (Figura 2.10) representa la vanguardia de la biónica comercial para niños; es una mano mioeléctrica multiagarre, impresa en 3D para un ajuste perfecto y famosa por sus cubiertas intercambiables de superhéroes que fomentan la confianza. En el espectro del código abierto, la comunidad e-NABLE ofrece diseños mecánicos y accionados por el propio cuerpo, como la *Phoenix Hand* (Figura 2.11), que son ultraligeros, robustos y prácticamente gratuitos, eliminando la barrera económica. Por su parte, el fabricante Ottobock ofrece con su *System Electric Hand 2000* (Figura 2.12) el estándar clínico: una prótesis mioeléctrica centrada en la máxima durabilidad y fiabilidad de un único y potente agarre, a menudo cubierta con un guante cosmético para una apariencia realista.

A la par de los avances en fabricación, los sistemas de control también se han adaptado al público infantil, centrándose en la intuición y el compromiso del usuario. Las soluciones mioeléctricas, que leen las señales musculares del brazo, se han simplificado y combinado con plataformas de entrenamiento basadas en la gamificación. Mediante videojuegos y aplicaciones interactivas, los niños pueden aprender a modular sus músculos de una forma lúdica y efectiva, reduciendo la frustración y acelerando el dominio de su prótesis. Sin embargo, a pesar de estos notables progresos, la funcionalidad de la mayoría de los dispositivos actuales, tanto comerciales como de código abierto, se limita a una serie de patrones de agarre preprogramados, lo que restringe la naturalidad de la interacción con objetos de formas y tamaños diversos.



Figura 2.10: *Hero ARM* de *Open Bionics*

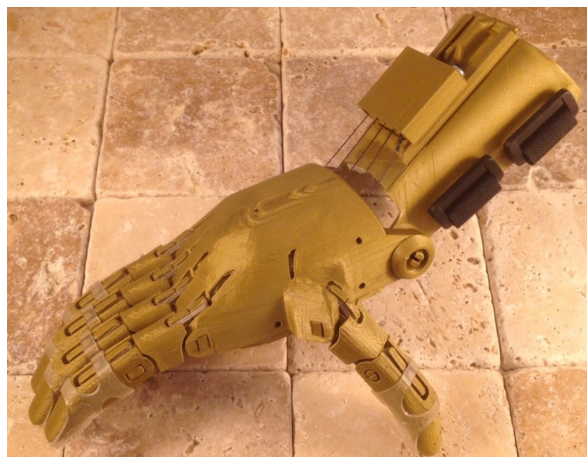


Figura 2.11: *Phoenix Hand V2* de e-NABLE



Figura 2.12: *System Electric Hand 2000* de Ottobock

Es precisamente en este contexto donde nuestro proyecto introduce una innovación sustancial. Mientras que el estándar actual se basa en que el usuario seleccione un tipo de agarre específico, nuestra prótesis incorpora un sistema de agarre adaptativo. Esto permite que la mano se ajuste de forma inteligente y automática a la geometría del objeto que está sujetando, logrando una manipulación mucho más versátil y fluida, similar a la de una mano humana. Esta capacidad representa un salto cualitativo respecto a los sistemas existentes.

Además, abordamos la que es considerada la mayor frontera pendiente en el campo de la prótesis pediátrica: la ausencia total de retroalimentación sensorial. Los niños que utilizan las prótesis actuales deben depender constantemente de la vista para controlar la fuerza que aplican, lo que dificulta tareas delicadas. Nuestro trabajo da un paso pionero al integrar un sistema de realimentación del tacto, comunicando al niño la información de presión que detectan los sensores de la mano. Esta reintroducción de una forma de "sentido del tacto" busca fomentar una conexión más profunda con el dispositivo, llevando la funcionalidad y la experiencia de usuario de las prótesis infantiles a un nivel que, hasta ahora, estaba confinado a la investigación avanzada para adultos.

2.2. Estado de la técnica

2.2.1. Sensórica

Sensado de tacto

El sentido del tacto artificial es fundamental para tareas como el agarre de objetos frágiles, la manipulación diestra y la interacción hombre-máquina. La información clave que un sensor de tacto debe proporcionar incluye la detección de contacto, la medición de la fuerza aplicada y, en sistemas más avanzados, la distribución de la presión a lo largo de una superficie.

A continuación, se describen las tecnologías más consolidadas y prometedoras para la implementación del sensado de tacto en prótesis y manos robóticas:

- **Sensores resistivos (FSR):** Un FSR (Figura 2.13) está formado por dos capas separadas por un espaciador. Una de las capas está recubierta con un material conductor y la otra con una tinta resistiva especial. Al aplicar presión, las dos capas entran en contacto. Cuanta más fuerza se aplica, mayor es el área de contacto entre las capas, lo que provoca una disminución de la resistencia eléctrica del dispositivo. Midiendo esta resistencia, se puede inferir la fuerza aplicada. Esta característica los hace extremadamente fáciles de integrar en circuitería simple. Gracias a su construcción fina y flexible, son ideales para recubrir superficies curvas como los dedos de una prótesis. Sin embargo, su principal inconveniente es su baja precisión y repetibilidad, lo que los hace más adecuados para una detección cualitativa de la fuerza (poca, media o mucha) que para mediciones de alta exactitud, además de sufrir un desgaste progresivo con el uso continuado.

- **Sensores capacitivos:** Estos sensores (Figura 2.14) operan como un condensador variable, donde una fuerza externa comprime un material dieléctrico elástico, reduciendo la distancia entre dos electrodos conductores. Esta proximidad incrementa la capacitancia del sistema de una manera predecible y medible. Su alta sensibilidad, especialmente a fuerzas muy bajas, y su bajo consumo los hacen idóneos para la construcción de matrices de alta resolución, conocidas como “pieles táctiles”, capaces de mapear la distribución de la presión. A pesar de sus ventajas en rendimiento, son más susceptibles al ruido electromagnético y requieren una circuitería de acondicionamiento más compleja que sus contrapartes resistivas.

Para solventar este desafío de la complejidad en el acondicionamiento de la señal, los fabricantes de microcontroladores han integrado periféricos dedicados. Un ejemplo destacado es la tecnología *Touch Sensing Controller* (TSC) de ST-Microelectronics, presente en su familia de microcontroladores STM32. Este no es un sensor en sí mismo, sino un bloque de *hardware* especializado que gestiona de forma autónoma todo el proceso de medición de la capacitancia. El TSC se encarga de cargar y descargar los electrodos y medir los tiempos de transferencia de carga, que son proporcionales a la capacitancia del sensor.

- **Sensores piezoeléctricos y piezoresistivos:** Son sensores basados en materiales "inteligentes" que alteran sus propiedades eléctricas bajo deformación mecánica. Los sensores piezoeléctricos, por ejemplo, generan una pequeña tensión eléctrica en respuesta a una fuerza dinámica o vibración, lo que los hace excelentes para detectar el instante de un impacto o la textura de una superficie, aunque son incapaces de medir una fuerza estática y constante. Por otro lado, los sensores piezorresistivos, fabricados comúnmente con semiconductores como el silicio, cambian su resistencia interna de forma muy precisa y lineal con la deformación. Esta propiedad les confiere una gran exactitud y fiabilidad, siendo el estándar en aplicaciones de alta gama, si bien su coste es mayor y su naturaleza más rígida puede complicar su integración en superficies flexibles.
- **Sensores ópticos y magnéticos:** Estos dos tipos de sensores son enfoques innovadores que van más allá de las tecnologías consolidadas anteriores. Los sensores ópticos, como TacTip, utilizan cámaras internas para analizar la deformación de una piel de silicona, obteniendo un mapa de altísima resolución de las fuerzas de contacto, incluyendo la dirección y el cizallamiento. Por su parte, los sensores magnéticos miden la fuerza a través del desplazamiento de un pequeño imán suspendido en un elastómero, detectando los cambios en el campo magnético.

Captura de imágenes

La implementación de la visión artificial en sistemas robóticos portátiles, como una prótesis de mano, impone una serie de restricciones críticas en la elección del hardware de captura de imagen. El componente seleccionado no solo debe ofrecer una calidad de imagen adecuada para los algoritmos de detección, sino que también debe cumplir con estrictos requisitos de tamaño, peso, consumo energético e interfaz de comunicación.



Figura 2.13: Sensor resistivo

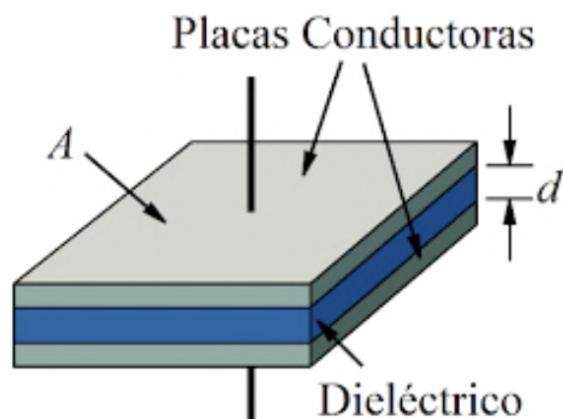


Figura 2.14: Sensor capacitivo

Una de las soluciones más extendidas y documentadas es la línea oficial de módulos de cámara de Raspberry Pi. El Módulo de Cámara V2 (*Camera Module V2*), basado en el sensor Sony IMX219 de 8 megapíxeles, se consolidó durante años como el estándar de facto por su equilibrio entre coste, calidad y una perfecta integración nativa. Se conecta directamente al puerto MIPI CSI-2 (*Camera Serial Interface 2*) de la Raspberry Pi, una interfaz de alta velocidad diseñada específicamente para el streaming de vídeo. Este tipo de conexión permite que los datos de la imagen sean procesados directamente por la GPU (Unidad de Procesamiento Gráfico) de la placa, liberando a la CPU de una carga de trabajo significativa. Esto se traduce en un mayor rendimiento y una menor latencia.

La evolución de esta línea ha dado lugar al Módulo de Cámara V3 (*Camera Module 3*) visto en la Figura 2.15, que no solo mejora la resolución con un sensor de 12 megapíxeles (Sony IMX708), sino que incorpora una característica fundamental para la robótica de manipulación: el autoenfoque.

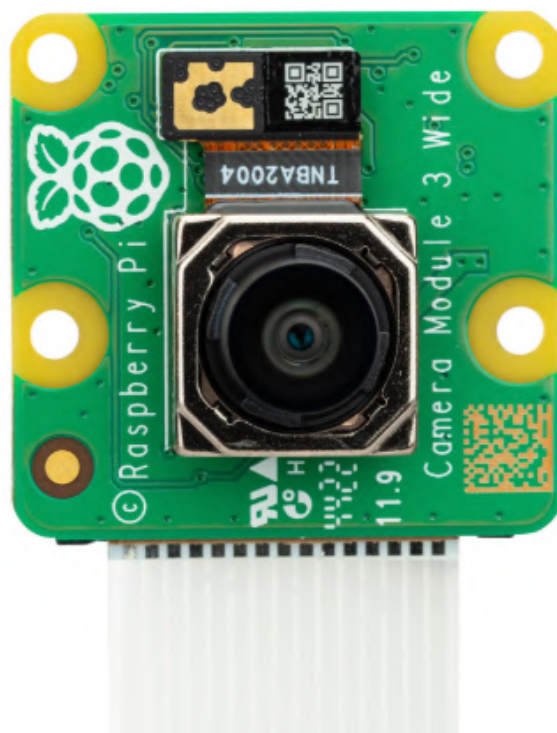


Figura 2.15: Raspberry Pi Camera Module 3

Más allá de los módulos oficiales, el mercado ofrece una amplia gama de alternativas. Fabricantes como Arducam se han especializado en desarrollar módulos de cámara compatibles con la interfaz CSI-2 que ofrecen una mayor versatilidad. Proporcionan sensores con diferentes características (mayor resolución, mejor rendimiento en baja luz, obturador global para objetos en rápido movimiento) y, lo que es más importante, una gran variedad de lentes intercambiables (M12/S-Mount), que permiten adaptar la óptica a las necesidades específicas de la aplicación.

Finalmente, una alternativa común por su simplicidad de conectar y usar son las cámaras web USB. Su principal ventaja es la universalidad; se pueden conectar a prácticamente cualquier sistema de procesamiento, incluyendo la Raspberry Pi, sin

necesidad de controladores específicos. Sin embargo, esta simplicidad tiene un coste en rendimiento. La interfaz USB introduce una mayor latencia y, fundamentalmente, aumenta la carga sobre la CPU, ya que el procesador principal debe gestionar la descompresión y el manejo del flujo de vídeo.

Sensores de posición para motores

La capacidad de una prótesis de mano robótica para ejecutar movimientos precisos y controlados, así como para mantener una posición de agarre estable, depende de forma crítica de su sistema de sensado de posición.

Una de las soluciones más tradicionales y de bajo coste para medir la posición angular es el potenciómetro rotatorio. Su principio de funcionamiento se basa en un cursor o contacto móvil que, conectado al eje del motor, se desliza sobre una pista resistiva circular. Al girar el eje, la resistencia eléctrica cambia, generando una señal de voltaje proporcional al ángulo. La gran ventaja de los potenciómetros es que proporcionan una posición absoluta, es decir, el sistema conoce la posición del motor incluso después de un reinicio. Sin embargo, su naturaleza mecánica es también su mayor debilidad; el contacto físico constante provoca un desgaste que degrada la señal con el tiempo, introduce ruido y limita su vida útil, haciéndolos poco adecuados para aplicaciones que exigen una alta durabilidad y ciclos de trabajo intensivos.

Para superar estas limitaciones, la robótica moderna se apoya mayoritariamente en los codificadores rotatorios, o encoders, que ofrecen una precisión y fiabilidad muy superiores. Los codificadores incrementales son una opción muy popular, ya que generan una serie de pulsos eléctricos a medida que el eje gira, permitiendo al controlador calcular el cambio de posición relativo contando dichos pulsos. Al no tener contacto mecánico, no sufren desgaste. Su principal inconveniente es la pérdida de la referencia de posición si el sistema se apaga, lo que obliga a realizar un ciclo de calibración al arrancar. Para solucionar esto, existen los codificadores absolutos, una variante más avanzada que utiliza un disco con un patrón único para cada ángulo. Esto permite al sensor conocer la posición exacta en todo momento, incluso tras un fallo de alimentación, pero a costa de un mayor precio y una electrónica de lectura más compleja.

2.2.2. Actuadores

La elección del actuador, es una de las decisiones de diseño más críticas en la robótica de manipulación.

Por un lado, las soluciones de alto rendimiento como los motores DC sin escobillas (BLDC) representan la vanguardia tecnológica, ofreciendo una eficiencia, durabilidad y relación potencia-peso excepcionales, siendo una de las mejores elecciones para las prótesis comerciales de gama alta. Sin embargo, su elevado coste y la complejidad de sus controladores electrónicos a menudo los reservan para productos con presupuestos de producción industrial.

En una línea muy diferente, los servomotores se han popularizado enormemente por su conveniencia, al empaquetar en una sola unidad el motor, la reductora y la

electrónica de control. Esta naturaleza agiliza enormemente el prototipado, aunque a menudo implica aceptar los componentes internos que incluye, como los potenciómetros, que pueden limitar la durabilidad y la flexibilidad en el control fino.

Frente a estas opciones, los motores DC con reductora y escobillas se presentan como un componente fundamental que ofrece un alto grado de libertad en el diseño. Su arquitectura modular, que separa el motor del sensor y del circuito de control, permite una notable flexibilidad, ya que es posible combinar un motor económico con un sensor de alta precisión y desarrollar un algoritmo a medida. Sin embargo, esta versatilidad conlleva sus propias desventajas. La principal es la mayor complejidad de integración, ya que recae en el diseñador la responsabilidad de seleccionar, acoplar y programar estos tres sistemas por separado. Además, su componente clave, las escobillas, sufren un desgaste mecánico por el contacto continuo, lo que limita su vida útil en comparación con las tecnologías sin escobillas y puede generar ruido eléctrico. A pesar de ello, estos inconvenientes son a menudo un compromiso asumible a cambio de la total personalización y el bajo coste que ofrecen para proyectos de investigación y desarrollo.

2.2.3. Control

La arquitectura de control de un sistema robótico complejo, como una prótesis biónica, se organiza jerárquicamente. Por un lado, los microcontroladores actúan como los nervios del sistema, gestionando las tareas más directas y rápidas. Por otro, un sistema central funciona como el cerebro principal, encargado de la toma de decisiones, la ejecución de algoritmos complejos y la coordinación general. Asimismo, estos elementos interactúan con circuitos integrados que permiten el control de muchos elementos, como motores o sensores.

Microcontroladores

- **AVR (Plataforma Arduino):** La plataforma Arduino, basada principalmente en los microcontroladores AVR de Atmel, es el estándar de facto para el prototipado rápido. Su principal fortaleza no es la potencia, sino su inmensa facilidad de uso gracias a un entorno de programación simplificado y miles de bibliotecas disponibles. Para un proyecto de investigación, Arduino es una herramienta excelente en las fases iniciales para validar conceptos de forma rápida.
- **ESP32:** Los ESP32 han revolucionado el campo del Internet de las Cosas y la robótica conectada. Su principal ventaja es la conectividad integrada: incluyen de serie Wi-Fi y Bluetooth en el propio chip. Además, cuentan con un procesador de doble núcleo, mucha más memoria que un Arduino y un bajo consumo, lo que los hace ideales para nodos de control que necesitan comunicarse de forma inalámbrica con el sistema central u otros dispositivos.
- **STM32:** basados en la arquitectura ARM Cortex-M, representan el estándar en la industria y en aplicaciones de alto rendimiento. Ofrecen una gama de prestaciones muy superior a las familias anteriores: mayor velocidad de procesamiento, periféricos avanzados (convertidores ADC/DAC de alta resolución, temporizadores

de control de motor, etc.) y una robustez de nivel industrial. Aunque su curva de aprendizaje es más pronunciada, son la opción preferida cuando se requiere un control de motores preciso o un procesamiento de señales de sensores complejo.

Sistema central

- **Raspberry Pi:** La Raspberry Pi es el ordenador de placa única (SBC) más popular del mundo. Modelos como la Raspberry Pi 4 o la Pi 5 ofrecen una potencia de cálculo comparable a la de un ordenador de escritorio de hace unos años en un formato diminuto. Su fortaleza reside en su versatilidad: puede ejecutar algoritmos de visión artificial con librerías como OpenCV, gestionar la comunicación entre todos los microcontroladores y servir como punto de acceso para la configuración o el control remoto. Su enorme comunidad de soporte y su bajo coste la convierten en la opción por defecto para la unidad central de la gran mayoría de proyectos robóticos de investigación y desarrollo.
- **NVIDIA Jetson:** Cuando la visión artificial y la inteligencia artificial son el núcleo del proyecto y se requiere un rendimiento muy superior, la familia NVIDIA Jetson es la elección profesional. Placas como el Jetson Nano o las más potentes de la serie Orin integran una potente GPU (Unidad de Procesamiento Gráfico) de NVIDIA en la propia placa. Esto les permite ejecutar redes neuronales complejas y algoritmos de visión en paralelo y en tiempo real con una eficiencia que una Raspberry Pi no puede alcanzar.

2.2.4. Algoritmos

Reconocimiento de voz

Para dotar a la prótesis de una modalidad de control por voz intuitiva y de bajo consumo, se ha optado por implementar un sistema de reconocimiento de palabras clave (*Keyword Spotting* KWS). Para llevar a cabo esta tarea de detección de patrones en el audio, se ha decidido emplear un modelo de Deep Learning, ya que las arquitecturas de redes neuronales han demostrado ser la solución más precisa y robusta para este problema [15]. A continuación, se analizan las distintas familias de modelos que la investigación actual ha considerado para esta aplicación.

- **Preprocesamiento:** Antes de que cualquier modelo pueda escuchar, la señal de audio en bruto debe ser transformada en una representación que una red neuronal pueda interpretar. La técnica más extendida y eficaz es la extracción de coeficientes cepstrales en la escala de Mel (MFCCs) [16]. Este proceso convierte la onda de sonido en un espectrograma que muestra cómo cambia la energía de las diferentes frecuencias de sonido a lo largo del tiempo y es el dato de entrada para la mayoría de los modelos KWS, ya que permite a las redes neuronales, especialmente a las convolucionales, encontrar patrones característicos en el habla.
- **Arquitecturas Basadas en Redes Neuronales Convolucionales (CNN):** Inspiradas en el campo de la visión artificial, las CNN se han adaptado al KWS.

Tratan el espectrograma de audio como si fuera una imagen y utilizan filtros convolucionales para detectar patrones. Una sílaba o un fonema puede ser visto como una característica de bajo nivel, y la combinación de estos en el tiempo forma la palabra clave, un patrón de alto nivel. Modelos como *TC-ResNet* o arquitecturas diseñadas específicamente por Google han demostrado ser muy eficientes [17]. La innovación clave para su uso en microcontroladores fue la introducción de las redes convolucionales separables en profundidad (*Depthwise Separable Convolutional Neural Networks* o DS-CNN). Esta arquitectura reduce drásticamente el número de parámetros y operaciones computacionales en comparación con una CNN estándar, logrando una precisión muy alta con un tamaño de modelo lo suficientemente pequeño como para caber en la memoria de un microcontrolador [18].

- **Arquitecturas Basadas en Redes Neuronales Recurrentes (RNN):** A diferencia de las CNN, que analizan el espacio del espectrograma, las RNN están diseñadas explícitamente para procesar secuencias de datos a lo largo del tiempo. Arquitecturas como las LSTM (*Long Short-Term Memory*) o las GRU (*Gated Recurrent Unit*) tienen una "memoria interna que les permite recordar información de pasos de tiempo anteriores. Esta capacidad para modelar el contexto temporal es teóricamente ideal para el habla. En la práctica, aunque pueden alcanzar una alta precisión, las RNN puras suelen ser computacionalmente más pesadas y más lentas de entrenar que las CNN optimizadas. Por ello, en el ámbito de KWS para sistemas embebidos, a menudo se utilizan en arquitecturas híbridas CRNN (*Convolutional Recurrent Neural Network*), donde una capa convolucional inicial extrae las características del audio y una capa recurrente final modela la secuencia temporal de dichas características.

Reconocimiento de imágenes

La visión artificial se ha consolidado en los últimos años como una de las áreas más dinámicas dentro de la inteligencia artificial, encontrando aplicaciones en sectores tan diversos como la robótica, la medicina, la automoción o la seguridad. En concreto, se ha llegado a aplicar al campo de las prótesis robóticas, llegando a mejorar el control de las mismas para agarrar distintos objetos [19]. Su desarrollo se ha visto favorecido por la disponibilidad de bibliotecas y plataformas que simplifican la implementación de algoritmos complejos, así como por el acceso a modelos de aprendizaje profundo previamente entrenados.

Entre las herramientas más relevantes se encuentra OpenCV [20], una de las bibliotecas más utilizadas en el ámbito de la visión artificial debido a su carácter abierto, multiplataforma y su amplia comunidad de desarrolladores. Esta biblioteca proporciona funciones para el procesamiento de imágenes, la detección de objetos mediante clasificadores en cascada, el reconocimiento facial o el seguimiento de movimiento. En sus versiones más recientes incorpora además el módulo *dnn* [21], que permite integrar modelos de redes neuronales profundas preentrenadas en frameworks como TensorFlow, Caffe, ONNX o Darknet, lo que facilita la detección de objetos con arquitecturas modernas como YOLO o SSD.

El auge del aprendizaje profundo también ha impulsado el desarrollo de frameworks especializados que proporcionan herramientas optimizadas para el entrenamiento y la implementación de modelos de visión por computadora. TensorFlow [22] y Keras [23] destacan por su flexibilidad y por disponer de un ecosistema amplio de modelos preentrenados aplicables tanto a clasificación de imágenes como a detección y segmentación. PyTorch [24] ha ganado una gran popularidad en entornos académicos e industriales por su facilidad de uso y su enfoque dinámico en la construcción de modelos, mientras que Caffe [25], aunque menos utilizada en la actualidad, fue pionera en ofrecer arquitecturas optimizadas para tareas de clasificación y detección en tiempo real.

En el ámbito de la detección de objetos en tiempo real sobresalen modelos como YOLO [26], que permite localizar y clasificar múltiples objetos en una sola pasada por la red de manera rápida y eficiente, o SSD [27], que ofrece un equilibrio entre velocidad y precisión. Estas arquitecturas se han convertido en referencia para aplicaciones donde la latencia es un factor crítico, como la conducción autónoma o los sistemas de videovigilancia.

2.2.5. Protocolos de comunicación

En un sistema electrónico complejo como una prótesis robótica, la correcta comunicación entre los diferentes módulos (sensores, microcontroladores, motores, etc.) es fundamental. Los protocolos de comunicación establecen las reglas para el intercambio de datos, definiendo la velocidad, la fiabilidad y la arquitectura de la red interna. La elección del protocolo adecuado depende de varios factores, como la distancia entre los componentes, el volumen de datos a transferir y la tolerancia a la latencia.

Protocolos de comunicación cableados

Estos protocolos utilizan conexiones físicas (cables o pistas en la PCB) para transmitir la información, lo que los hace ideales para la comunicación interna entre los componentes del mismo dispositivo.

El más común por su simplicidad es el I²C (*Inter-Integrated Circuit*), un bus que con solo dos hilos (uno para datos y otro para el reloj) permite a un microcontrolador maestro comunicarse con múltiples dispositivos esclavos, como sensores u otros microcontroladores. Sin embargo, su velocidad relativamente baja (del orden de cientos de kHz), la longitud del bus es limitada y es susceptible al ruido si no se diseña correctamente.

Cuando se requiere una mayor velocidad, por ejemplo para leer datos de una pantalla o una memoria, se utiliza el protocolo SPI (*Serial Peripheral Interface*). Aunque necesita de 4 hilos, permite una comunicación *full-duplex* (envío y recepción simultáneos) a velocidades muy superiores (del orden de varios MHz).

Para una comunicación punto a punto más sencilla, el protocolo UART (asíncrono) es una solución robusta que solo requiere dos hilos: RX (Receptor) y TX (Transmisor). Asíncrono significa que no hay una señal de reloj compartida entre el emisor y

el receptor. La sincronización se logra definiendo una velocidad de transmisión (*baud rate*) común para ambos.

Finalmente, para sistemas que exigen la máxima fiabilidad, el bus CAN, heredado de la industria automotriz, ofrece un método de comunicación extremadamente robusto y tolerante a fallos entre múltiples nodos mediante un único bus, con un sistema de prioridad para los mensajes que minimiza las colisiones. Sin embargo, es más complejo de implementar que protocolos como I²C o SPI.

Protocolos de comunicación inalámbricos

Estos protocolos eliminan la necesidad de cables físicos, lo que es ideal para la comunicación con otros dispositivos fuera de la prótesis o para la interconexión con el sistema de control remoto.

En primer lugar encontramos Bluetooth, un estándar de comunicación inalámbrica de corto alcance (hasta 10-20 metros). En el contexto de los dispositivos portátiles, *Bluetooth Low Energy* (BLE) es la variante más relevante, optimizada para un consumo de energía extremadamente bajo. Como desventaja, presenta una baja velocidad de transferencia.

Cuando se necesita un gran ancho de banda, por ejemplo para transmitir un flujo de vídeo en tiempo real desde una cámara integrada, Wi-Fi es el estándar indiscutible. Ofrece una alta velocidad y un mayor alcance, pero su principal inconveniente es un consumo energético significativamente más elevado, lo que puede comprometer la autonomía del dispositivo si se utiliza de forma continua.

2.2.6. Programación y software en prótesis

El software se ha consolidado como el componente central que define la funcionalidad y la inteligencia de las prótesis modernas. Su evolución ha pasado de simples controladores embebidos hasta complejas arquitecturas distribuidas que gestionan la comunicación, el procesamiento de señales y la interacción con el usuario. El análisis de la investigación en este campo revela un enfoque en arquitecturas modulares, sistemas de control jerárquicos e interfaces de alto nivel para la configuración y validación. La creciente complejidad de las prótesis, especialmente aquellas con múltiples grados de libertad, ha impulsado un cambio desde sistemas monolíticos hacia arquitecturas de software modulares y escalables. Un ejemplo paradigmático es la tecnología desarrollada para el programa *Revolutionizing Prosthetics*, que introduce una capa de control del sistema (*System Control Layer*, SCL) que actúa como un sistema nervioso para la prótesis. Esta arquitectura, basada en un patrón de comunicación de publicación/suscripción, permite que diferentes módulos de software operen de forma independiente, facilitando la integración de nuevos algoritmos y desacoplando el desarrollo del software del hardware [28]. En paralelo, para el control de la locomoción en prótesis de pierna, el software adopta un enfoque de control jerárquico, donde una máquina de estados finitos de alto nivel interpreta el entorno y la intención del usuario para seleccionar el

modo de operación adecuado (caminar, subir escaleras), supervisando a los controladores de bajo nivel que gestionan las articulaciones. Este sistema permite a la prótesis adaptarse de forma autónoma y fluida al terreno [29].

El software no solo reside en el sistema embebido, sino que también juega un papel fundamental en la interfaz con el usuario y el médico. Se han desarrollado aplicaciones externas que actúan como un puente con el hardware, facilitando la configuración y el entrenamiento. Por ejemplo, se ha documentado el uso de software en un PC con una interfaz gráfica que se comunica vía serie con el microcontrolador de la prótesis, permitiendo ajustar parámetros y validar el dispositivo con el paciente [30]. Este principio de una arquitectura dual se mantiene en diseños más contemporáneos, que aprovechan herramientas de desarrollo modernas y accesibles. Por ejemplo, un proyecto reciente utiliza un microcontrolador PIC12F1822, que se comunica por un puerto serie con una aplicación de control en un PC. Dicha aplicación proporciona una interfaz gráfica para el diagnóstico, la prueba y el envío de comandos a la mano robótica, demostrando la vigencia de este modelo con tecnología actual [31]. Esta arquitectura distribuida se ha modernizado con tecnologías inalámbricas como *Bluetooth Low Energy* (BLE). En otros sistemas, una aplicación externa en un PC o *smartphone* asume las tareas computacionalmente exigentes, como procesar señales de una banda mioeléctrica para reconocer gestos, y luego envía los comandos de control a la prótesis. Este modelo permite que el microcontrolador de la prótesis se dedique exclusivamente al control motor, mientras que el procesamiento inteligente se delega a una plataforma más potente [32].

En resumen, el estado del arte refleja que el software protésico es un ecosistema dual: por un lado, un sistema embebido de bajo nivel, robusto y en tiempo real, responsable del control físico; y por otro, un sistema de alto nivel, a menudo externo, que gestiona la lógica compleja, la inteligencia artificial y la interacción con el usuario, facilitando una mayor funcionalidad y personalización.

3. Plan de gestión de proyecto

En esta sección se describen todos los aspectos importantes relativos a la gestión del proyecto: metodología, planificación, tecnologías, recursos y costes, riesgos y tipos de pruebas realizadas.

3.1. Metodología de desarrollo

Para llevar a cabo este proyecto, se ha seguido una metodología incremental por prototipos, debido a que tenemos distintos subsistemas que forman nuestro sistema final. Esto permite hacer pruebas del funcionamiento independiente de cada subsistema software con prototipos hardware para, finalmente, probar la integración de todos ellos en el sistema final. El proceso se estructuró en las siguientes fases:

1. **Fase de pruebas de concepto, aprendizaje y selección de plataforma**

Inicialmente, se hicieron pruebas principalmente de la comunicación I²C y SPI entre placas Arduino UNO (véase Figura 3.1) y entre Arduino UNO y sensores, usando lenguaje Arduino para aprender a usar estos protocolos de comunicación. Este tipo de pruebas era importante, debido a que para la colaboración entre los distintos elementos era fundamental la comunicación entre los mismos.

También se llevó a cabo un estudio de los posibles IDE a utilizar para programar los microcontroladores STM32 y se realizaron pruebas con las placas de desarrollo *NUCLEO-F446RE* (véase Figura 3.2a) y *NUCLEO-L432KC* (véase Figura 3.2b), por ser las que teníamos disponibles en el laboratorio de la familia STM.

Finalmente, también se realizaron pruebas con la Raspberry para aprender su uso y comunicación con los microcontroladores elegidos.

2. **Fase de desarrollo y validación de subsistemas**

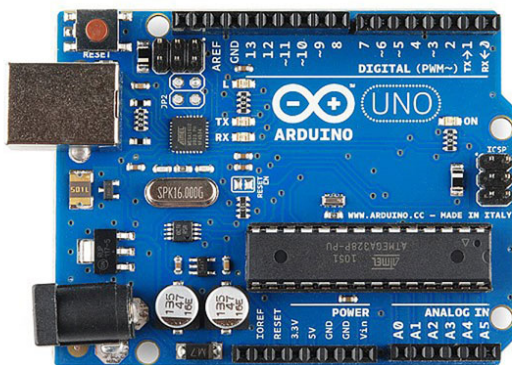
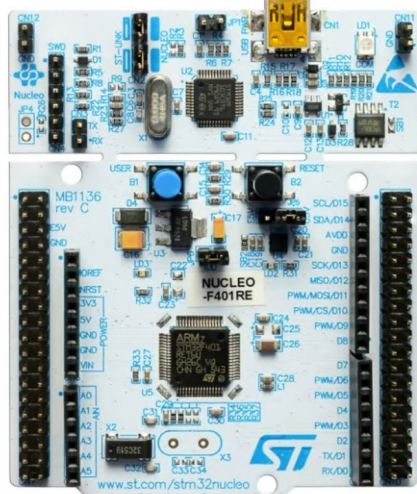
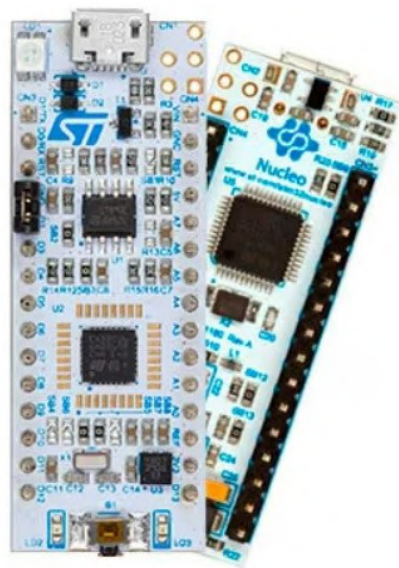


Figura 3.1: Arduino UNO



(a) NUCLEO-F446RE



(b) NUCLEO-L432KC

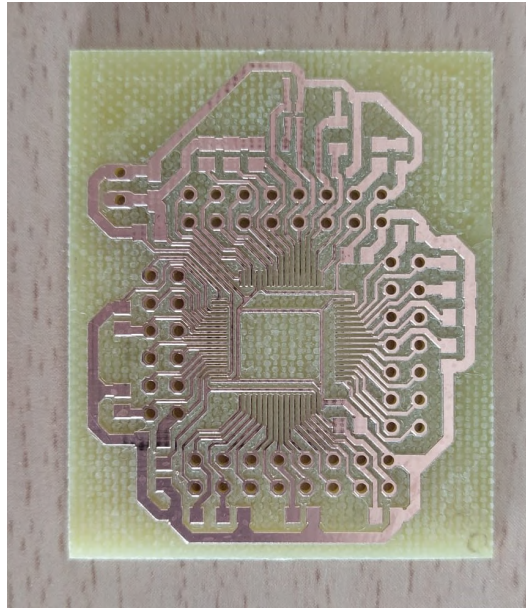
Figura 3.2: Placas de desarrollo STM

Se diseñaron y construyeron prototipos específicos para validar cada uno de los subsistemas de forma aislada:

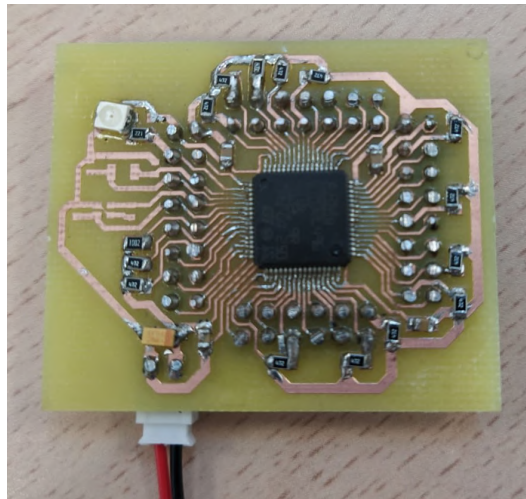
- a) **Subsistema de control de motores y subsistema de sensado de tacto:** Se diseñó, fabricó y soldó un primer prototipo que consistía en una placa de pruebas con un *STM32L496RGT6* (véase Figura 3.3), que nos permitió probar la programación y las comunicaciones con más detalle. También se hizo lo mismo con una segunda placa la cual incluía el circuito integrado para leer los sensores capacitivos y manejar el driver de control de los motores (véase Figura 3.4).
- b) **Subsistema de reconocimiento de voz:** Se utilizó una placa de desarrollo NUCLEO-G474RET6 junto a un módulo auxiliar diseñado a medida, que contenía el micrófono MEMS y su electrónica de acondicionamiento.
- c) **Subsistema de salida de audio:** Para la retroalimentación sonora, se empleó un módulo comercial Adafruit *MAX98357* (véase Figura 3.5) con un altavoz, controlado directamente desde la Raspberry Pi.
- d) **Subsistema de iluminación RGB:** Se emplearon leds rgb de Adafruit para realizar pruebas de funcionamiento y control desde la Raspberry Pi.

Para el subsistema de reconocimiento de objetos no fue necesario construir un prototipo, ya que con la Raspberry Pi y su cámara fue suficiente para realizar las pruebas. Algo parecido ocurre con el sensado EMG y la retroalimentación de tacto con motores vibradores, ya que para este subsistema dependíamos de los compañeros que lo estaban desarrollando para su propio Trabajo de Fin de Máster y no lo tuvimos hasta que lo finalizaron completamente.

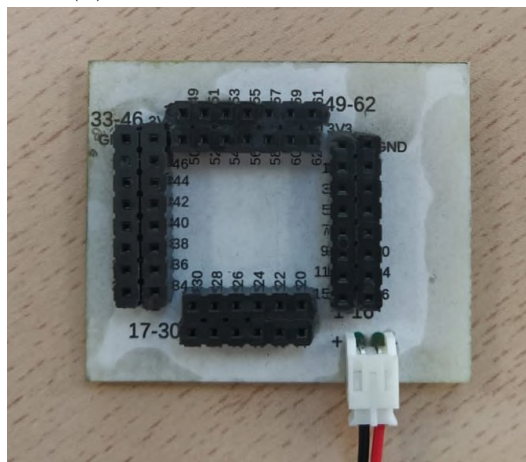
Las placas de prueba mencionadas fueron desarrolladas por nosotros en el laboratorio usando una *LPKF* (véase figura 3.6).



(a) PCB STM32-L496RGT6

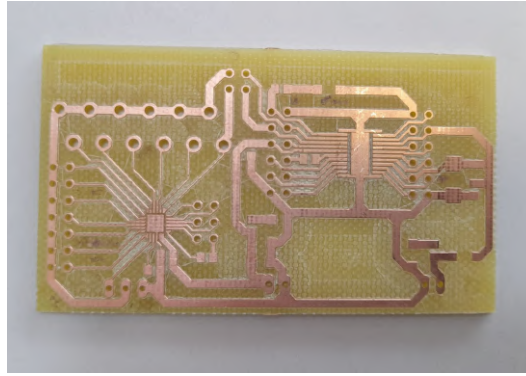


(b) PCB STM32-L496RGT6 soldada

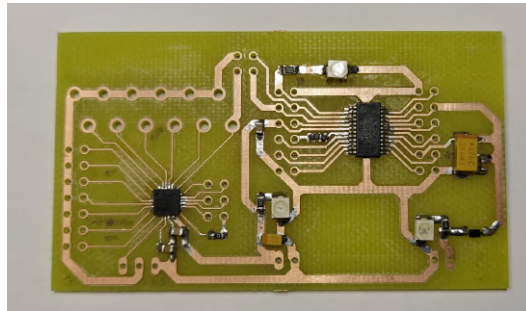


(c) PCB STM32-L496RGT6 pines

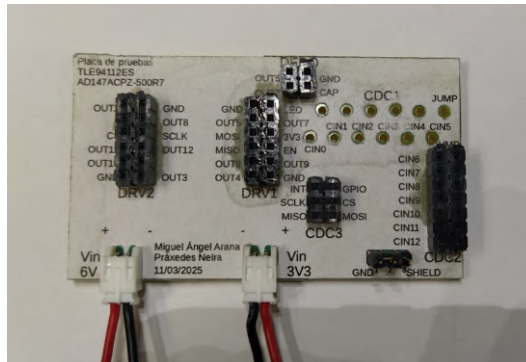
Figura 3.3: PCB STM32-L496RGT6



(a) PCB Capacitivos y Motores



(b) PCB Capacitivos y Motores soldada



(c) PCB Capacitivos y Motores final

Figura 3.4: PCB Capacitivos y Motores

3. Fase de integración incremental

Con todos estos prototipos se pudo probar en primera instancia cada uno de los subsistemas de manera independiente, para luego conectarlos uno a uno con la Raspberry Pi, que actúa como unidad central de procesamiento, para tener un prototipo del sistema integrado por partes.

3.1.1. Trabajos previos

Al ser un trabajo de integración, no se ha partido de 0, sino que se han aprovechado parte de los trabajos y diseños previos de otros compañeros que, para sus Trabajos de Fin de Grado y Máster, desarrollaron parte de algunos de los subsistemas aquí mencionados. Aún así, para poder realizar la integración, se ha tenido que

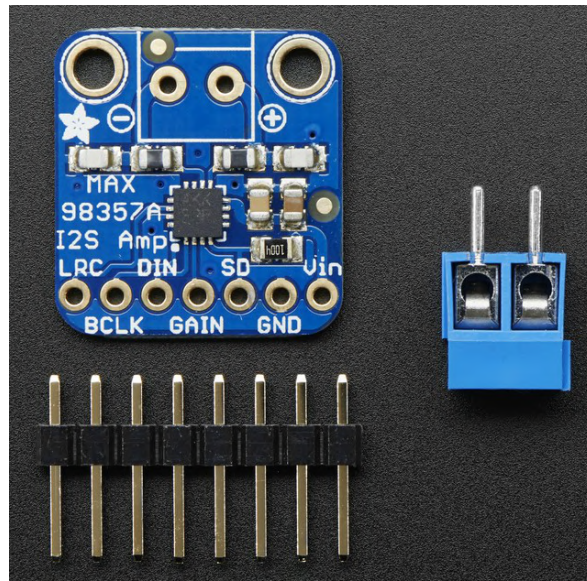


Figura 3.5: Módulo MAX98357 de Adafruit

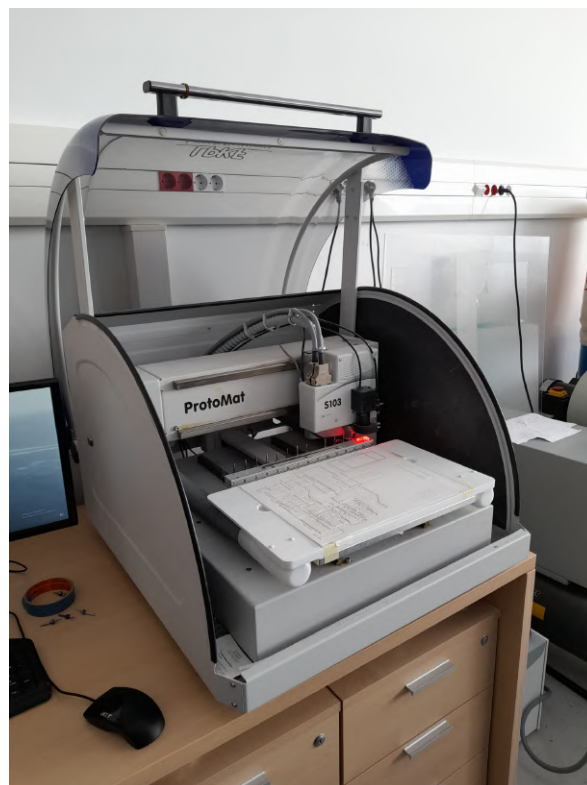


Figura 3.6: LPKF para fabricación de PCBs

rediseñar la mayoría del *software* y *hardware* para adaptarse a los nuevos objetivos, lo que también ha conllevado numerosas pruebas. En esta subsección se tendrá un primer acercamiento a dichos trabajos, explicándose con más detalle en la sección 4.1, donde también se hablará de los cambios, mejoras y adaptaciones realizadas para integrarlos en este TFG.

Trabajo sobre el control de motores, comunicación con Raspberry y arquitectura

Este proyecto es el principal en el que nos hemos basado para continuar y mejorar el desarrollo de la prótesis robótica. En este proyecto se desarrolló la arquitectura de la prótesis, tanto a nivel hardware como software. En cuanto a la implementación, se usaron microcontroladores de la familia AVR programados en lenguaje Arduino para llevar a cabo principalmente el control de motores y su comunicación con una Raspberry Pi Zero W. Aunque se diseñase la arquitectura, no se llevó a cabo la implementación de los demás subsistemas que no fuesen lo relacionado con los motores y la comunicación con el subsistema de reconocimiento de voz, implementado en otro trabajo.

La arquitectura que se propuso en este proyecto es la que podemos ver en la Figura 3.7. El problema que presentaba era que el AD7147 usado para leer los sensores capacitivos se conectaba directamente al bus I²C común de todos los elementos, lo que hacía que todo el sistema se saturase cuando otro compañero realizó pruebas de concepto.

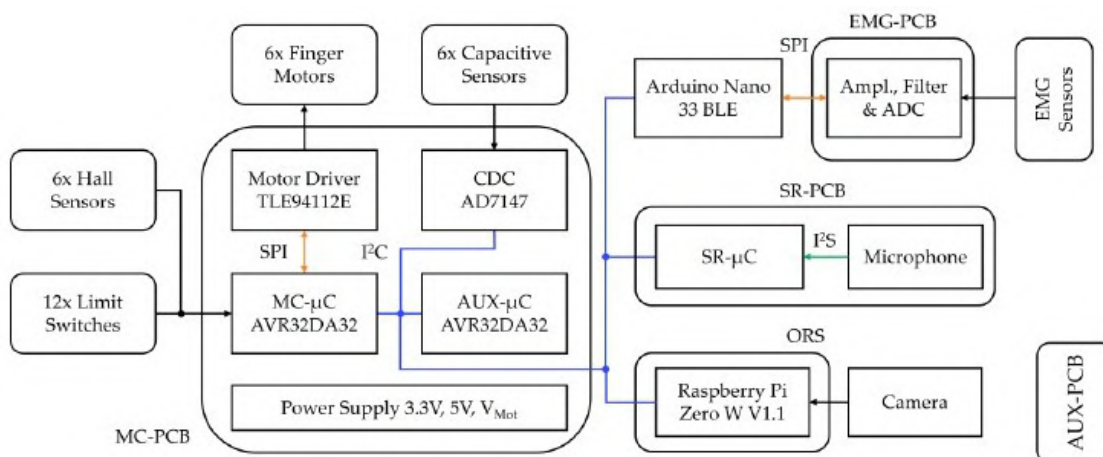


Figura 3.7: Arquitectura hardware propuesta en trabajo previo

Trabajo sobre reconocimiento de voz

Este fue un Trabajo de Fin de Grado que se basaba en implementar un sistema de reconocimiento por voz en un microcontrolador usando un modelo de *Deep Learning*, en concreto *CNN*, que empleaba espectrogramas logarítmicos de Mel como entradas, siendo implementado y entrenado en MATLAB. En las Figuras 3.8 podemos ver el diseño del modelo *CNN* en el *Deep Network Designer* de MATLAB y en la Figura 3.9

la matriz de confusión obtenida en la validación. El entrenamiento fue llevado a cabo para que se reconocieran 7 clases, siendo estas los números del 1 al 5 en castellano, más una sexta clase que identificaba ruido de fondo y una séptima para salidas desconocidas. El microcontrolador empleado fue un *STM32 G474RET6*. La placa diseñada en este trabajo la podemos ver en la Figura 3.10.

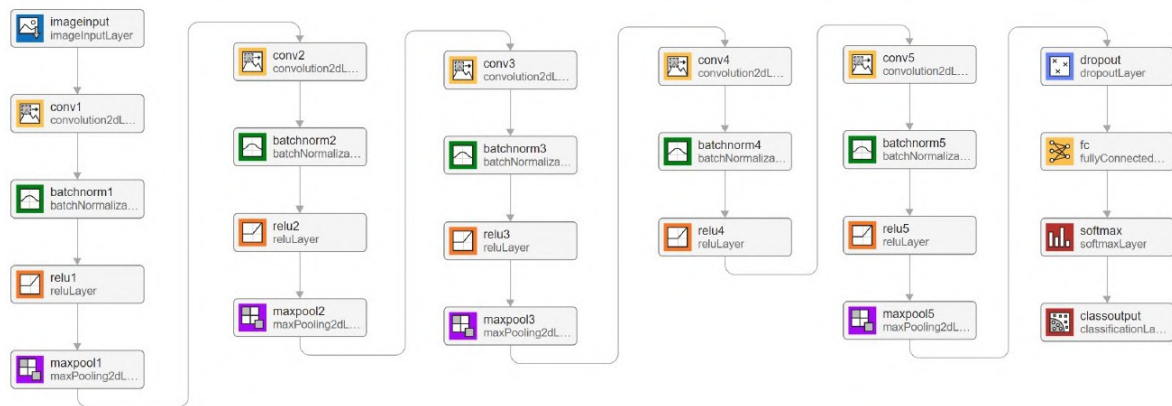


Figura 3.8: Diseño del Modelo CNN en el *Deep Network Designer* de MATLAB.

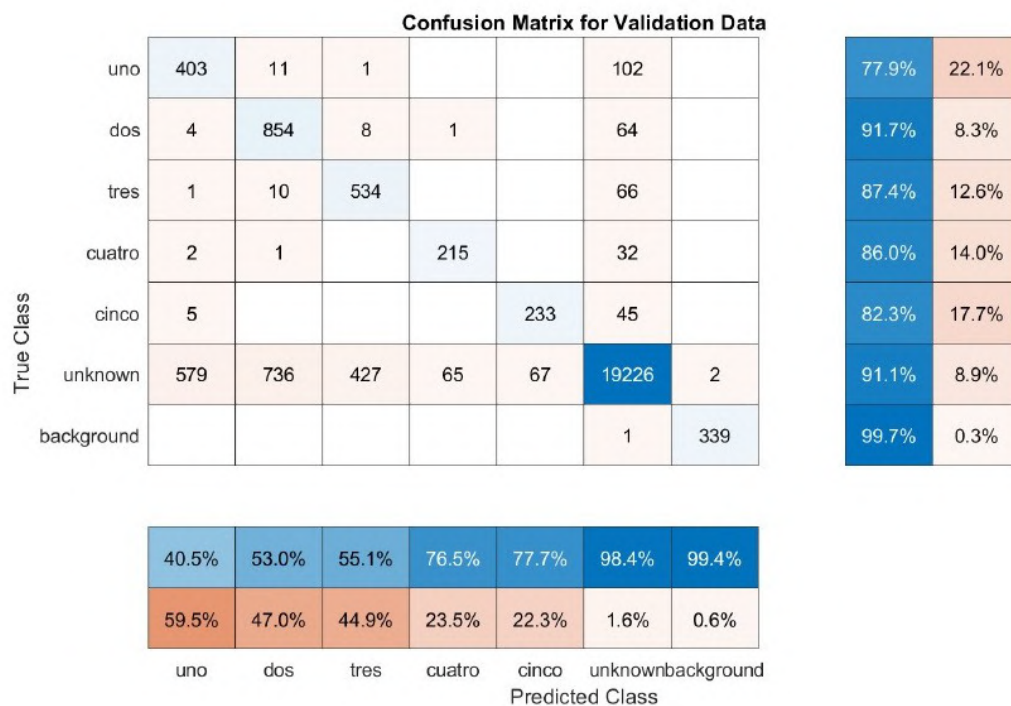


Figura 3.9: Matriz de confusión para los datos de validación con el modelo CNN.

Trabajo sobre reconocimiento de objetos mediante visión artificial y feed-back capacitivo

En este proyecto, se desarrolló un sistema que reconociese objetos mediante *OpenCV* usando una Raspberry Pi Zero 2W y su cámara. Además, hacía que se activasen motores vibradores en función de si se detectaba tacto o no en unos sensores



Figura 3.10: PCB antigua para el trabajo de reconocimiento de voz

capacitivos, siendo este sistema gobernado por un microcontrolador *AVR32DA32* (véase Figuras 3.11 y 3.12).

Trabajo sobre sensores EMG

Este Trabajo de Fin de Máster se encargó del diseño y fabricación de unas PCB (véase Figura 3.13) que, a modo de brazalete, eran capaces de recoger señales EMG y extraer características de los mismos, para luego pasarlas a su PCB principal, la cual ejecuta una red neuronal que clasificaba el tipo de gesto realizado con los dedos de la mano. En las PCB se usaron microcontroladores STM32 de la familia L4 (*L496RGT6* en el principal y *L431KCU6* en las auxiliares).

Para entrenar la red neuronal, se obtuvo un dataset a base de leer datos de su propio antebrazo y se entrenó usando un script de Python con la API de Keras, que era capaz de clasificar 4 gestos: puño cerrado, mano abierta, pulgar y símbolo de la paz. Para formar el dataset, se desarrolló un flujo de *STM32CubeMonitor*, que se encargaba de activar la comunicación I²C entre el microcontrolador principal y los auxiliares para que estos comenzaran la recogida de señales EMG para, posteriormente, introducir esos datos etiquetados con su clase en un CSV. La comunicación I²C empleada aparece en la Figura 3.14. La arquitectura de su red neuronal y la matriz de confusión obtenida las podemos ver en las Figuras 3.15 y 3.16, respectivamente.

Muchos de estos proyectos se implementaron y probaron en la prótesis que podemos ver en la Figura 3.17.

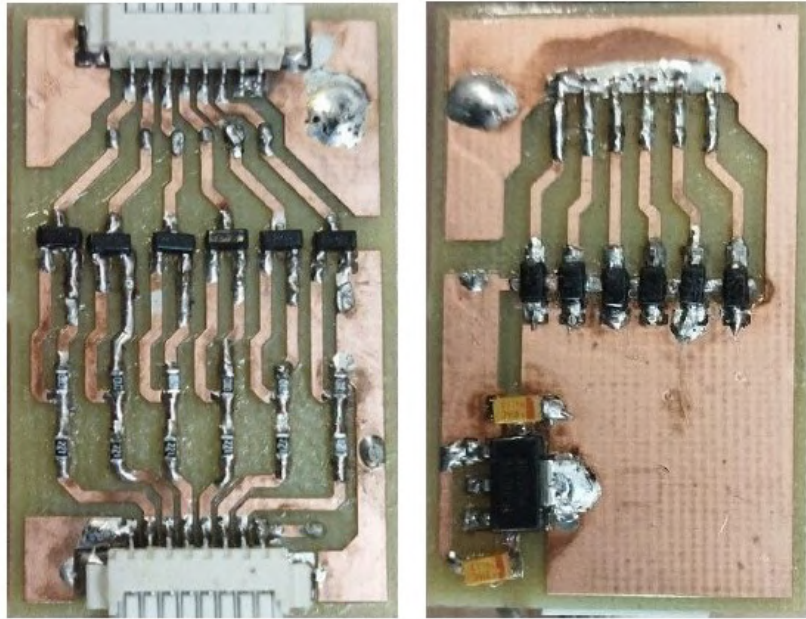


Figura 3.11: PCB para driver de motores vibradores

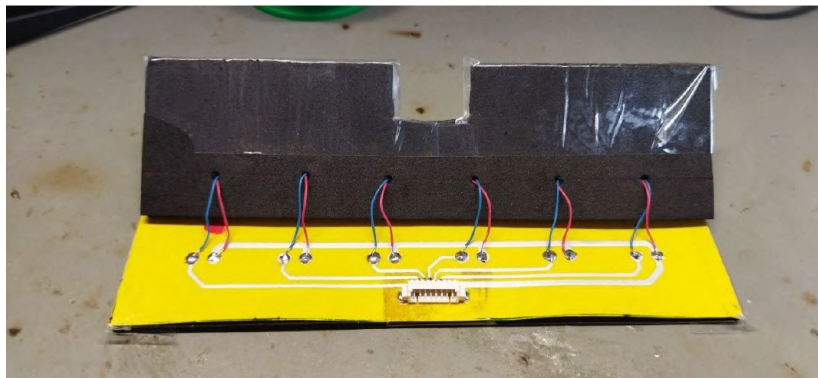


Figura 3.12: PCB flexible para motores vibradores

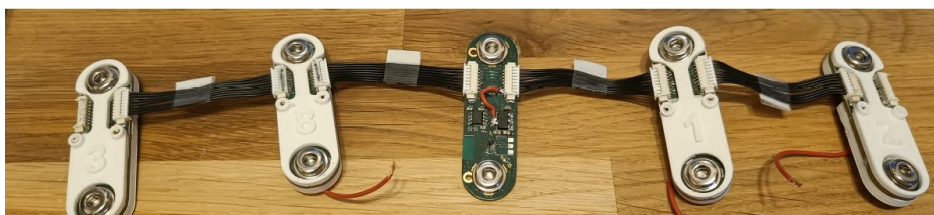


Figura 3.13: Brazalete con PCB que recogen las señales EMG y PCB principal que las procesa

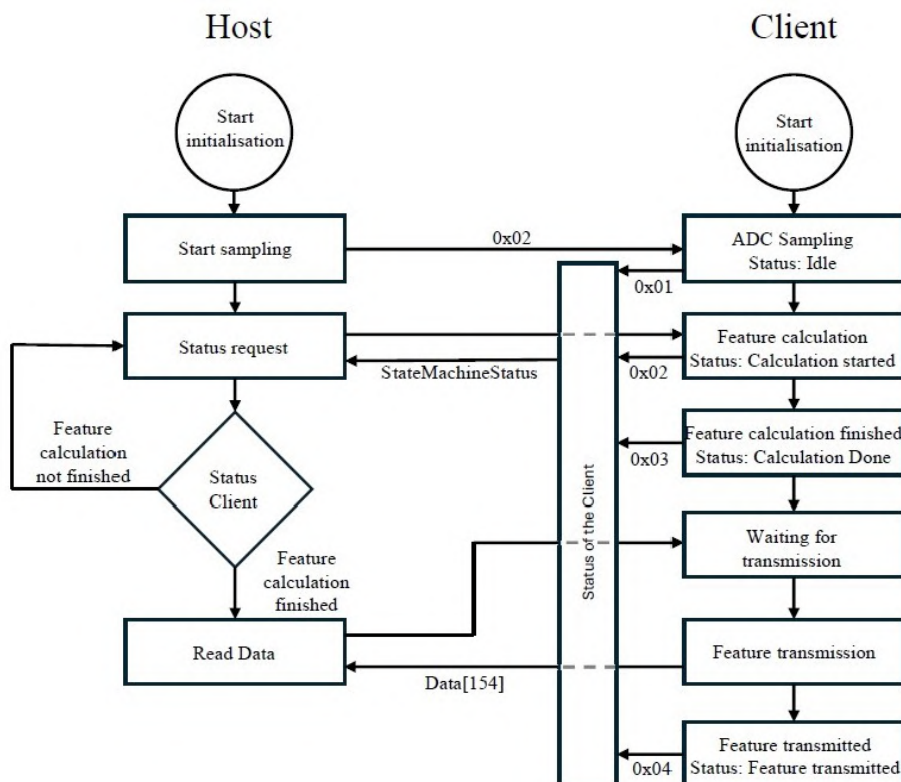


Figura 3.14: Flujo de la comunicación I²C entre el microcontrolador principal y los auxiliares para el trabajo de sensado EMG

3.2. Planificación del proyecto

Como ya hemos mencionado en la sección 3.1, al seguir una metodología incremental por prototipos, las fases en las que se ha dividido parte del proceso fueron guiadas por esta idea. Aún así, hay fases previas y posteriores a los prototipos que también deben ser consideradas. Dicho esto, podemos dividir el proyecto en lo siguiente:

1. Investigación inicial y estudio de trabajos previos

Durante esta fase llevé a cabo una investigación sobre las prótesis robóticas de manos en general y luego específicamente sobre prótesis robóticas para niños, para ver las distintas alternativas que ya habían sido desarrolladas. Además, como mi proyecto se centraba en la integración de los trabajos realizados por otros compañeros, me dediqué a estudiar cada uno de estos trabajos en profundidad, tanto a nivel *hardware* como *software*, para ver cómo era su funcionamiento y arquitectura y por qué habían tomado ciertas decisiones.

2. Análisis y diseño de la arquitectura hardware

Antes de empezar con cualquier elemento software, discutí con mi compañero Miguel Ángel Arana (el cual se encargó de desarrollar el diseño electrónico del sistema completo) cuáles eran las mejores opciones a nivel de diseño *hardware*. Esto comprendía elección de familia de microcontroladores, a qué se iba a dedicar cada uno, qué protocolo usaríamos para comunicarlos entre ellos y con los sensores, qué circuitos integrados íbamos a usar para manejar motores y sensores

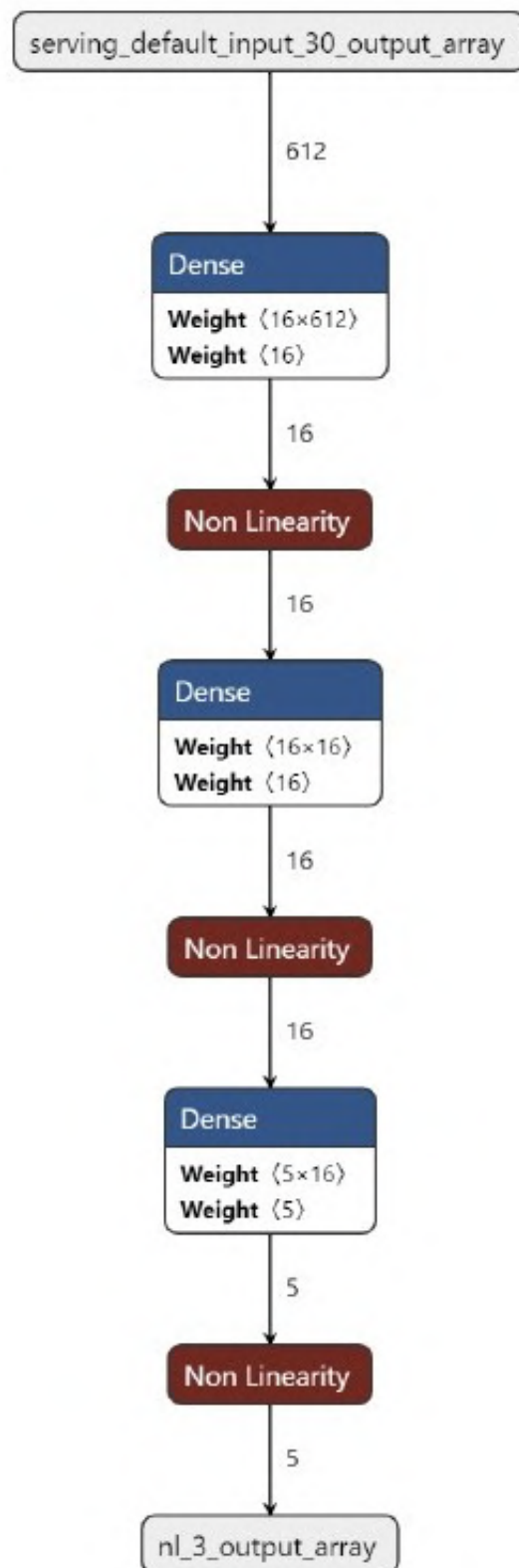


Figura 3.15: Arquitectura de la red neuronal del microcontrolador principal

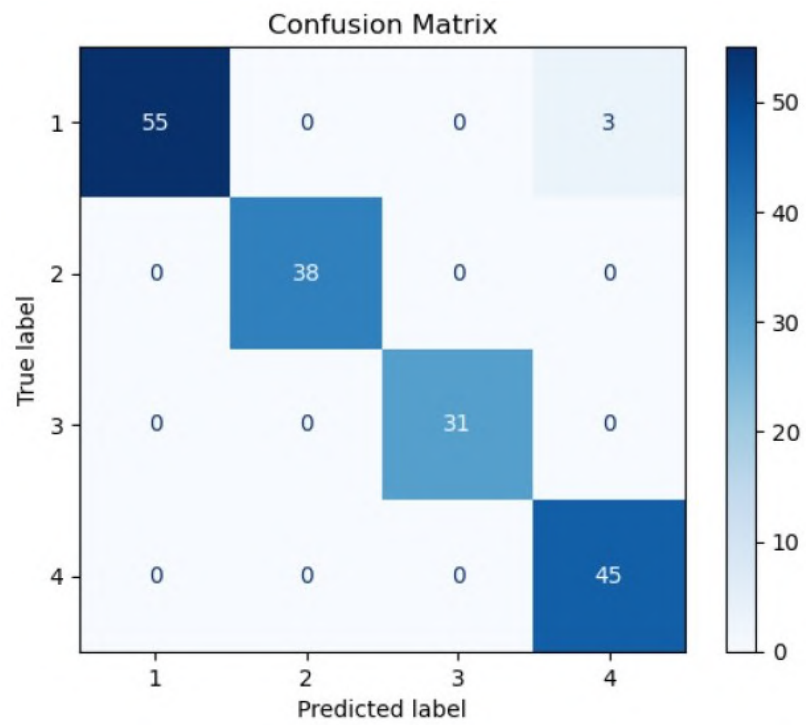


Figura 3.16: Matriz de confusión del entrenamiento de 4 clases con sensores EMG

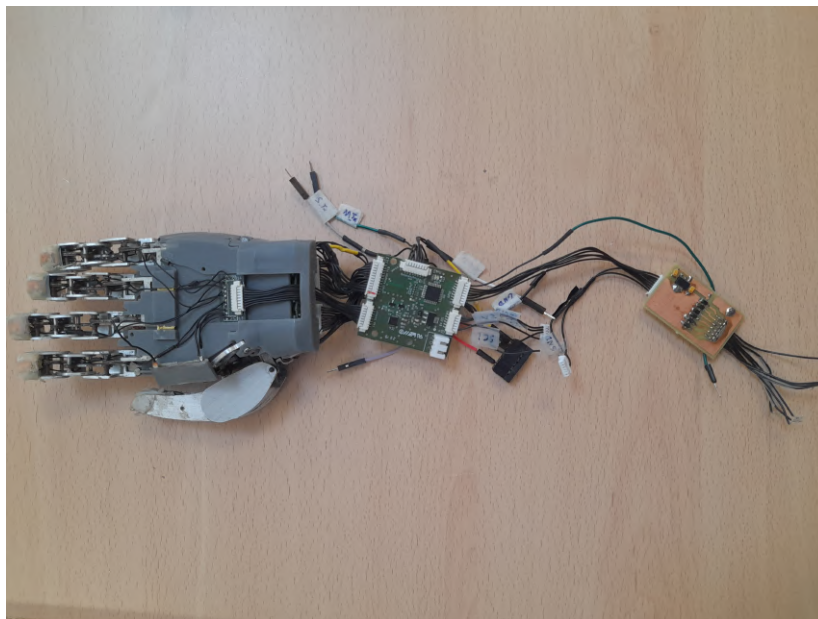


Figura 3.17: Prótesis antigua

y qué sensores íbamos a elegir.

3. Análisis y diseño de la arquitectura software

Una vez realizado el diseño de la arquitectura hardware, realicé un análisis de la funcionalidad que se iba a requerir en la prótesis y en base a ello diseñé una arquitectura software acorde a la misma, de manera que en ella se integraran cada uno de los sistemas y colaborasen entre ellos.

4. Pruebas de concepto, aprendizaje y selección de plataforma

Se llevaron a cabo pruebas para decidir qué IDE y lenguaje emplear para la programación de los microcontroladores elegidos y para aprender más en profundidad el uso de los mismos y sus periféricos, principalmente de los buses I²C y SPI.

5. Desarrollo y validación de subsistemas

En esta fase se realizaron distintos prototipos para desarrollar y probar cada uno de los subsistemas que integrarían el sistema final y que ya hemos explicado en la sección 3.1.

6. Integración incremental

Aquí se pudo probar cada subsistema en colaboración con la Raspberry Pi (elemento central), para luego comunicarlos entre ellos. Finalmente se probó todo de nuevo en una PCB fabricada específicamente para integrar prácticamente todos los elementos *hardware*.

7. Adaptaciones finales y documentación

Esta última fase comprende los ajustes y adiciones realizados al sistema para perfilar o mejorar su funcionamiento final. Además, aquí también se llevó a cabo la redacción de la memoria final del proyecto. Cabe destacar que durante toda la vida del proyecto se fueron documentando los logros e investigaciones realizadas, pero estos no se comenzaron a integrar en el documento final hasta esta fase.

En la Figura 3.18 podemos ver el diagrama de Gantt asociado a la distribución de tiempos de cada una de las fases.

Como podemos observar, muchas de las fases se solapan, como es el caso del análisis y diseño de la arquitectura software con las pruebas de concepto, o el desarrollo y validación de subsistemas con la integración incremental. En este último caso, al seguir una metodología por prototipos, cuando finalizaba el prototipo de un subsistema, lo probaba y comunicaba con la Raspberry Pi antes de seguir con el siguiente, por ello el solapamiento.

A modo de conclusión de la planificación, los tiempos que aparecen en el diagrama son bastante parecidos a lo que estimé en un inicio para la duración de cada fase.

3.3. Tecnologías

Al ser este trabajo de una naturaleza *software* aplicado a un *hardware* concreto como es la prótesis robótica, en esta sección haré una distinción entre estos dos tipos.

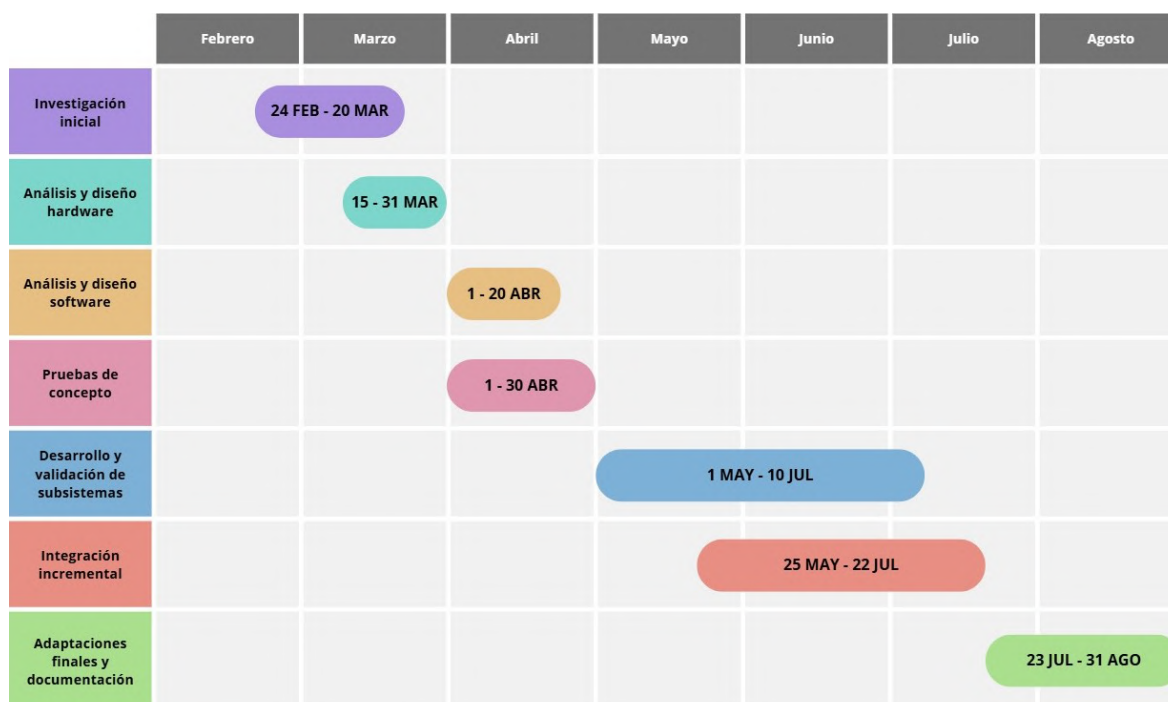


Figura 3.18: Diagrama de Gantt de las fases del proyecto

3.3.1. Hardware

STM32

La familia de microcontroladores elegida para controlar los subsistemas ha sido STM32. Esta decisión se basa en que son microcontroladores potentes, con núcleos ARM Cortex-M4 y M7; cuentan con una amplia variedad de opciones, gamas (como bajo consumo o alta potencia), periféricos y configuración de los mismos; y ofrecen capacidades avanzadas de procesamiento digital de señales y unidad de punto flotante, ideal para la sensórica y procesamiento de una prótesis robótica. Además, cuentan con una comunidad activa, un ecosistema de desarrollo robusto (STM32CubeMX, STM32CubeAI, HAL/LL *drivers*) y una amplia documentación y foros *online* lo que, unido a su escalabilidad, facilita tanto el prototipado como la transición a versiones más avanzadas ([33], [34]). En la tabla 3.1 podemos ver una comparativa con otros microcontroladores.

Raspberry Pi Zero 2W

Como elemento de control central y orquestador del sistema, se ha elegido una Raspberry Pi Zero 2W (véase Figura 3.19). En primer lugar, se ha elegido Raspberry Pi y no otro tipo de plataforma (como Orange Pi o Banana Pi) por ser la más desarrollada, conocida, con mayor comunidad y soporte en línea. Además, con respecto a un microcontrolador presenta ciertas ventajas, como el hecho de mayor capacidad de procesamiento; facilidad de desarrollo e integración al permitir usar bibliotecas de alto nivel, como OpenCV, TensorFlowLite y desarrollo en C/C++, Python o incluso ROS;

Característica	STM32	AVR32	ESP32
Arquitectura	ARM Cortex-M	AVR32 (propia de Atmel)	Xtensa LX6/LX7 o RISC-V
Ecosistema y mercado	Líder del mercado embebido. Ecosistema masivo y maduro.	Desplazado por los Cortex-M.	Estándar de facto para IoT.
Rendimiento	Muy escalable.	Comparable a un Cortex-M3/4 temprano.	Alto rendimiento, especialmente con doble núcleo.
Conectividad integrada	Muy variada. Periféricos estándar (SPI, I ² C) y avanzados, como Ethernet, CAN, USB.	Periféricos estándar robustos. Menos variedad que en STM32.	Principal ventaja: Wi-Fi y Bluetooth integrados de serie en la mayoría de modelos.
Consumo de energía	Excelente optimización, con familias dedicadas de ultra bajo consumo.	Moderado.	Moderado a alto por la radio, pero con modos de sueño profundo.
Herramientas de desarrollo	Ecosistema muy rico: STM32CubeIDE, Keil, Arduino. Generador de código STM32CubeMX.	Principalmente de Microchip: Atmel Studio. Soporte limitado de herramientas de terceros.	Accesible y potente: ESP-IDF, Arduino, PlatformIo, MicroPython.
Coste	Amplio rango.	Generalmente más caros que sus equivalentes ARM Cortex-M.	Excelente relación coste/beneficio, sobre todo por la conectividad que incluye.

Tabla 3.1

Tabla comparativa de diferentes familias de microcontroladores.

posibilidad de multitarea real; y soporte para cámaras y visión artificial. Por otro lado, se ha elegido este modelo de Raspberry y no otro debido principalmente a su reducido tamaño, ideal para integrarse en la prótesis robótica. A pesar de sus dimensiones, su capacidad de procesamiento es notable, suficiente para ejecutar los algoritmos de alto nivel y orquestar el sistema.

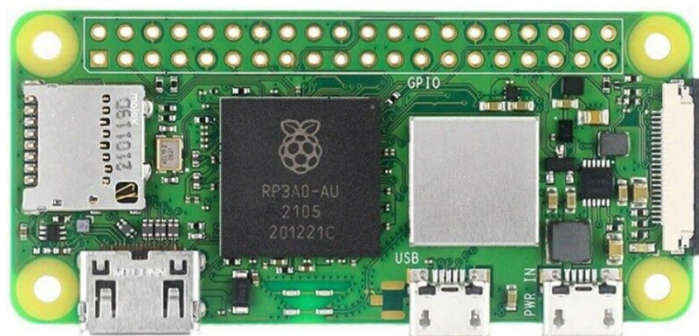


Figura 3.19: Raspberry Pi Zero 2W

Cámara Raspberry Pi

Se ha decidido emplear la *Raspberry Pi Camera Module 2* (Figura 3.20) [35], que se encargará de obtener la entrada de vídeo que servirá de muestreo para el reconocimiento de objetos. Es un modelo simple, con la resolución suficiente para el algoritmo de reconocimiento de objetos que usaremos y directamente compatible con nuestro módulo de Raspberry Pi. Además, contamos con ella en el laboratorio, lo que influye, obviamente, en su elección.



Figura 3.20: Modelo de cámara Raspberry utilizado (V2)

Motores DC

Para mover cada dedo, se ha decidido emplear un motor DC (Figura 3.21), así que, sabiendo que el meñique y el anular comparten motor y que el pulgar tiene un motor para movimiento lineal y otro para rotativo, contamos con 5 motores DC. Se ha decidido emplear estos motores en lugar de motores paso a paso porque con un único driver podemos controlar los 5 y no necesitamos bloquear el código para moverlos.

Con un mensaje SPI por cada motor es suficiente. Además, se puede reaprovechar el driver que se empleó anteriormente y que ya conocemos cómo se usa. Por otro lado, la reductora de estos motores debe ser grande, para que se muevan a una velocidad no demasiado rápida y así podamos tener un control de la posición de los mismos, además de tener fuerza. Es por ello que decidimos usar motores de 6V a 35RPM [36].

Nota: Debido a que tuvimos problemas con el diseño mecánico, los motores de 35 RPM mencionados en esta sección no son los que están montados en el prototipo. En él tenemos motores de reductora 30:1, que son más rápidos y, por lo tanto, menos controlables que los que tenemos pensados para el diseño final.



Figura 3.21: Motor DC de cada dedo

Potenciómetros

Para saber la posición absoluta de cada dedo, se ha decidido incluir un potenciómetro para cada motor. Estos potenciómetros son multivuelta, aunque con el diseño mecánico final, con media vuelta será suficiente para mover cada dedo. Se han empleado potenciómetros en lugar de otro sistema porque otorgan posición absoluta y es un sistema sencillo y fácilmente adaptable al eje de cada motor.

Sensores capacitivos

Para medir el tacto, se ha optado por usar sensores capacitivos por los siguientes aspectos: tienen mayor sensibilidad y resolución que los resistivos; posibilitan la detección por proximidad, lo que permite anticipar el cierre de dedos o mejorar la seguridad al agarrar objetos frágiles; y se pueden integrar fácilmente en superficies curvas, como las yemas de los dedos de la prótesis. Además, al ser un prototipo, queríamos integrar los sensores capacitivos desarrollados por el compañero Francisco José Alcaide para su Trabajo de Fin de Máster y así probar su funcionalidad.

Driver TLE94112ES

El manejo de motores DC requiere de un *driver* y se ha resuelto utilizar el *TLE94112ES* (Figura 3.22) porque ya se usó en el proyecto previo que se encargaba

de mover los dedos, por lo que teníamos un conocimiento mayor de su uso y, además, en ese proyecto el funcionamiento parecía cumplir con las expectativas. También tiene 12 medios puentes, lo que nos permite controlar hasta 6 motores con un solo *driver*.

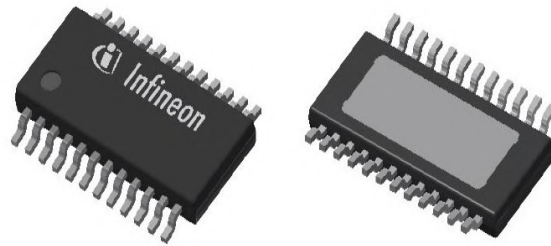


Figura 3.22: Driver de motores DC TLE94112ES

Conversor capacitivo-digital AD7147

Se optó por este circuito integrado (Figura 3.23) ya que aporta una resolución de 16 bits, frente a los 12 bits del TSC de STM; solo necesitamos los pines para la comunicación, en lugar de mínimo un pin por cada sensor, lo que lo hace escalable a más sensores; y usa el plano *ACSHIELD* para reducir el ruido en la medida. Todo esto hace que sea un candidato idóneo frente a las otras alternativas.

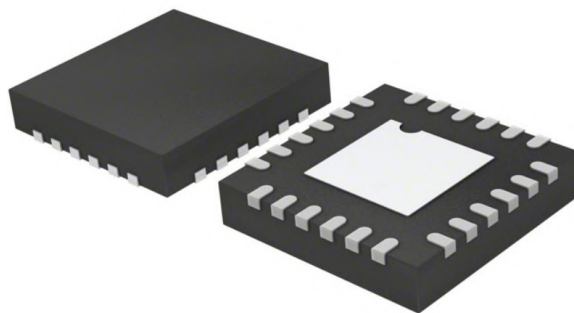


Figura 3.23: Conversor capacitivo-digital AD7147

Bus I²C

Para la comunicación entre nuestro sistema central, que es la Raspberry Pi, y el resto de microcontroladores, se ha decidido usar este protocolo de comunicación, siendo la Raspberry Pi el maestro, porque con 2 líneas físicas podemos acceder a todos los esclavos, lo cual es una ventaja enorme al considerar el escaso número de pines que tiene la Raspberry.

Bus SPI

En el caso de la comunicación de los microcontroladores con circuitos integrados (como el driver de motores o el convertidor capacitivo-digital), se ha optado por el uso

de este protocolo que, aunque requiere más líneas físicas, puede alcanzar velocidades mucho mayores que el protocolo I²C, que es lo que nos interesa en este caso. Además, liberamos al bus I²C general de la sobrecarga que produce el AD7147, como ocurría en la arquitectura que vimos en la Figura 3.7.

Nota: No se incluye aquí el sistema de sensado EMG, ya que al ser un proyecto que estaba siendo desarrollado de manera paralela por otros compañeros, las decisiones fueron tomadas por ellos.

3.3.2. Software

STM32CubeIDE

Como se van a emplear microcontroladores de la familia STM32, se ha decidido que la mejor opción para desarrollar los programas es usar el IDE que proporciona el propio STM (Figura 3.24). En un inicio se hicieron pruebas con el IDE de Arduino, pero para utilizarlo era necesario un gestor de placas específico de STM, el cual no permitía explotar todas las características del microcontrolador, como acceso sencillo a los modos de bajo consumo, depuración avanzada con SWD o breakpoints, gestión avanzada de interrupciones y prioridades, acceso a *middleware* oficial, como STM32Cube.AI o las pilas USB y control total del *hardware*, entre otros. Además, en ese gestor no estaban todavía portados todos los microcontroladores STM que íbamos a usar. Es por ello que se optó por el STM32CubeIDE porque, aunque Arduino es más sencillo, también te limita el acceso a periféricos avanzados y configuraciones finas, siendo el IDE de STM más completo e intuitivo a la hora de programar sus microcontroladores, ofreciendo muchas más posibilidades y configuraciones.

C/C++

Se ha resuelto por emplear C y C++ (Figura 3.25) como lenguajes de programación de los microcontroladores, ya que STM ofrece la HAL (*Hardware Abstraction Layer*) escrita en C, para poder manejar sus periféricos a un más alto nivel y de manera que el código generado para un microcontrolador de STM sea portable para otro. Por otro lado, C++ se emplea para implementar la orientación a objetos que nos permitirá en algunos de los códigos una mayor legibilidad y claridad a la hora de programar, así como una estructura clara en los microcontroladores que requieran una programación más compleja, como en los de los motores y los sensores capacitivos.

Python

Python (Figura 3.26) es el lenguaje de programación que se empleará en nuestro sistema central, que es una Raspberry Pi. Las razones de su elección son su facilidad de uso para el manejo de pines de la Raspberry; y su amplia compatibilidad con librerías para el manejo de periféricos como un altavoz, los LEDs RGB y visión artificial

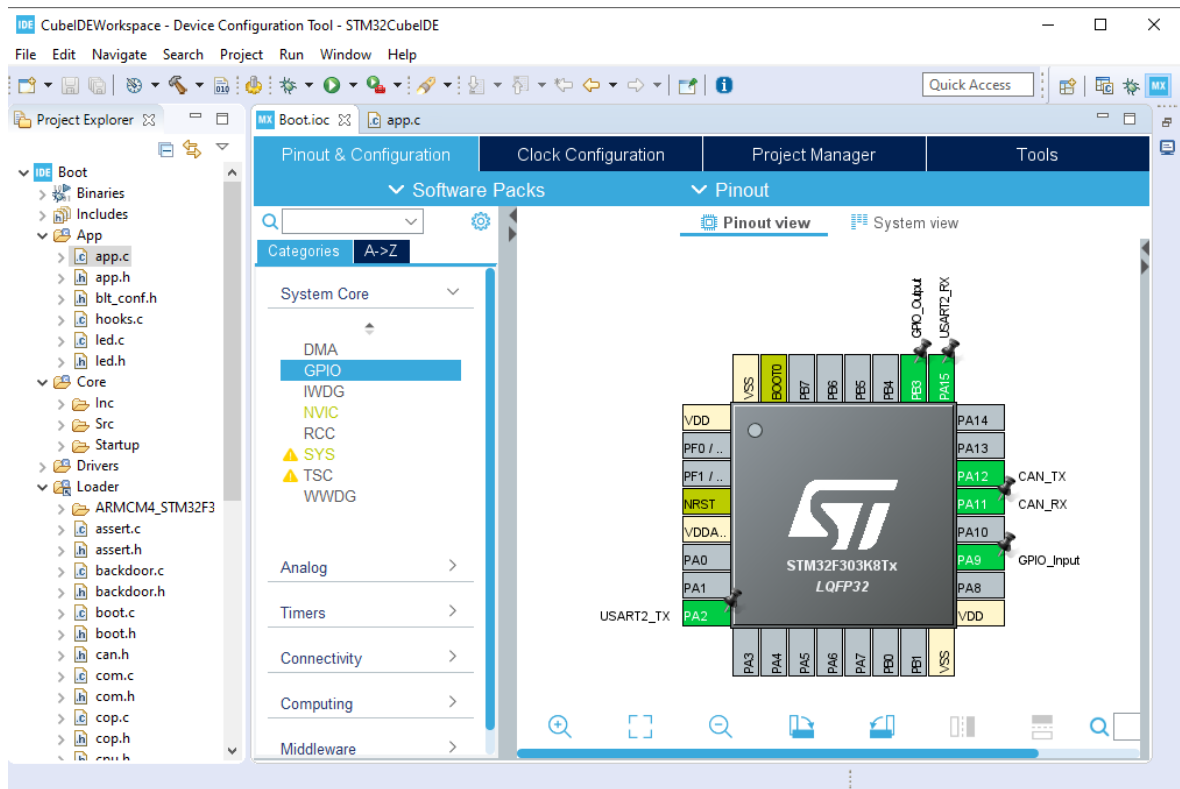


Figura 3.24: IDE de STM



Figura 3.25: Logos de los lenguajes C y C++

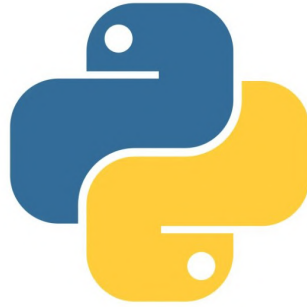


Figura 3.26: Logo de Python

con OpenCV. Asimismo, fuera de su uso en la Raspberry, permite implementar, ejecutar, entrenar y probar con facilidad modelos de *deep learning* usando herramientas y librerías específicas.

TensorFlow

Esta biblioteca de código abierto permite construir, entrenar y probar modelos de *deep learning* de manera simple y fácil, usando su API de alto nivel, *Keras*. Además, se puede combinar con otras bibliotecas muy utilizadas en este campo como *Matplotlib*, *Numpy*, *Pandas*, etc. Por otro lado, permite la conversión del modelo entrenado a otros formatos, como ONNX, y su compactación usando TensorFlow Lite, lo que lo hace ideal para implementar estos modelos en microcontroladores. Todo esto, unido a que Python es el lenguaje que mejor lo soporta, convierte a TensorFlow en la biblioteca que más se ajusta a nuestras necesidades a la hora de implementar un modelo de red neuronal. (Figura 3.27).



Figura 3.27: Logo de TensorFlow

STM32Cube.AI

STM32Cube.AI es el *Middleware* que ofrece STM para implementar en sus microcontroladores modelos de redes neuronales previamente entrenados. Esta herramienta convierte el modelo a código C embebido, optimizado para su ejecución en STM32,

y genera el código junto con funciones auxiliares para integrarlo directamente en un proyecto STM32CubeIDE.

OpenOCD

Para programar los microcontroladores empleando directamente la Raspberry y sin necesidad de conectores extras, lo que nos ahorra espacio en la PCB final, se ha decidido emplear OpenOCD, una herramienta de software libre que permite programar y depurar microcontroladores mediante interfaces JTAG o SWD, necesitando solamente el binario del código que queramos programar. Gracias a esto, se pueden programar los microcontroladores de una manera centralizada desde la Raspberry y, por tanto, inalámbrica, puesto que a la misma podemos acceder mediante SSH. Los pasos necesarios para la instalación y uso de esta herramienta aparecen en el Manual del Desarrollador, en el Anexo [A.2.2](#).

Raspbian Buster

Debido a que en los proyectos anteriores se usó esta versión del sistema operativo de Raspberry Pi, se ha optado por seguir con ella aunque no sea la versión más reciente, por demostrar ser estable con el software anterior, entre ellos OpenCV, el cual es bastante complicado y tedioso de instalar.

OpenCV

OpenCV (Figura [3.28](#)) es una de las mayores bibliotecas de código abierto de visión artificial actualmente y que permite a los sistemas interpretar y procesar imágenes y vídeo en tiempo real. Su amplia documentación y comunidad, su potencia y su compatibilidad con Python y las cámaras de Raspberry, la hacen una elección idónea para este proyecto. Además, es más eficiente para sistemas embebidos, con recursos limitados y un hardware menos potente que otras alternativas como YOLO. Los pasos necesarios para la instalación de OpenCV en la Raspberry Pi Zero 2W se pueden consultar en el Manual del Desarrollador, en el Anexo [A.2.2](#).



Figura 3.28: Logo de OpenCV

SSH

Para realizar comunicación remota con nuestra Raspberry Pi, se ha empleado el protocolo SSH, que nos permite ejecutar todo tipo de comandos en la misma tal y como si estuviéramos conectados con un teclado.

WinSCP

WinSCP es un popular cliente de código abierto para Windows, diseñado para facilitar la transferencia segura de archivos entre un ordenador local y un sistema remoto como la Raspberry Pi. A través de una intuitiva interfaz gráfica, simplifica la gestión del sistema de archivos al soportar protocolos como SFTP, SCP y FTP, permitiendo administrar carpetas y editar ficheros de forma remota y eficiente.

3.4. Costes

Para estimar el coste del proyecto, se deben tener en cuenta los costes de los recursos materiales y humanos, así como los costes indirectos.

Recursos utilizados:

- **Hardware:** STM32-L496RGT6, STM32-G474RET6, Adafruit MAX98357, Raspberry Pi Zero 2W, Raspberry Pi Camera Module 2, tarjeta microSD de 32 GB, componentes electrónicos varios, motores DC, PCB de la palma.

Nota: Las placas de desarrollo NUCLEO y la LPKF no las menciono porque ya teníamos en el laboratorio de otros proyectos anteriores y no hizo falta comprarlas para este.

- **Software:** Raspbian Buster, Python, C, C++, STM32CubeIDE, STM32CubeAI, OpenCV, OpenOCD, TensorFlow, WinSCP, SSH.

Costes materiales:

- STM32-L496RGT6 (3 unidades, 2 para la PCB final y una para la PCB de pruebas): 27 € (aprox. 9 € por unidad)
- STM32-G474RET6 (1 unidad): 7,25 €
- Adafruit MAX98357 (1 unidad): 5,14 €
- Raspberry Pi Zero 2W (1 unidad): 14 €
- Raspberry Pi Camera Module 2 (1 unidad): 17,44 €
- Tarjeta microSD 32 GB (1 unidad): 10 €
- Componentes electrónicos varios de la PCB palma: 43,94 €
- Motor DC 6V 35 RPM (5 unidades): 60 € (11,91 € por unidad)
- Fabricación PCB palma: 120 €

- Licencias software: 0 € (todo es software libre o código abierto)

Total estimado materiales: 304,77 €

Costes humanos: Mirando el diagrama de Gantt expuesto anteriormente, se puede observar que la duración del proyecto ha sido de 184 días y con una media de 3 horas diarias de trabajo. Esta media es aproximada debido a que el tiempo dedicado al desarrollo varía bastante, teniendo algunos con un desarrollo nulo y otros de hasta 6 horas. Además, no cuenta solo el trabajo dedicado al software (aunque sí en su mayoría), porque también he tenido que realizar algunas tareas relacionadas con el hardware, como soldadura o montaje. Con toda esta información, tenemos un total de 552 horas.

Sabiendo que el sueldo más bajo que puede ganar un ingeniero informático en España (asumimos un recién titulado) es de 13,33 €/h [37] y con el total de 552 horas anteriores, tenemos un **coste humano** de 7358,16 €.

Costes indirectos: El principal coste indirecto es la electricidad, ya que el resto de elementos que podrían aparecer en este apartado no han necesitado ser comprados porque ya disponíamos de ellos en el laboratorio, como fuentes de alimentación, soldadores, etc. Por tanto, vamos ahora a desglosar el consumo de los principales elementos:

- Raspberry Pi Zero 2W: 0.7 Wh [38]
- Lenovo IdeaPad3 15ITL6 cargando: 30 Wh (aprox.)

Suponiendo que la Raspberry ha estado encendida durante algunas horas de las fases de desarrollo y validación de subsistemas y la integración incremental, puede haber estado consumiendo durante un cuarto de las horas, es decir, 138 horas. Por otro lado, el portátil habrá estado consumiendo electricidad a pagar durante las horas de carga, que se estima que sean un tercio de las horas totales, es decir, 184 horas.

Con esto y sabiendo que el coste medio de la electricidad durante un día de agosto en España es de 0,094 kWh [39], tenemos que el **coste indirecto** de la electricidad ha sido de 0,60 € aproximadamente.

Por lo tanto, el **coste total estimado del proyecto ha sido de 7663,53 €**.

3.5. Riesgos

A continuación, se enumeran los principales riesgos del proyecto, con su posible impacto, la probabilidad de ocurrencia y el plan de mitigación.

3.6. Tipos de pruebas a realizar

Para asegurar una calidad aceptable del sistema, se realizarán pruebas que garanticen su correcto funcionamiento e integración, documentadas en la sección 8:

Riesgos			
Riesgo	Impacto	Probabilidad	Mitigación
Fallo de integración	Alto	Alta	Pruebas incrementales
Fallo hardware	Medio	Media	Repuestos y pruebas previas
Pérdida de datos	Alto	Baja	Copias de seguridad
Retraso en el calendario	Medio	Media	Planificación y previsión
Incompatibilidad hard/soft	Alto	Media	Prototipos previos

Tabla 3.2

Riesgos del proyecto

- **Pruebas de funcionamiento:** Estas pruebas serán las que se llevarán a cabo para aprender a usar o comprobar el funcionamiento de elementos hardware, como circuitos integrados específicos y sensores.
- **Pruebas unitarias:** Serán las pruebas para comprobar que cada uno de los subsistemas que se desarrollen funciona por sí solo y de la manera que se espera de él.
- **Pruebas de integración:** Estas pruebas permitirán evaluar si cada uno de los subsistemas es capaz de comunicarse con el sistema central y si el sistema central es capaz de utilizar la información que estos le brindan para efectuar una colaboración entre los mismos.

Parte II

Desarrollo

4. Análisis del Sistema

En este capítulo se cubre el análisis del sistema a desarrollar, comentando los trabajos de los que se parte, qué cambios se realizarán para adaptar su funcionalidad a la que actualmente se desea y cuál será el funcionamiento de cada subsistema para cumplir con el alcance y objetivos comentados en la sección 1.2.

4.1. Trabajos previos y cambios realizados

Como ya se observó en la sección 3.1.1, en este proyecto no se parte de la nada, sino que se aprovecha parte del trabajo realizado previamente por otros compañeros para conformar la funcionalidad total de la prótesis. Por supuesto, gran parte de estos trabajos necesita cambios para adaptarlos a nuestros nuevos objetivos, arquitectura, microcontroladores, elementos de sensado y actuadores, etc., teniendo que rehacer algunos, y eso es lo que se explicará en esta sección.

4.1.1. Análisis y adaptación del trabajo sobre control de motores, comunicación con Raspberry y arquitectura

Análisis del trabajo previo

Este trabajo es uno de los más potencialmente aprovechables, debido a que en él se tuvo una primera concepción de la integración de distintos subsistemas en la prótesis y de ahí surgió la arquitectura que pudimos ver en la Figura 3.7. Era una arquitectura modular que contemplaba la adición de distintos elementos para colaborar entre ellos y que se aprovechará como concepto inicial para nuestra arquitectura, aunque de esto se hablará más en detalle en la sección 5.1.

Esto llevó consigo una estructura de comunicación entre la Raspberry y otros subsistemas, en concreto el que se encarga de los motores, algo que puede ser reutilizado y escalado sin ningún tipo de problema en nuestro proyecto para la comunicación entre todos los elementos.

Hablando de motores, otro de los principales aspectos de este trabajo fue el control de los mismos con el uso de un driver, sensores *Hall*, que se encargaban de conocer la posición relativa de cada dedo, y finales de carrera, para saber si se había llegado a la posición inicial o final. Las posiciones de cada uno de los sensores, al ser relativas, se escribían en memoria no volátil (EEPROM) para que, cuando se reiniciara el sistema, se siguiera conociendo la posición de cada dedo. Además, el microcontrolador empleado para este control fue un AVR32DA32.

Con respecto al *software* asociado a este control de motores, en el trabajo previo se empleó la estructura de clases que podemos ver en la figura 4.1. En ella, la clase *Hand-*

Control sirve como entidad de control central, responsable de gestionar la comunicación y las tareas de control. Está compuesta por 5 instancias de la clase *Finger*, una por cada motor (meñique/anular, corazón, índice, pulgar lineal y pulgar rotativo). La clase *Finger* encapsula los atributos y métodos del motor de un dedo, proporciona control de la articulación a bajo nivel y obtiene su configuración de la estructura *FingerConfig*. Además, emplea la clase de alto nivel *Tle94112Motor* para controlar el driver de los motores, que a su vez usa la clase *Tle94112*, de más bajo nivel, proporcionada por el fabricante. Los métodos de prueba están encapsulados en una clase separada, llamada *TestHandControl*, con el objetivo de facilitar su activación/desactivación a través del uso de las directivas de preprocesado de C/C++.

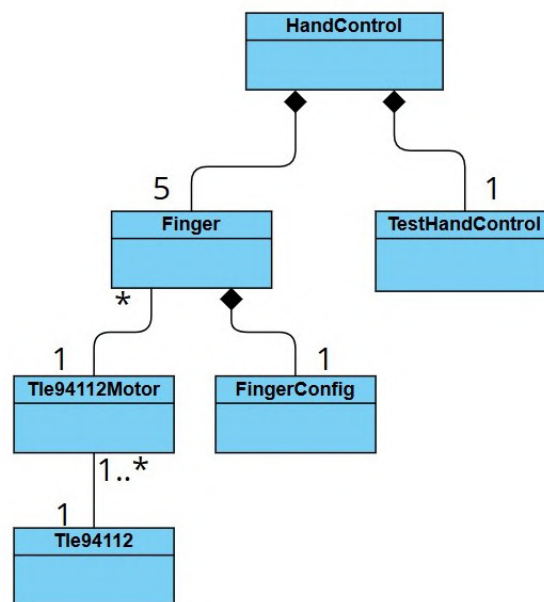


Figura 4.1: Diagrama de clases para el control de motores usado en trabajo previo y reutilizado en el actual

Adaptaciones

En cuanto al *hardware*, el primer cambio será la sustitución del microcontrolador AVR32DA32 por un STM32L496RGT6 por distintas razones:

- Mayor capacidad de cómputo, con frecuencia máxima de 80 MHz y arquitectura de 32 bits frente a los 24 MHz y 8 bits del AVR.
- Ultra bajo consumo de la serie STM32L, con distintos modos llegando a consumir menos de 1 μ A en *standby*.
- Mayor memoria y capacidad para algoritmos complejos, con multitud de periféricos y capacidades específicas para IA.

Aunque no aprovechemos todas estas ventajas en el proyecto actual, la elección de este microcontrolador permite incluir mejoras a futuro que requieran mayor potencia o capacidades (como IA) sin tener que sustituirlo, pudiendo aprovechar todo el trabajo previo que se realice.

El segundo cambio importante es el uso de un sensor de posición absoluto, un potenciómetro. Con esto, nos eliminamos el tener que emplear una memoria no volátil y los sensores finales de carrera. Además, por el nuevo diseño mecánico que se está realizando, los dedos meñique y anular compartirán un motor, así que ahora nos bastará con 5 motores en lugar de 6.

Por parte del *software*, hay varios cambios importantes:

- Estructura de clases de la Raspberry: Se desarrollará la estructura de clases que podemos ver en el diagrama de la Figura 4.2 y que se implementará en la Raspberry para la comunicación con cada subsistema. Como vemos, se parte de una clase Control, que es la principal y en ella se encuentra una clase por cada subsistema. Cada una de estas clases pretende ofrecer funciones para el manejo de su subsistema asociado. Además, vemos cómo en algunas de ellas está el componente I2cDevice. Este componente es esencial, pues es el encargado de realizar la comunicación I²C entre el sistema central y cada uno de los subsistemas que usan este bus.
- *Software* asociado al control de motores: Después de su estudio, se ha llegado a la conclusión de que la estructura de clases planteada en el trabajo previo es una estructura limpia y lógica, que da claridad acerca de cómo se maneja el movimiento completo de la mano. Es por ello que se ha decidido reutilizarla en gran parte, con cambios para adaptarse a la nueva funcionalidad y al nuevo hardware. Los cambios serán principalmente asociados al uso de C y C++ en lugar de lenguaje Arduino y de las funciones HAL de STM32 para el manejo de alto nivel de los periféricos empleados, añadiendo también los cambios para el uso de potenciómetros en lugar de sensores *Hall* y finales de carrera.

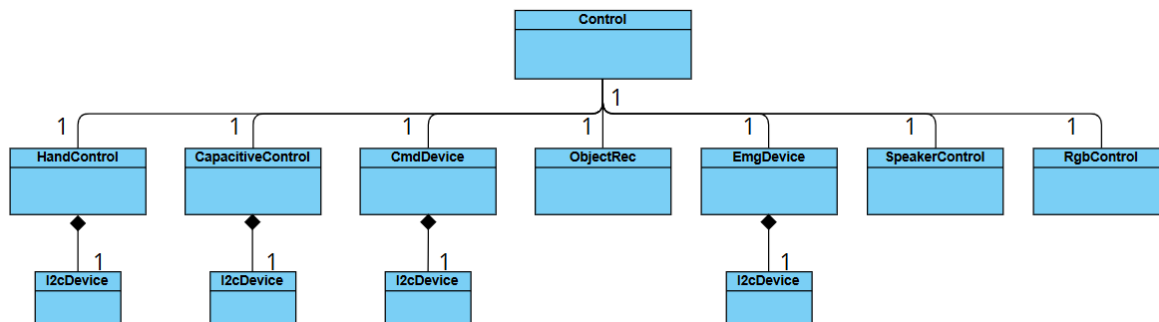


Figura 4.2: Diagrama de clases implementado en la Raspberry para el manejo de cada subsistema en nuestro trabajo actual

4.1.2. Análisis y adaptación del trabajo sobre reconocimiento de voz

Análisis del trabajo previo

El objetivo de este proyecto era implementar un sistema de reconocimiento de voz basado en la detección de palabras clave (*KWS*, *Keyword Spotting*) mediante *Deep*

Learning en un microcontrolador, para conseguir un sistema *edge ai*. Tras un estudio exhaustivo de distintas alternativas, se decidió usar como método de extracción de características y, por tanto, entrada del modelo, el espectrograma logarítmico de Mel, por ser el método más usado en sistemas tipo *deep KWS* [40]. Para la obtención del modelo de *Deep Learning*, se llevó a cabo una evaluación de 3 arquitecturas con MATLAB: *CNN*, *CLDNN* y *CGDNN*. Como *datasets* para el entrenamiento, se empleó un subconjunto del *Multilingual Spoken Words Corpus (MSWC)*, que tiene grabaciones de más de 24800 palabras en idioma español realizadas en ambientes naturales y *UrbanSound8k*, con grabaciones de ruidos para que los modelos puedan aprender la ausencia de voz humana y así proporcionar robustez. Se usaron 5 palabras clave para el entrenamiento, que consistían en los dígitos del 1 al 5 en español y 7 clases de salida, que identificaban cada una de estas palabras y 2 extras, una para identificar ruido de fondo y la otra para salidas desconocidas. El estudio arrojó los resultados que podemos observar en la tabla 4.1.

Modelo	Exactitud	Error de entrenamiento	Error de validación	Tamaño en memoria
Basado en CNN	91,49 %	1,53 %	8,51 %	278,24 kB
Basado en CLDNN	92,41 %	0,58 %	7,59 %	2850,60 kB
Basado en CGDNN	92,56 %	0,21 %	7,44 %	2207, 53 kB

Tabla 4.1

Resultados del entrenamiento de los modelos de reconocimiento de voz evaluados en el proyecto previo.

A la vista de los resultados anteriores, se escogió el modelo basado en CNN porque, aunque tenía una exactitud ligeramente menor, los otros dos modelos ocupaban una memoria que los hacía incompatibles con la mayoría de microcontroladores si no se incluía memoria RAM o Flash externa. La matriz de confusión obtenida para este modelo ya la vimos en la Figura 3.9.

En cuanto al aspecto *hardware*, se decidió emplear un microcontrolador de la familia STM32 debido a que cuentan con periféricos *I2S* y *PDM*, útiles para establecer comunicación de manera sencilla con las interfaces de salida de micrófonos *MEMS*. Esta decisión favoreció el uso de la herramienta ofrecida por el propio STM llamada STM32CubeAI, una *suite TinyML* para la optimización del modelo de *Deep Learning* con vistas a su implementación en el microcontrolador. Además, STM32Cube.AI genera código en lenguaje C compatible con los microcontroladores de esta familia a partir de modelos de *Deep Learning* provenientes de una variedad de *frameworks*, entre ellos MATLAB.

Adaptaciones

Como microcontrolador, se ha decidido utilizar en el proyecto actual el mismo que ya se empleó en este proyecto previo, el STM32-G474RET6. La decisión de seguir

con la familia STM32 se lleva a cabo por las mismas razones que ya se dieron en este trabajo previo. Por otro lado, el seguir empleando el mismo microcontrolador o cambiar fue una decisión que se tuvo que valorar más. Para ello, usamos la herramienta en línea STM32CubeAI Cloud, que es capaz de medir el rendimiento de un modelo entrenado en distintos microcontroladores de STM y que podemos ver en la Figura 4.3. En ella, vemos cómo el G4 no tiene el menor tiempo de inferencia, pero es un tiempo bastante aceptable para lo que se quiere realizar. Otra opción que consideramos fue sustituirlo por un L496RGT6 y así tener el mismo microcontrolador en todos los subsistemas, pero vemos cómo el tiempo de inferencia es mayor al ser la gama L dedicada a bajo consumo y tener menor frecuencia máxima (80 MHz del L496RGT6 frente a los 170 MHz del G474RET6), algo que no nos interesa en este caso. Además, en el proyecto previo, el G474RET6 tuvo un desempeño muy bueno, así que finalmente decidimos conservarlo.

En cuanto a los cambios *software*, podemos considerar varios aspectos:

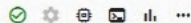
- Modelo de *Deep Learning*: Tras el exhaustivo estudio realizado por el compañero que llevó a cabo este trabajo para conseguir el modelo que mejor se adecuase al sistema de reconocimiento de voz, se ha considerado que todo lo relacionado al mismo permanezca inalterado, por todas las justificaciones dadas por él en su proyecto y comentadas aquí y porque el desempeño final fue el esperado. Además, tampoco se ha considerado modificar el entrenamiento, porque con las 7 clases nos es suficiente para cumplir los objetivos de nuestro sistema.
- Código del microcontrolador: El código recibirá muchos cambios drásticos. En primer lugar, se emplearán modos de bajo consumo al ser este microcontrolador el más potente del sistema. También se añadirán mejoras enfocadas a una mayor abstracción y claridad en el código que se comentarán con detalle en la sección 5.3.4. Lo que principalmente se aprovechará del código previo es el uso de la red neuronal.

4.1.3. Análisis y adaptación del trabajo sobre reconocimiento de objetos mediante visión artificial y feedback de tacto

Análisis del trabajo previo

El trabajo que ahora nos toca tratar se centraba en el desarrollo e implementación de un sistema de visión artificial y retroalimentación sensorial que emplease la prótesis antigua que aparecía en la Figura 3.17. En él se optó por usar OpenCV con el dataset COCO para la detección de objetos, implementado en una Raspberry Pi Zero 2W. Por otro lado, para el *feedback* de tacto, se usaron sensores capacitivos fabricados en el laboratorio (uno en cada yema y otro extra en la palma) y el circuito integrado AD7147 para medir la capacidad al presionar estos sensores. En el caso de que se detectara proximidad o presión, se ponían en marcha unos motores vibradores tipo moneda para dar la información de tacto al usuario.

La arquitectura que se siguió en este proyecto fue el uso de la Raspberry como esclavo I²C del microcontrolador AVR32DA32 encargado de la lectura de los sensores

	STM32H7 S78-DK Appli Arm Cortex-M7 800 MHz Available internal RAM for AI: 455 KB Total internal RAM: 620 KB External RAM: 16 MB Internal flash: 64 KB External flash: 128 MB	 Measured inference time 30.41 ms
	STM32H735G-DK Arm Cortex-M7 550 MHz Available internal RAM for AI: 560 KB Total internal RAM: 594 KB External RAM: 16 MB Internal flash: 1024 KB External flash: 64 MB	 Measured inference time 28.20 ms
	NUCLEO-H743ZJ2 Arm Cortex-M7 480 MHz Internal RAM: 1024 KB Internal flash: 2048 KB	 Measured inference time 32.92 ms
	STM32H747I-DISCO CortexM7 Arm Cortex-M7+M4 Dual 400 MHz Available internal RAM for AI: 704 KB Total internal RAM: 1024 KB External RAM: 8 MB Available internal flash for AI: 1024 KB Total internal flash: 2048 KB External flash: 128 MB 256-Mbit SDRAM	 Measured inference time 38.86 ms
	STM32H7B3I-DK Arm Cortex-M7 280 MHz Internal RAM: 1184 KB External RAM: 16 MB Internal flash: 2048 KB External flash: 64 MB 128-Mbit SDRAM	 Measured inference time 56.38 ms
	STM32H573I-DK Arm Cortex-M33 250 MHz Internal RAM: 640 KB Internal flash: 2048 KB External flash: 64 MB	 Measured inference time 89.92 ms
	STM32F769I-DISCO Arm Cortex-M7 216 MHz Internal RAM: 512 KB External RAM: 16 MB Internal flash: 2048 KB External flash: 64 MB	 Measured inference time 72.74 ms
	STM32F746G-DISCO Arm Cortex-M7 216 MHz Internal RAM: 320 KB External RAM: 16 MB Internal flash: 1024 KB External flash: 16 MB 128-Mbit SDRAM (64 Mbits accessible)	 Measured inference time 74.15 ms
	B-US85I-IOT02A Arm Cortex-M33 180 MHz Available internal RAM for AI: 768 KB Total internal RAM: 768 KB Internal flash: 2048 KB External flash: 64 MB	 Measured inference time 137.5 ms

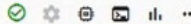
	NUCLEO-G474RE Arm Cortex-M4 170 MHz Internal RAM: 128 KB Internal flash: 512 KB	 Measured inference time 171.6 ms
	STM32F469I-DISCO Arm Cortex-M4 180 MHz Internal RAM: 384 KB External RAM: 16 MB Internal flash: 2048 KB External flash: 16 MB	 Measured inference time 152.4 ms
	NUCLEO-F401RE Arm Cortex-M4 84 MHz Internal RAM: 96 KB Internal flash: 512 KB	 Measured inference time 299.2 ms
	STM32L4R9I-DISCO Arm Cortex-M4 120 MHz Internal RAM: 640 KB Internal flash: 2048 KB External flash: 64 MB	 Measured inference time 249.5 ms

Figura 4.3: Rendimiento del modelo de reconocimiento de voz en distintos microcontroladores STM32

capacitivos que, a su vez, también era maestro del AVR32DA32 que se encargaba del movimiento de los motores de los dedos de la prótesis de la Figura 3.17.

Con todo esto, el funcionamiento era el siguiente:

- El microcontrolador encargado de leer los sensores capacitivos los leía y hacía que vibrasen o no los motores vibradores en base a la lectura.
- En el caso de que se pulsase un botón, se enviaba un mensaje I²C a la Raspberry para que comenzase el reconocimiento de objetos y devolviese el objeto detectado.
- En función del objeto detectado, se enviaba un comando distinto por I²C al otro AVR32DA32 encargado de mover los dedos.

Adaptaciones

Por supuesto, para nuestro trabajo aprovecharemos ciertas partes de lo que acabamos de comentar, pero con los siguientes cambios:

- Arquitectura: Aunque se detallará en el apartado de diseño, si nuestro sistema integra muchas otras funcionalidades, no tiene sentido que el maestro de las comunicaciones sea el microcontrolador encargado de los sensores capacitivos, sino la Raspberry.
- Hardware: Cambiaremos el microcontrolador a uno de la familia STM32, en concreto el STM32-L496RGT6, el mismo que el empleado para el control de motores y por las mismas razones que dimos en su caso. Por otro lado, seguiremos usando sensores capacitivos y con el mismo circuito integrado para leerlos por los motivos que ya explicamos en la sección 3.3. Mantendremos la Raspberry Pi Zero 2W para el reconocimiento de objetos.
- Software: Al cambiar la arquitectura y el microcontrolador, deberá haber cambios en el software. Primero a nivel de lenguaje, ya que se empleó Arduino y nosotros usaremos C/C++. También a nivel de funcionamiento del software, que ya comentaremos en la siguiente sección. Asimismo, se desarrollarán mejoras del código del microcontrolador, concretamente incluyendo orientación a objetos para incrementar la claridad, separación de tareas y la facilidad de uso. El diagrama de clases lo podemos ver en la Figura 4.4. En ella, tenemos la clase *AD7147*, que es la encargada de todo lo relacionado con el circuito integrado que lee los sensores capacitivos (configuración, sensibilidad, comunicación, etc.) y la clase *CapacitiveControl*, que es la entidad de control central, responsable de gestionar la comunicación con el maestro y el procesado de los datos provenientes del integrado que lee los sensores. También, como en el caso de los motores, hemos encapsulado los métodos de prueba en la clase *TestCapacitiveControl*. Por otro lado, el software de reconocimiento de objetos usará las funciones propias de la biblioteca OpenCV, tomando algunos apartados realizados por el compañero, pero mejorando su código y añadiendo métodos para su correcta integración. Por último, ahora este microcontrolador no se va a encargar de proporcionar el feedback con los motores vibradores, ya que estarán instalados en el subsistema de sensores EMG. De este modo, el microcontrolador solo se encargará de informar

a la Raspberry del tacto de cada sensor cuando exceda un límite y la Raspberry decidirá si comunicar al subsistema de sensado EMG si debe hacer vibrar algún motor o no.

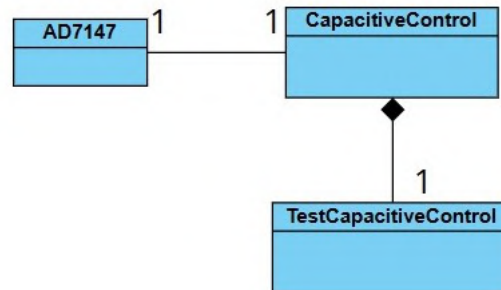


Figura 4.4: Diagrama de clases para el control de sensores capacitivos desarrollado para el trabajo actual

4.1.4. Análisis y adaptación del trabajo sobre sensores EMG

Análisis del trabajo previo

Este es el último trabajo que incluiremos como subsistema de nuestra prótesis. Su objetivo era conseguir medir señales EMG para controlar el movimiento de una mano robótica. Para ello, desarrollaron PCBs llamadas de amplificación, que se encargaban de recoger estas señales, extraer sus características y pasarlas todas ellas a la PCB principal mediante I²C en *daisy chaining* (véase Figura 3.14 para más información sobre la comunicación I²C), la cual las introducía en una red neuronal que fuese capaz de clasificar el movimiento realizado. Todas estas PCB se introducían en carcassas fabricadas mediante impresión 3D, de tal forma que formasen un brazalete alrededor del antebrazo.

Para la PCB principal optaron por usar el microcontrolador STM32L496RGT6, el mismo que decidimos emplear nosotros para el control de motores y sensores capacitivos. Esta decisión la tomaron por su relativamente alta frecuencia de reloj, bajo consumo y alta capacidad de almacenamiento en SRAM, que permitiría ejecutar una red neuronal. Para las PCB de amplificación, cada una llevaba un STM32L431, menos potente en características que el anterior, pero con 64 kB de memoria SRAM, que sería suficiente para almacenar bastantes muestras del ADC de 12 bits.

Para la recogida de características para la red neuronal, empleaban la técnica de *sliding window*, en la que el flujo de datos completo se separa en ventanas que se solapan y las características se calculan para cada una de estas ventanas. Implementaron 19 características, de las cuales 17 eran realmente relevantes.

En cuanto al dataset empleado, no se encuentran disponibles datasets de señales EMG con las características implementadas, por lo que usaron la herramienta

STM32CubeMonitor para desarrollar un flujo que les permitiera comenzar el proceso de muestreo y almacenar las características en un formato apropiado de .csv. El *dashboard* del flujo lo podemos ver en la Figura 4.5.

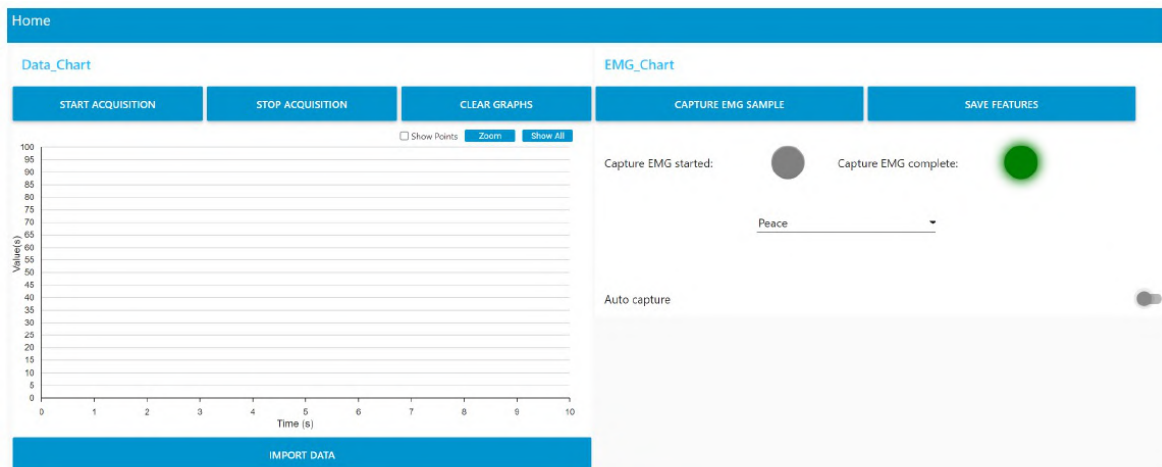


Figura 4.5: *Dashboard* del flujo creado en STM32CubeMonitor

Finalmente, para el entrenamiento crearon un *script* en Python empleando TensorFlow y Keras y con el *dataset* anterior, desarrollaron una red neuronal para entrenar 4 posturas: puño cerrado, mano abierta, pulgar y símbolo de la paz (Figura 4.6). Utilizaron 4 amplificadores (1,2,6,7) y no más porque es lo que puede caber en el antebrazo de un niño. El modelo entrenado se exportó a formato *.tflite* y con la herramienta STM32Cube.AI se introdujo en el microcontrolador principal.

Adaptaciones

En cuanto a los cambios a este proyecto, debido a que nos llegó recientemente de Alemania y se desarrolló de manera paralela al nuestro, no habrá demasiados:

- Con respecto al *hardware*, no se realizará ningún cambio, debido a que este sub-sistema estaba diseñado de antemano pensando en integrarse en nuestra prótesis y su diseño hardware ha sido consensuado con nosotros. Simplemente se soldarán motores vibradores de tipo moneda a las PCB auxiliares para implementar la realimentación del tacto.
- Por parte del *software*, la parte de comunicación entre sus PCB mediante I²C es correcta y no requiere de ningún cambio ni ajuste. Con adquisición de señales EMG y extracción de características ocurre lo mismo, así como con la ejecución de la red neuronal en el microcontrolador principal. Los cambios tendrán que ver con incluir en el código del microcontrolador principal la comunicación por otro puerto I²C con nuestro sistema central para que sea este el que le diga cuándo tiene que avisar a los microcontroladores auxiliares para que comiencen el reconocimiento. Además, también se modificará la trama I²C entre la Raspberry y el microcontrolador principal y entre este y los auxiliares para incluir en el mensaje si algún motor vibrador debe actuar o no en función del tacto detectado. Y es



Figura 4.6: Clases de entrenamiento para subsistema EMG

que ahora los motores vibradores estarán en cada una de las PCB amplificadoras. También se realizarán ajustes y mejoras en el *script* de python usado para crear y entrenar la red neuronal, como preprocesamiento de los datos mediante escalado, *resample* para equilibrar las muestras con muestras artificiales, cambio en la estructura y número neuronas (ya que ahora tiene 5 clases de salida cuando solo ha entrenado 4) y más aspectos que se explicarán en el apartado 5.3.5.

4.2. Funcionamiento de cada subsistema

En esta sección mencionaremos cada uno de los subsistemas que vamos a tener en la prótesis y explicaremos el funcionamiento de cada uno de ellos, mediante diagramas de flujo. Toda la explicación dada aquí ya se refiere a los subsistemas desarrollados por nosotros, no a los trabajos previos.

4.2.1. Subsistema de control de motores

El funcionamiento que se propone es el que podemos ver en el diagrama de la Figura 4.7. En primer lugar, inicializaremos los motores y leeremos la posición en la que se encuentran. Luego, entraremos en un bucle que consistirá en comprobar si se ha recibido un nuevo comando de posiciones. En caso afirmativo, decodificaremos el mensaje para obtener las posiciones a las que debe moverse cada motor y comenzaremos a moverlo o, si debe parar, lo pararemos. Se haga o no este paso anterior, a continuación

actualizaremos la posición de cada dedo y comprobaremos si alguno ha alcanzado su límite, en cuyo caso pararemos el motor. Tras esto, actualizaremos el mensaje de respuesta al maestro con las posiciones de cada dedo y comprobaremos si alguno sigue moviéndose. Finalmente, verificaremos si se cumple alguna de las condiciones para avisar al maestro de que debe leer nuestro mensaje. Las condiciones serán que el dedo se haya parado por una orden externa o que haya alcanzado su límite.

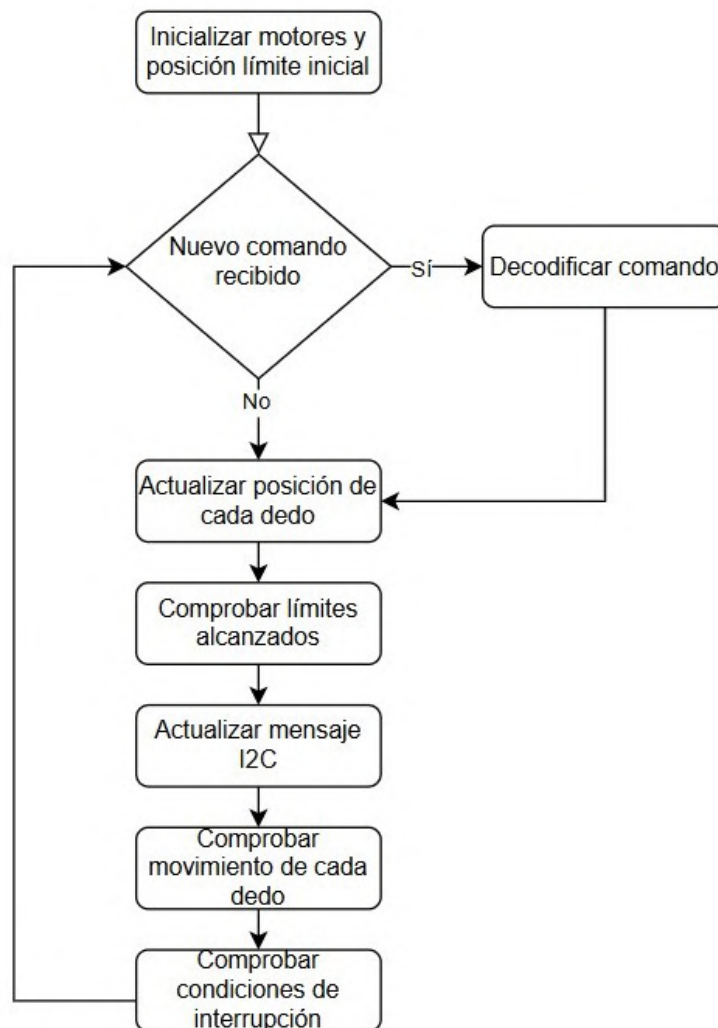


Figura 4.7: Diagrama de flujo del subsistema de control de motores

4.2.2. Subsistema de reconocimiento de voz

El diagrama de flujo de la Figura 4.8 muestra cómo, mientras no se pulse el botón, no se activa el reconocimiento de voz y el microcontrolador entraría en modo de bajo consumo. Si se despierta y el botón está pulsado (el propio botón es el que lo despierta), se enciende el LED para indicar que se puede hablar al micrófono y este comienza a recoger datos. En el momento en el que los datos actualmente escuchados se hayan transferido a la mitad a un *buffer*, se procesan esos datos. Lo mismo ocurre con la mitad superior de la transferencia, es decir, cuando se ha hecho una transferencia

completa. Tras esto, tendremos los datos suficientes para completar un *frame* de audio, el cual podremos procesar mediante el espectrograma logarítmico de Mel para obtener las características de esa muestra. Este proceso será el que se siga haciendo hasta que tengamos 1 segundo de audio, en cuyo caso apagaremos el LED para indicar que ya no se está escuchando por el micrófono y pasaremos las características como entrada de la red neuronal, la cual nos dará el resultado del comando reconocido y su probabilidad. Estos datos se transmitirán al maestro y, una vez completada esta transmisión, dormiremos al microcontrolador introduciéndolo en un modo de bajo consumo. Solo se despertará de este modo si se pulsa de nuevo el botón, momento en el que se ejecutará todo lo explicado hasta ahora. La comprobación de si el botón está pulsado tras despertar el microcontrolador es para diferenciar si se ha despertado de un modo de bajo consumo o es por la primera vez que el microcontrolador ha recibido alimentación. Si está despierto y el botón no está pulsado es por este último caso, con lo cual procederemos a dormirlo hasta que se despierte por una pulsación del botón.

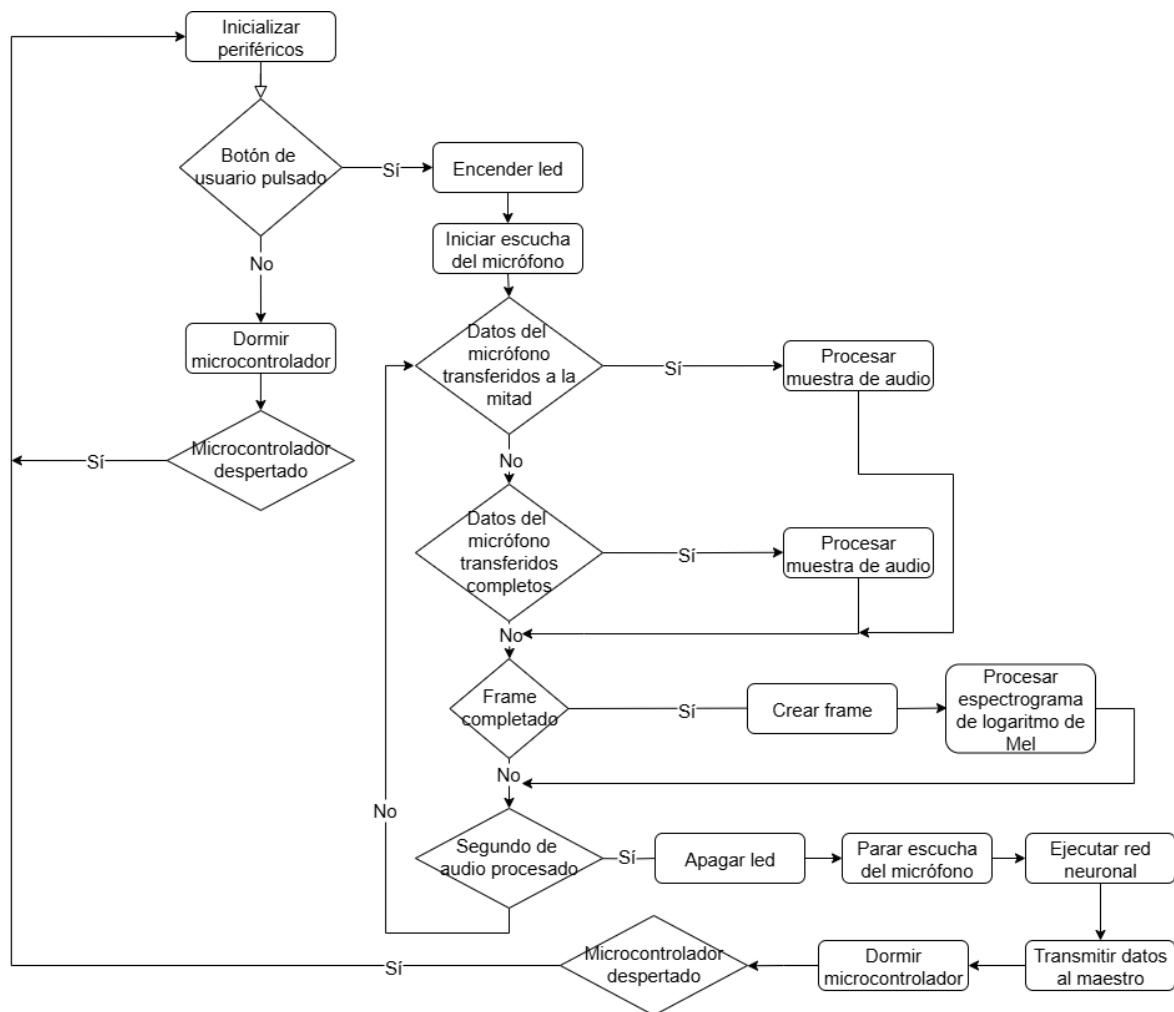


Figura 4.8: Diagrama de flujo del subsistema de reconocimiento de voz

4.2.3. Subsistema de sensado de tacto

Para este subsistema debemos analizar el diagrama de la Figura 4.9. En él observamos cómo, tras hacer las inicializaciones pertinentes, leemos el valor de cada sensor y lo establecemos como valor base, que significa el valor en reposo del sensor, cuando no está siendo presionado. Esto se hace porque, cada vez que se inicia este subsistema, por condiciones del propio ambiente, como humedad, temperatura, etc., el valor inicial que tomamos puede ser distinto y, por tanto, no podemos dejarlo predefinido de antemano. A continuación, viene el funcionamiento en bucle del subsistema. Primero leeremos el valor de cada sensor para, seguidamente, calcular la diferencia en valor absoluto entre este y el valor base que obtuvimos previamente, que indica la diferencia de capacidad. Esta información es la relevante de cada sensor, porque es la que nos indica si se está presionando el sensor y cuánto. El valor por sí mismo que se obtiene de leer el AD7147 no tiene significado, sino que el significado lo da la diferencia de valores, que indica la diferencia de capacidad. Sabiendo esto, actualizaremos el mensaje I²C con el valor de cada sensor y en el caso de que alguno haya alcanzado su cota preestablecida, se avisará al maestro para que lea el mensaje que le tenemos preparado.

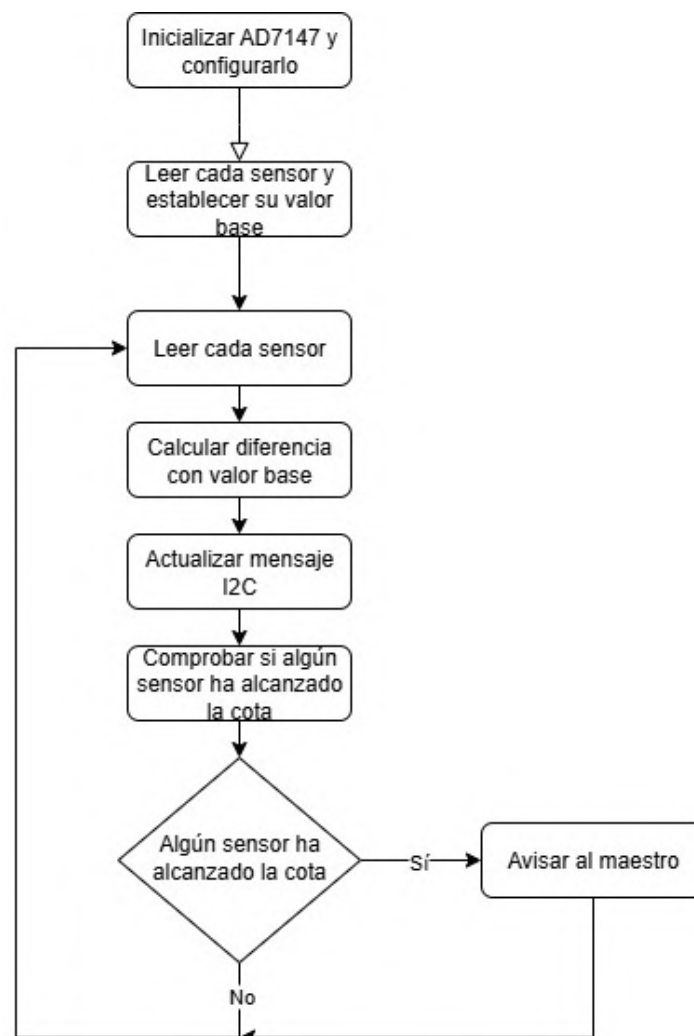


Figura 4.9: Diagrama de flujo del subsistema de sensado de tacto

4.2.4. Subsistema de reconocimiento de objetos

Al estar implementado en Python en la Raspberry, el funcionamiento de este subsistema es más simple, ya que la comunicación con el sistema central es mediante la llamada a una función que emplea otras funciones de la biblioteca de OpenCV. Cuando se llama a la función principal, en un hilo separado comenzará la detección de objetos hasta que se le ordene parar. En esta detección se devolverá de todos los objetos reconocidos en la imagen captada por la cámara, el de mayor calidad. En nuestra función principal lo que haremos es, durante unos segundos, ir leyendo ese mejor objeto y actualizar nuestro mejor objeto si la calidad es mejor. Tras esto, pararemos la detección y devolveremos el objeto que tenemos almacenado como mejor. El funcionamiento de la función principal aparece en el diagrama de la Figura 4.10.

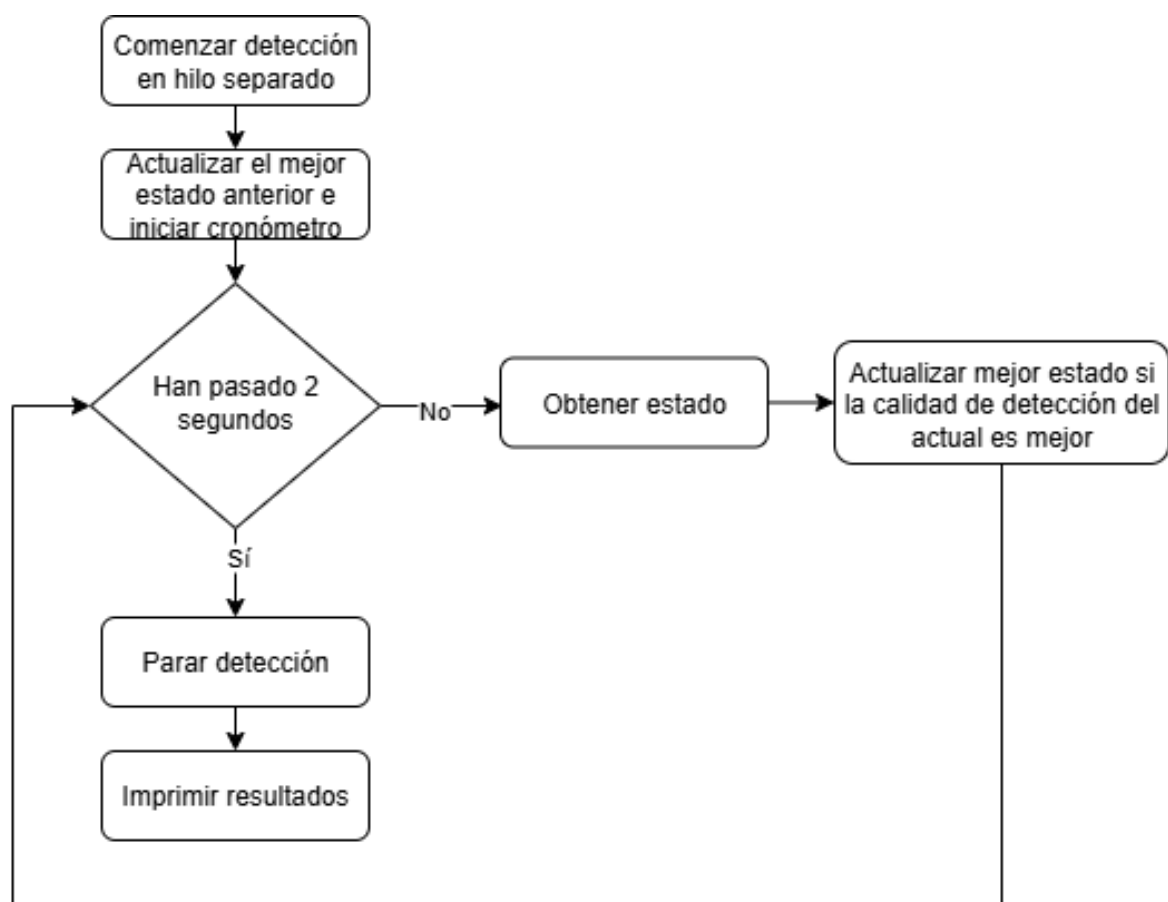


Figura 4.10: Diagrama de flujo de la función llamada desde el programa principal para el reconocimiento de objetos

4.2.5. Subsistema de sensado EMG y retroalimentación de tacto

En este caso tenemos 2 diagramas de flujo, uno para el microcontrolador principal y otro para cada microcontrolador auxiliar (véanse Figuras 4.11 y 4.12).

Comenzando con el principal (Figura 4.11), primero se comprueba si han llegado datos por el bus I²C que va a nuestro sistema central. En caso afirmativo, se comprueba si entre esos datos está la bandera que indica que se debe leer la señal EMG. Si es así, en cada uno de los mensajes que se van a mandar a los microcontroladores auxiliares se inserta esa bandera. De cualquier manera, en esos mensajes también se debe insertar la información de si un motor debe vibrar o no, que también estaba contenida en el mensaje principal que nos llegó desde el maestro I²C y se manda el mensaje a cada uno de los microcontroladores auxiliares. Tras esto, se lee de los microcontroladores auxiliares hasta que todos nos comuniquen que han acabado el cálculo. Esto indicará que han leído las señales EMG y han obtenido las características de ellas. Seguidamente, volveremos a leerlos para que nos manden esas características, que luego normalizaremos para, finalmente, introducirlas en la red neuronal. De sus salidas, nos quedaremos con la clase de mayor porcentaje de detección y esa es la que enviaremos a nuestro sistema central.

En cuanto a los microcontroladores auxiliares (Figura 4.12), en algún momento nos llegará un mensaje. De este mensaje, una parte indicará si el motor debe vibrar o no y otra si debe empezar a leer la señal EMG. En el caso de que se deba empezar a leer, se comenzará a leer y se activará el estado ocupado. Cuando se complete la lectura, se activará el estado de cálculo comenzado y se calcularán las características. Después de esto, se pondrá el estado de cálculo realizado.

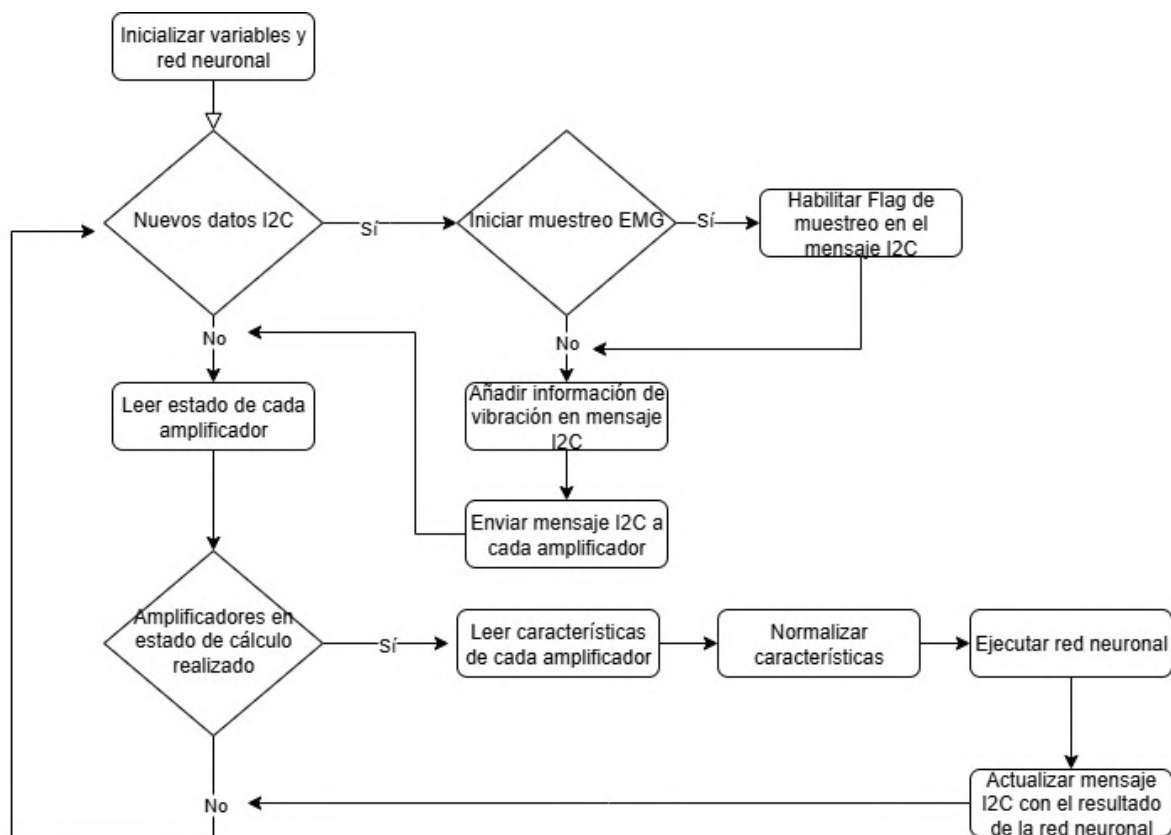


Figura 4.11: Diagrama de flujo del microcontrolador principal del subsistema EMG

Nota: Para los subsistemas de salida de audio e iluminación RGB no se incluyen diagramas de flujo ni se explica su funcionamiento por consistir su uso en el empleo de

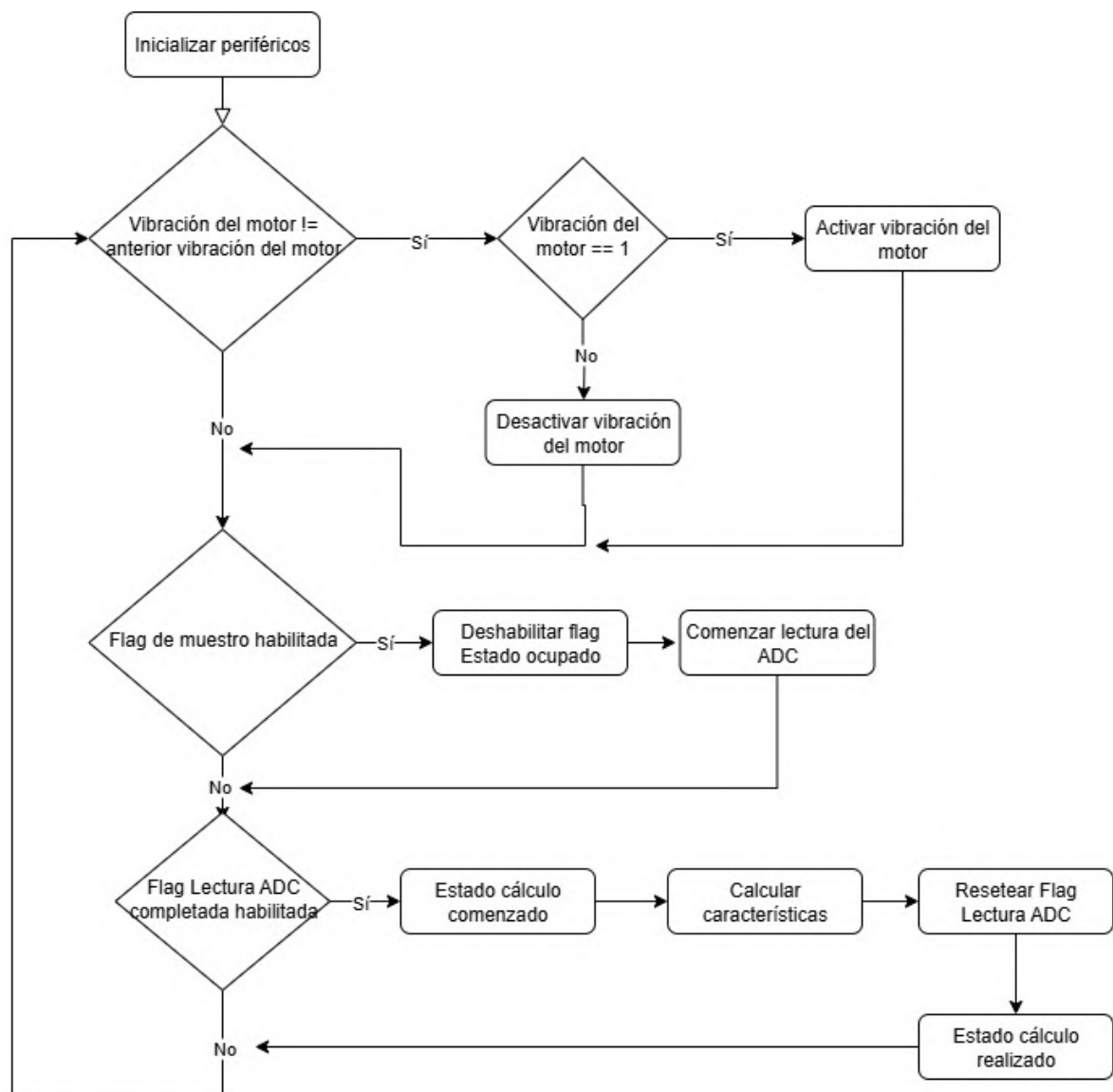


Figura 4.12: Diagrama de flujo de cada microcontrolador auxiliar del subsistema EMG

escasas funciones de bibliotecas.

4.3. Funcionamiento del sistema central

A continuación, explicaremos el funcionamiento del sistema central en colaboración con el resto.

Como podemos ver en el diagrama de flujo de la Figura 4.13, nuestro sistema central funciona como una máquina de estados en la que tenemos los estados: Inicial, Mano, Voz y Cámara.

Al principio, lo que hacemos es inicializar los objetos que representan cada uno de nuestros subsistemas y establecer nuestro estado como inicial. Este estado simplemente se utiliza para configurar las funciones de interrupción que usaremos y lo representaremos con el color blanco en nuestros LED RGB. Al final de este estado, estableceremos el modo ‘Mano’ y colocaremos su color, el rojo.

Para los 3 estados restantes el funcionamiento es el mismo, pero utilizando sus subsistemas específicos. Cambiaremos entre estos 3 estados empleando 3 botones externos, cada uno asociado a uno. Los subsistemas de control de motores y sensado de tacto estarán activados en los 3 estados, como es lógico. En el estado Mano, estableceremos como subsistema el de los EMG, en el estado Voz, el de reconocimiento de voz y en el estado Cámara, el de reconocimiento de objetos. A continuación explicaremos las funciones de actualizar dispositivos y decidir:

- **Actualizar dispositivos:** Primero comprueba si el subsistema de motores le ha avisado para que lea su información, en cuyo caso lo hará, para ver la posición de cada motor. Ahora el funcionamiento dependerá del estado en el que nos encontremos. En el caso de estar en el modo Mano, y se haya pulsado su botón habilitaremos en su mensaje la bandera para comenzar el reconocimiento EMG y le enviaremos dicho mensaje. Si estamos en modo Voz y su subsistema nos ha avisado de que ha terminado el reconocimiento de voz (porque previamente hemos pulsado su botón), leeremos su información. Para el modo Cámara, si hemos pulsado el botón, comenzaremos la detección del objeto.
- **Decidir:** También dependerá su funcionamiento del modo. Si hemos pulsado el botón de modo Mano previamente, leeremos el mensaje que nos tiene preparado y si es distinto al anterior de este modo, moveremos la mano en consecuencia con el tipo de movimiento que le tengamos asignado, abriendo previamente la mano. Si hemos pulsado los botones de modo Voz o Cámara, como ya hemos leído en Actualizar dispositivos su mensaje, haremos lo mismo que en el modo Mano.

Los botones juegan un papel importante, y eso vamos a explicar a continuación:

- Si se pulsa el botón del modo Mano y estábamos en otro estado distinto, cambiaremos el estado, el color de los LED a rojo y por el altavoz sonará que estamos en este modo, activando también una bandera que permitirá identificar en las funciones anteriores que se ha pulsado este botón.

- Si se pulsa el botón del modo Cámara, el funcionamiento es igual que en el modo Mano, pero con los LED en verde y el altavoz con su sonido propio.
- Si se pulsa el botón del modo Voz, el funcionamiento es igual que en los casos anteriores, pero despertando además al microcontrolador de este subsistema para que comience el reconocimiento.

Por último, los subsistemas de motores, sensado de tacto y reconocimiento de voz, son capaces de avisar al sistema principal mediante pines de interrupción de que tienen nueva información lista para él (como ya vimos en sus diagramas de flujo). En el caso de motores y voz, simplemente habilitan una bandera, pero para el tacto, además, hace que lea directamente sus datos y compruebe dos cosas:

- Si alguno de los sensores superó su cota y la mano no se está abriendo, se ordenará parar los dedos asociados al sensor que ha superado su cota.
- Si alguno de los sensores ha superado el 80 % de su cota, se actualizará el mensaje para el subsistema de EMG para que vibren los motores correspondientes y así se tenga realimentación del tacto.

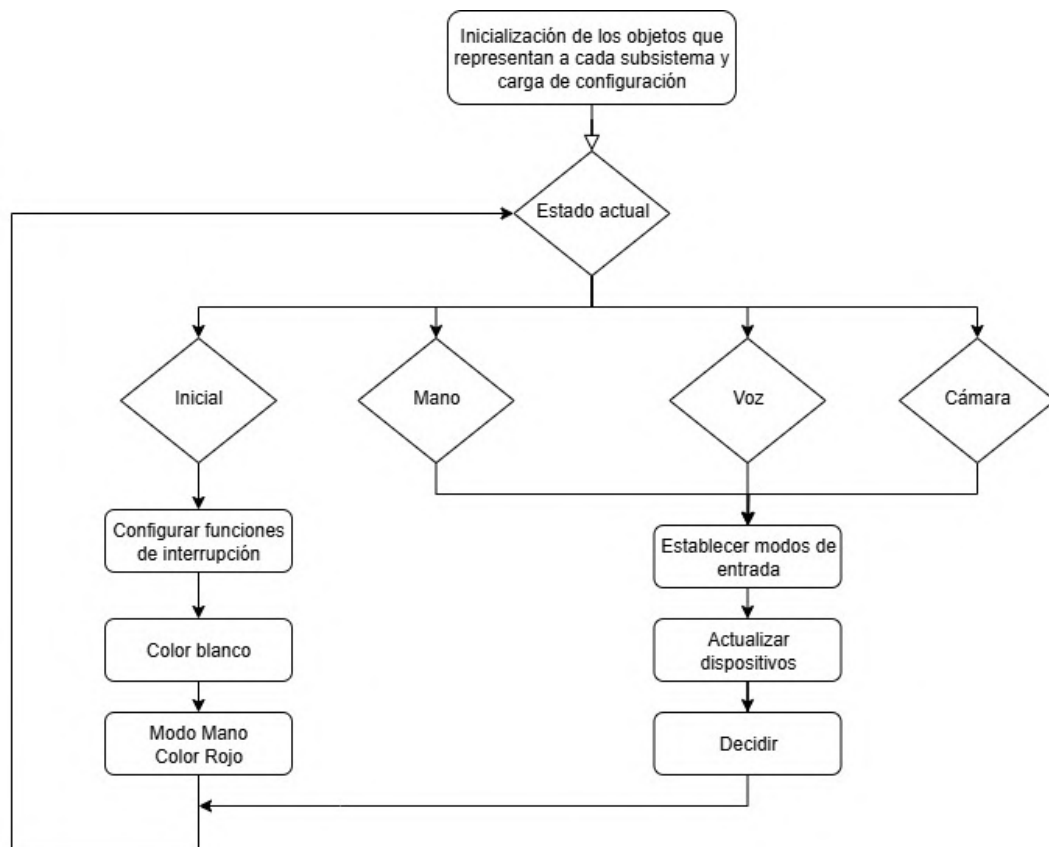


Figura 4.13: Diagrama de flujo del sistema central

5. Diseño del Sistema

En este capítulo se recoge el diseño de la arquitectura del sistema, tanto a nivel *hardware* como *software*, el formato de comunicación entre los distintos elementos y los protocolos empleados para ello. También se incluirá una sección dedicada a las innovaciones y mejoras aplicadas a este proyecto, con respecto a los proyectos anteriores.

5.1. Arquitectura del Sistema

En esta sección se define la arquitectura general del sistema, especificando la infraestructura tecnológica necesaria para dar soporte al software y la estructura de los componentes que lo forman, así como los formatos de comunicación entre los distintos elementos.

La arquitectura es la que aparece en la Figura 5.1. En ella apreciamos cómo la Raspberry Pi Zero 2W es el elemento central, comunicándose con cada microcontrolador de cada subsistema actuando como maestro del bus I²C. La decisión de que ella sea el maestro se ha tomado en base a varios criterios:

- Potencia de cómputo teórica: Aunque los microcontroladores elegidos son potentes, la Raspberry los supera en frecuencia de reloj, con 4 núcleos a 1 GHz cada uno (frente a los 170 MHz máximos de nuestro microcontrolador más potente) y por su arquitectura ARM Cortex-A53 de 64 bits, frente a la ARM Cortex-M4 de 32 bits de nuestros microcontroladores.
- Potencia de cómputo disponible: En este aspecto, la Raspberry gana no solo por la potencia de cómputo teórica, sino también porque es parte de subsistemas simples, como los de iluminación y salida de audio, o activados solamente a petición del usuario, como el subsistema de reconocimiento de objetos. Aunque esto último también ocurre con los subsistemas EMG y de reconocimiento de voz, ambos presentan una memoria más limitada de base (320 KB de SRAM en el mejor caso, frente a los 512 MB de RAM de la Raspberry), que está en gran parte ya ocupada por las redes neuronales que ejecutan.

Vemos, por tanto, que el protocolo I²C es el elegido para la comunicación entre los subsistemas, mientras que el protocolo SPI se ha reservado para la comunicación de los microcontroladores con sus circuitos integrados, como ya justificamos en la sección 3.3.

Algo que se añade para complementar y mejorar la comunicación y que vemos en la Figura 5.1, son líneas de interrupción de los microcontroladores de los subsistemas de control de motores, de sensado de tacto y de reconocimiento de voz hacia la Raspberry. Estas líneas se han diseñado con el objetivo de evitar que nuestro sistema central tenga que leer en cada iteración de su código a cada uno de los sistemas si es que estos no tienen información relevante que proporcionarle. De este modo, solo los leerá por el

bus I²C en el caso de que ellos le reporten, activando una interrupción mediante estas líneas, que tienen información relevante.

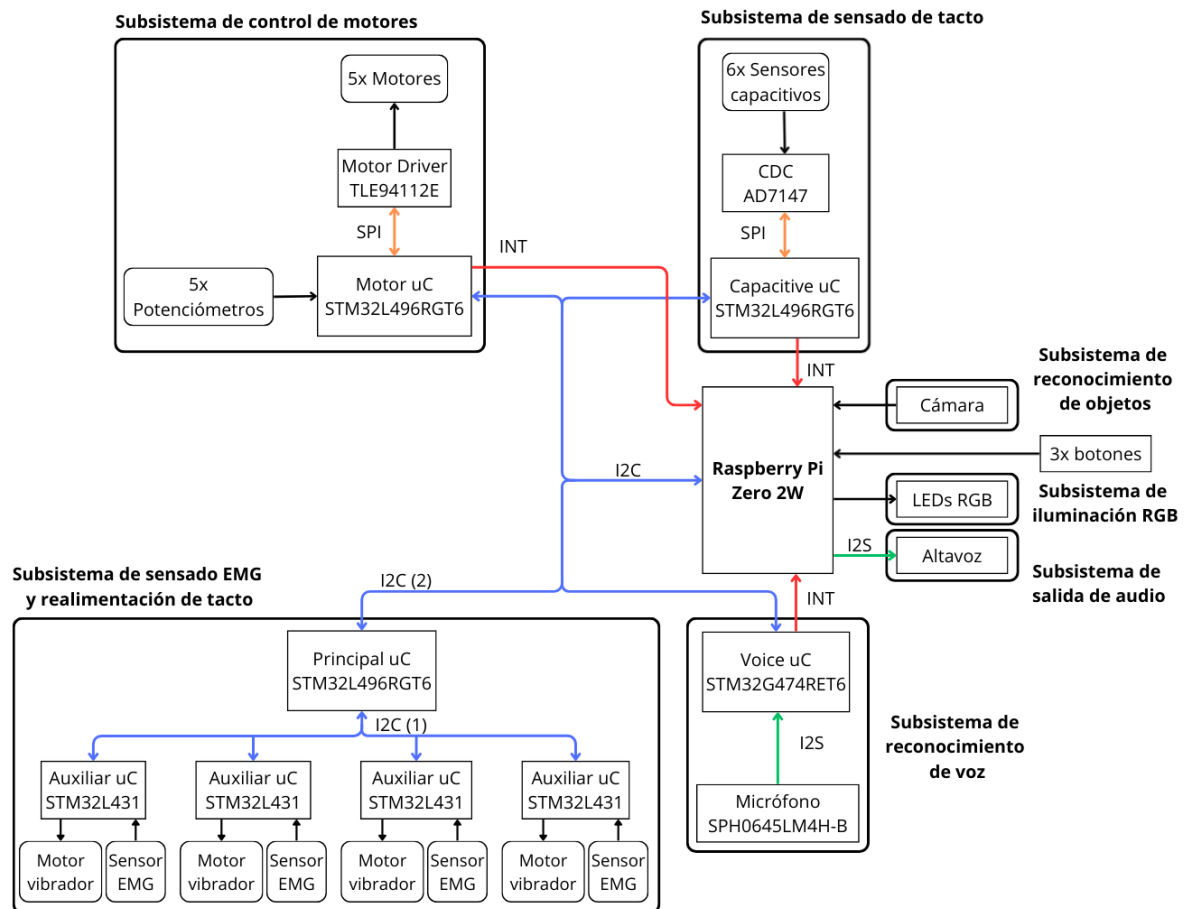


Figura 5.1: Arquitectura del sistema, donde se observan los 7 subsistemas: control de motores, sensado de tacto, reconocimiento de objetos, iluminación RGB, salida de audio, reconocimiento de voz y sensado EMG y realimentación de tacto. También aparece la forma de comunicación entre elementos, como el bus I²C, los buses SPI y los buses I2S. Además, se puede apreciar cómo el subsistema de sensado EMG y realimentación de tacto usa un bus I²C (I²C2) para la comunicación con el sistema central, y otro bus I²C (I²C1) para la comunicación con sus microcontroladores auxiliares.

5.1.1. Comunicación de subsistemas

A continuación, veremos los formatos de comunicación empleados en el bus I²C para comunicar cada subsistema que emplea este bus con el sistema central y también cuándo consideran cada uno de ellos que tienen información relevante y, por tanto, avisan al maestro a través de la línea de interrupción.

Comunicación del subsistema de control de motores con el sistema central

Para comunicar estos dos elementos hemos elegido el formato que se puede ver en la Figura 5.2. La comunicación en este caso es bidireccional, debido a que tenemos actuadores, que son los motores, y debemos decirles cómo queremos que se muevan. En ambos sentidos, la comunicación sigue este mismo formato en el que hay un BCG (Byte de Comando General) y dos bytes para cada motor. El significado sí cambia ligeramente en función del sentido:

- De Raspberry a microcontrolador: Los dos bytes de cada motor indican la posición límite (máxima o mínima) a la que se debe mover el motor. El BCG si es 2, para todos los motores cuyo valor límite en el mensaje sea 5000 y deja a los demás inalterados (siguen moviéndose en el caso de que lo estuvieran haciendo, ignorando su valor en el mensaje) y si es 3, indica la posición límite a la que se debe mover cada motor y los moverá hacia allí, excepto los que su valor sea 5000, que los parará.
- De microcontrolador a Raspberry: El BCG simplemente es el mismo que recibió anteriormente el microcontrolador de la Raspberry y el valor de cada motor es su posición actual, para informar de dónde se encuentra cada uno.

Cada motor necesita 2 bytes (16 bits) en lugar de 1, porque el ADC que lee la posición de cada potenciómetro que está unido a cada motor es de 12 bits, lo que nos da un rango de posiciones de $[0, 4095]$. Al ser 4095 nuestra posición máxima, hemos elegido 5000, un valor fuera de ese rango, para indicar la parada individual.

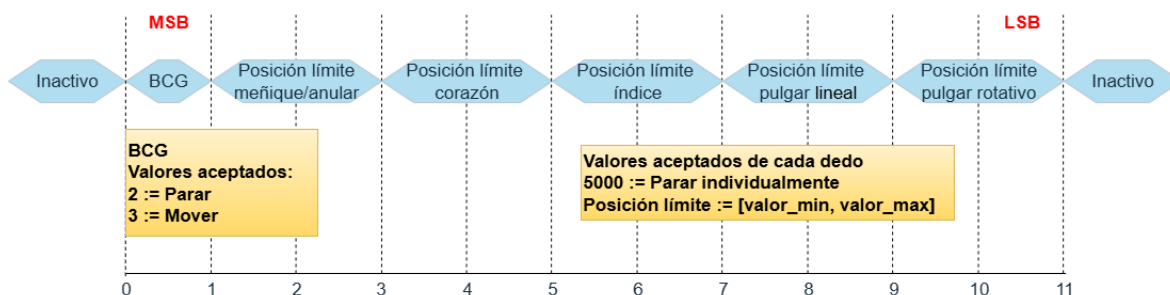


Figura 5.2: Formato de comunicación I²C entre el microcontrolador de motores y la Raspberry

Este microcontrolador considera relevante avisar a la Raspberry mediante su línea de interrupción, cuando alguno de los dedos se ha parado por alguna causa externa, o porque haya llegado a su posición límite. Así, la Raspberry tendrá la información necesaria de la posición de cada dedo.

Comunicación del subsistema de sensado de tacto con el sistema central

En este caso, la comunicación en el bus I²C pasa a ser unidireccional, ya que tenemos sensores y solo queremos saber la lectura de cada uno. Para ello, la comunicación sigue el formato de la Figura 5.3.

Como ya comentamos en la sección 4.2.3, lo interesante de cada sensor es la diferencia del valor que mide actualmente con el valor que tenía el sensor en reposo. Cada entrada del circuito integrado AD7147 nos puede dar un valor de 16 bits, es decir, en el rango $[0, 65535]$. Aunque lo que devolvamos sea diferencia de valores, con 1 byte no nos es suficiente, ya que solo podríamos tener valores en el rango $[0, 255]$. Por ello, se emplean 2 bytes para el valor de cada sensor.

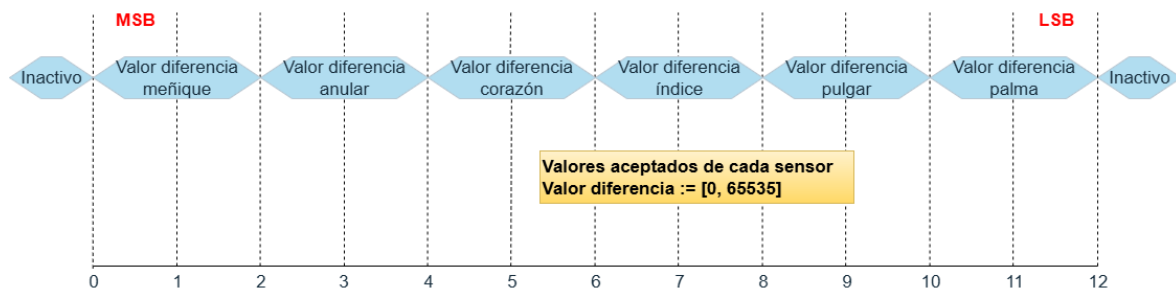


Figura 5.3: Formato de comunicación I²C entre el microcontrolador de sensores de tacto y la Raspberry

Como los sensores tienen pequeñas fluctuaciones de su valor leído debido a las condiciones ambientales y, por tanto, de la diferencia que reportan, este microcontrolador avisa al maestro mediante la línea de interrupción cuando la diferencia de alguno de los sensores con el valor base supera una cierta cota predefinida. Esta cota nos asegura que no se avisa al maestro constantemente, sino solo cuando se produce tacto real. El valor de la cota depende de las características del sensor, de los elementos que tenga alrededor y de la sensibilidad que queramos que tenga. Por ejemplo, si queremos que se avise al maestro cuando el sensor se está aproximando al tacto, el valor deberá ser menor.

Comunicación del subsistema de reconocimiento de voz con el sistema central

En este subsistema, la comunicación I²C también es unidireccional. El formato enviado es mucho más simple y lo podemos ver en la Figura 5.4. Solo necesitamos 2 bytes esta vez: el primero para indicar qué clase ha inferido la red neuronal del subsistema de reconocimiento de voz y el segundo que muestra con qué seguridad ha predicho la clase en formato porcentual.

Por otra parte, aquí la línea de interrupción se activa cuando el microcontrolador ha terminado la inferencia y ha obtenido los resultados.

Comunicación del subsistema de sensado EMG con el sistema central

En este caso, volvemos a una comunicación de tipo bidireccional en el bus I²C, cuyo formato cambia dependiendo de la dirección de la comunicación:

- Comunicación de Raspberry a microcontrolador (Figura 5.5): La trama I²C está formada por 2 bytes. El más significativo (MSB), es una bandera que si es 0x2

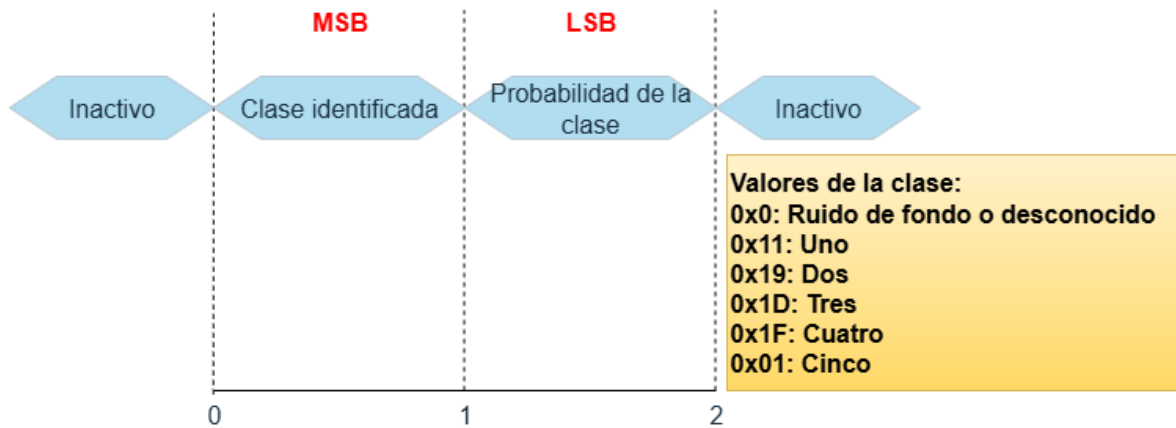


Figura 5.4: Formato de comunicación I²C entre el microcontrolador de reconocimiento de voz y la Raspberry

dará inicio al proceso de sensado EMG y posterior ejecución de la red neuronal, mientras que si es 0x0 no lo realizará. El menos significativo (LSB), se emplea para informar a cada microcontrolador auxiliar del subsistema EMG si debe comenzar a vibrar su motor o dejar de hacerlo. Se emplean 2 bits por cada motor para que, aunque ahora solo se tenga la información de si debe vibrar o no, en un futuro se pueda incluir hasta 3 intensidades de vibración empleando PWM en función de la presión detectada por cada sensor de tacto, sin tener que cambiar la trama I²C. Este LSB no lo utiliza el microcontrolador principal, sino que simplemente enviará la parte necesaria a cada microcontrolador auxiliar.

- Comunicación del microcontrolador a la Raspberry (Figura 5.6): Esta trama solo se da en el caso de que previamente le hayamos mandado al microcontrolador la trama anterior con el byte MSB con valor 0x2. La trama también está formada por 2 bytes, pero con un significado distinto. El byte MSB ahora indica la clase que ha identificado la red neuronal de las 4 posibles (Figura 4.6), mientras que el el LSB indica la probabilidad porcentual con la que está segura la red neuronal de que sea esa clase.

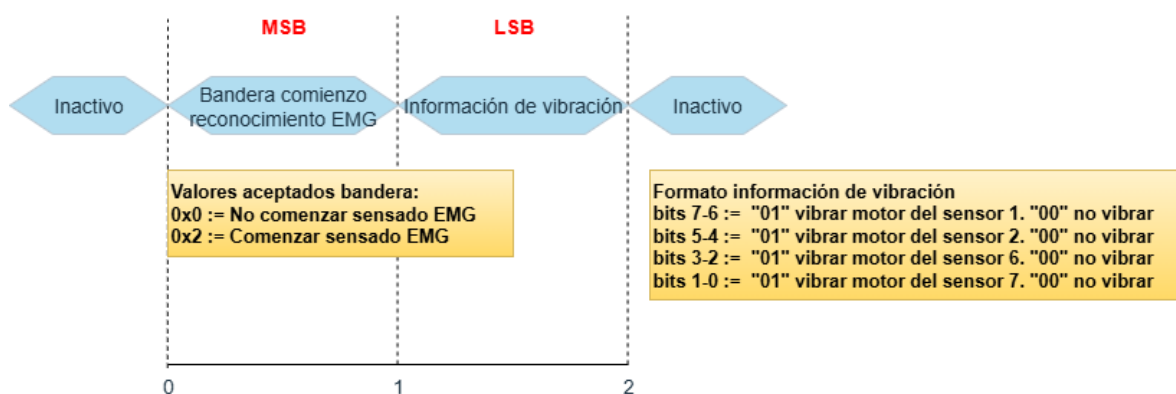


Figura 5.5: Formato de comunicación I²C de la Raspberry al microcontrolador principal de sensado EMG

Por otro lado, tenemos la comunicación entre el microcontrolador principal de

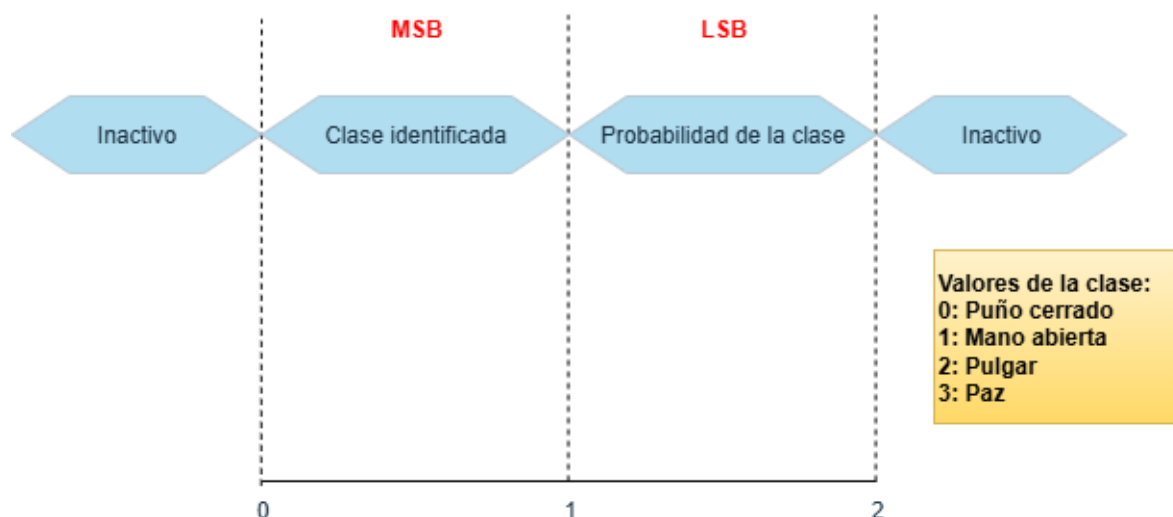


Figura 5.6: Formato de comunicación I²C del microcontrolador principal de sensor EMG a la Raspberry

este subsistema y sus microcontroladores auxiliares. El formato es el mismo que ya vimos en la Figura 3.14, con la diferencia de que el primer mensaje ahora estaría formado por 2 bytes: el primero sigue siendo igual y es la bandera de inicio o no del sensor EMG, pero se añade un segundo que es la información de si debe vibrar o no su motor. Esta información está contenida en los 2 bits LSB de este segundo byte. El microcontrolador principal procesó su byte LSB que le llegó de la Raspberry para formar un nuevo mensaje para cada microcontrolador auxiliar, poniendo la información de vibración en los 2 bits LSB.

Con respecto a la línea de interrupción, este subsistema no cuenta con una, por ser un diseño realizado por otros compañeros.

5.2. Diseño de la Interfaz de Usuario

Como interfaz de usuario de nuestro sistema, se propone el uso de 3 botones, que son los que permiten ir cambiando entre los 3 modos que ya comentamos en la sección 4.3. Así, cada vez que se pulse un botón, se entrará en ese modo, lo cual se indicará mediante el altavoz y los LEDs RGB, y se realizará una petición al subsistema pertinente para realizar el reconocimiento asociado a ese modo y mover los dedos en consecuencia.

Por otro lado, se ha incluido una pantalla LCD que, aunque ahora no esté en funcionamiento debido a que no constaba en el alcance del proyecto, permitirá en un futuro visualizar todo tipo de datos acerca de la prótesis, como la posición de cada dedo, el modo en el que nos encontramos, etcétera, o podrá usarse de manera lúdica teniendo una imagen en movimiento, por ejemplo.

Podemos ver esta interfaz con los botones y la pantalla en la Figura 5.7.



Figura 5.7: Interfaz disponible para el usuario de la prótesis. Contamos con 3 botones para cambiar de modo y una pantalla LCD que podrá ser usada en el futuro para visualización de datos o de manera lúdica.

La explicación detallada de cómo usar la prótesis a través de esta interfaz aparece en el Manual de usuario, en el Anexo [B.3](#).

5.3. Innovación y mejora

Aunque para este trabajo se han tomado como base algunos trabajos previos, no solo se ha hecho una simple readaptación de los mismos, sino que se han mejorado y reinterpretado, según las necesidades de la nueva arquitectura propuesta para poder llevar a cabo una integración completa y funcional en el sistema final. Muchos aspectos han tenido que ser rehechos y, aún así, se han añadido elementos innovadores y que serán explicados en los siguientes apartados, como:

- Agarre adaptativo, para conseguir agarres que se ajusten al objeto real que se pretende sujetar.
- Programación centralizada e inalámbrica, con OpenOCD.
- Depuración centralizada de cada microcontrolador, desde la Raspberry.
- Mejoras de consumo, funcionalidad y claridad en los códigos.
- Mejoras en la red neuronal de sensado EMG y adición a dicho subsistema de la realimentación del tacto.
- Rediseño y mejora en el sensado del tacto.
- Mejoras en el reconocimiento de objetos mediante OpenCV, con una funcionalidad completamente correcta, no como la que existía anteriormente.

5.3.1. Agarre adaptativo

Una de las innovaciones más importantes a la hora del funcionamiento de la prótesis ha sido la inclusión de la funcionalidad de agarre adaptativo. El movimiento de la prótesis anterior se basaba en llevar cada dedo a una posición predefinida, de tal manera que no se paraba hasta que se llegase ahí. Esto hacía que si se quería programar

la prótesis para agarrar un objeto cilíndrico como una taza, los dedos se movían a una posición predeterminada para coger esa taza de ese tamaño, pero si se quería agarrar un objeto cilíndrico de menor radio como una botella, la posición previa no es válida, mientras que para el caso de una botella de radio mayor, se apretaba sin control.

Con la colaboración del subsistema de sensado de tacto y el subsistema de control de motores a través del sistema central, se ha decidido incluir una funcionalidad que permita hacer agarres del mismo tipo, pero que se adapten al objeto real que se desea agarrar. De este modo, cuando la Raspberry recibe información de que alguno de los sensores capacitivos está sobrepasando una cota predefinida en la propia Raspberry, esta decide parar el dedo asociado a ese sensor, ya que esto significa que se está comenzando a ejercer cierta presión con ese dedo.

Gracias a esta nueva funcionalidad, conseguimos un agarre controlado, que también aporta seguridad al sistema, especialmente al estar enfocado a niños.

5.3.2. Programación con OpenOCD

Uno de los retos más importantes del sistema a nivel de diseño electrónico y mecánico era el conseguir un tamaño muy reducido, pero que sea capaz de albergar todos los elementos electrónicos, como la PCB final, y mecánicos, como los motores con su mecanismo para cada dedo. Esto es así debido a que la prótesis debe tener el tamaño de la mano de un niño de entre 8 y 10 años.

Es por ello que algo importante de abordar era la programación de los 3 microcontroladores que iban a formar nuestra PCB. Si se empleaba una programación convencional, habría que incluir el circuito integrado que nos permitía programar cada microcontrolador, además de conexiones externas donde conectar nuestro ordenador para introducir el programa. Esto suponía un incremento del tamaño de la PCB.

De esta manera, nos preguntamos si se podría programar cada microcontrolador desde la propia Raspberry, ya que era el elemento central que tenía acceso a los demás, lo que nos ahorraría todo el espacio anterior y permitiría realizar la programación desde un único elemento centralizado y al que podíamos tener acceso inalámbrico mediante SSH, sin necesidad de una conexión extra. Con esto, descubrimos la existencia de OpenOCD, una herramienta de software libre que permite programar microcontroladores mediante interfaces JTAG o SWD. Así, empleamos 2 pines GPIO de la Raspberry por cada microcontrolador, uno para datos mediante SWD y otro para el *reset*. Como señal de reloj, pudimos compartir la misma entre los 3 para así ahorrarnos 2 pines de la Raspberry. De este modo, para realizar la programación solo necesitábamos compilar el programa que queríamos subir para obtener el fichero binario, pasarlo a la Raspberry y desde ahí subirlo al microcontrolador deseado.

La instalación de esta herramienta y su uso se detalla en el Manual del desarrollador, en el Anexo [A.2.2](#).

5.3.3. Depuración de cada subsistema mediante UART virtual en Raspberry

Para realizar pruebas una vez que tuviéramos nuestra PCB final, se nos hacía necesario tener acceso a la UART empleada en cada microcontrolador para poder depurar. Sin embargo, teníamos el mismo problema que con la programación: si queríamos hacer una depuración convencional mediante un circuito integrado al que conectar nuestro ordenador, necesitaríamos más espacio en la PCB.

Al igual que con la programación, decidimos emplear la Raspberry, conectando los pines TX de cada microcontrolador a pines RX de la Raspberry. No obstante, buscando información y haciendo pruebas previas, usar la UART por defecto (GPIO 14 y 15) con otras UART nos daba problemas. Por ello, decidimos emplear UART virtual (por *software*), mediante el servicio *pigpio*. Empleaban los pines RX que trae la propia Raspberry, pero empleando este servicio. Con ello podríamos leer en la Raspberry lo que se enviaba a la UART de cada microcontrolador.

La instalación y uso de este sistema aparece en el Manual del desarrollador, en el Anexo [A.2.2](#).

5.3.4. Mejora en el código de reconocimiento de voz y modo de bajo consumo

En este código hay dos aspectos en los que podemos realizar mejoras: manejar los periféricos en alto nivel e introducir un modo de bajo consumo.

En primer lugar, el programa empleado manejaba todos los periféricos a nivel de registro. Esto no es incorrecto, pero hace que el código sea mucho menos legible y, lo más importante, no era portable a ningún otro microcontrolador, aunque se siguiera empleando STM, ya que los registros cambian de unos a otros. Esto hacía que si en un futuro se decidiera cambiar el microcontrolador empleado por uno más potente, aunque también fuese de STM, se tendría que readaptar de nuevo el código. Por ello, se decidió reescribir el código para emplear las funciones de alto nivel de la biblioteca HAL de STM para ganar en claridad y portabilidad. Así, con un nuevo microcontrolador STM, el código se podría copiar y pegar, simplemente cambiando los pines usados.

Por otra parte, como ya se ha visto cuando hemos explicado la funcionalidad, el subsistema de reconocimiento de voz solo se emplea cuando se pulsa el botón para reconocer la voz. Mientras no se pulsa este botón, tenemos un microcontrolador bastante potente, a su máxima frecuencia (170 MHz), consumiendo energía innecesariamente. Debido a esto, se decidió que el microcontrolador entrase en modo de bajo consumo y solo se despertase cuando el usuario pulse el botón de su modo. De todos los modos de bajo consumo que ofrece STM, empleamos el *shutdown mode* (Figura 5.8) por ser el de menor consumo. En él, prácticamente todos los periféricos están apagados y pasa a consumir 87 nA a 25 °C y con un VDD de 3.6 V. Para poner esto en perspectiva, el anterior modo de más bajo consumo, el *Standby mode*, en las mismas condiciones consume 180 nA. El *shutdown mode* equivale a un *reset* del programa, porque cuando se despierta, vuelve a ejecutar todo desde el inicio, pero el tiempo que requiere es

despreciable y vale la pena por el ahorro de energía.

Para despertar a este microcontrolador de su modo, empleamos un pin de *Wake Up* conectado a un GPIO de la Raspberry, de tal manera que cada vez que pulsemos el botón del modo voz, la Raspberry pondrá ese pin en alto, de modo que ese flanco de subida sacará al microcontrolador de su modo de bajo consumo para que se ponga a ejecutar su programa. Cuando acabe y haya mandado los datos necesarios a la Raspberry, entrará de nuevo en este letargo hasta que se le vuelva a despertar.

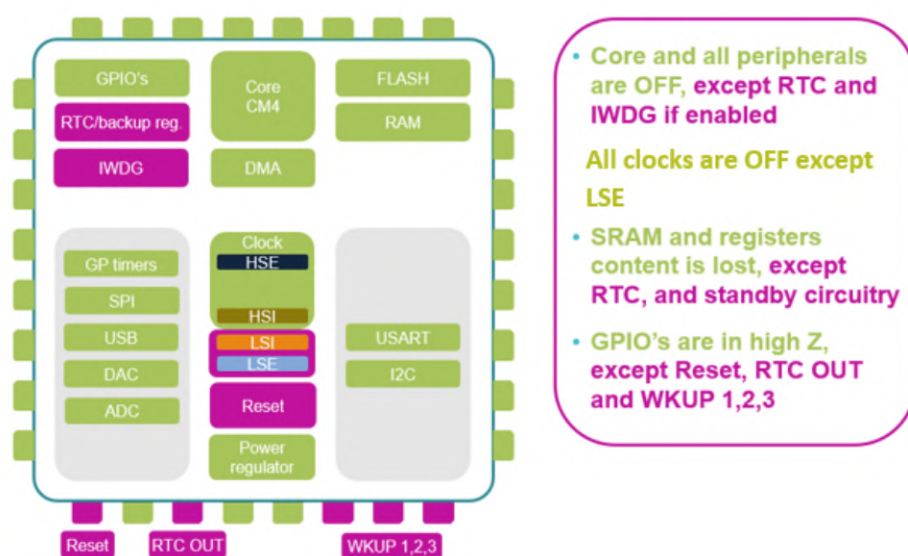


Figura 5.8: Modo de bajo consumo *shutdown mode* de STM

5.3.5. Mejora en la red neuronal del subsistema de sensado EMG y adición de motores vibradores para realimentación del tacto

En este caso, como los compañeros que hicieron este trabajo no pudieron dedicarle todo el tiempo que quisieron al apartado de su red neuronal, se llevó a cabo una mejora tanto en su estructura como en su entrenamiento, aunque ya vimos que la matriz de confusión del sistema era bastante buena (Figura 3.16), pero tenía un problema y es que siempre predecía una de las clases con el 100 % de probabilidad y con 0 % en el resto, lo que podría indicar un *overfitting*.

Primero, en cuanto a la estructura, antes se tenía como salida 5 clases, cuando realmente solo se habían entrenado 4. Esto hacía que no se predijera bien y que las probabilidades no fueran las que deberían porque se repartían con una quinta clase. Por ello, se cambió esto a 4 clases. Antes también se empleaban 2 capas ocultas de 16 neuronas cada una, pero tras hacer pruebas, esto se decidió cambiar a una capa con 32 y la siguiente con la mitad, 16, ya que es lo que se suele hacer en este tipo de red neuronal y lo que mejor resultado nos dio en el entrenamiento.

Además, se han empleado medidas para evitar el *overfitting*. Una de ellas ha sido utilizar regularización L2 y un *dropout* del 50 % de las neuronas en las dos capas

ocultas. Esta regularización penaliza los pesos grandes y el *dropout* “apaga” el 50 % de las neuronas. Así, evitamos que memorice demasiado para obtener una mejor generalización. La otra medida ha sido emplear *early stopping*, que supervisa la pérdida de validación (*val_loss*) en cada *epoch* para que, si no mejora durante *x epochs*, detenga el entrenamiento prematuramente.

En cuanto al *dataset* y su manejo también ha habido cambios. En primer lugar, se ha realizado un *resample* del *dataset* empleando *SMOTE* para equilibrar las muestras de todas las clases y así añadir muestras artificiales en las clases que cuenten con menos. Gracias a esto, nuestro *dataset* también se hace más grande, lo cual nos viene muy bien teniendo en cuenta que lo hemos tenido que generar nosotros con mediciones de nuestro antebrazo y no era sencillo. Aún así, también hemos ampliado su tamaño creando un dataset nuevo con muchas más muestras, llegando a unas 600 frente a las 200 que emplearon los compañeros en su proyecto para el entrenamiento. Otro aspecto importante ha sido el escalado de los datos utilizando *standard scaler*, para convertirlos en una distribución con media 0 y desviación estándar 1. Gracias a esto, conseguimos reducir el rango de los datos, obteniendo un conjunto más homogéneo, con lo que ganaremos precisión en el entrenamiento y evitaremos que una característica domine el aprendizaje solo por su escala (valor *outliner*). Al hacer esto, también ha sido necesario aplicar este escalado al programa del microcontrolador principal, para que cuando obtenga las características de cada auxiliar, las preprocese de este modo para que la red neuronal reciba una entrada que tenga la misma forma que en el entrenamiento. Finalmente, de todo el conjunto de datos, el 80 % se ha usado para entrenamiento, el 20 % para *tests* y de ese 80 % anterior, se ha usado otro 20 % para validación, pero este 20 % ha estado aislado para que no afecte en ningún momento al entrenamiento y solo se emplee para validación.

Gracias a estos cambios, hemos obtenido un *accuracy* en el entrenamiento de 0,96, bastante superior al que obtuvieron los compañeros, que oscilaba entre 0,7 y 0,85. Además, ahora en las predicciones no obteníamos como salida una clase al 100 % de probabilidad y el resto con 0, sino que obteníamos resultados más plausibles. La gráfica de precisión de entrenamiento y validación obtenida se puede ver en la Figura 5.9, la de pérdida de entrenamiento y validación en la Figura 5.10 y la matriz de confusión aparece en la Figura 5.11.

Por otro lado, se han soldado motores vibradores de tipo moneda a cada una de las PCB auxiliares de este subsistema para que se pueda incluir la vibración cuando se detecte tacto en alguno de los sensores. Esto también ha traído consigo una modificación en los formatos de comunicación dentro de este subsistema y entre él y el sistema central.

5.3.6. Mejora y rediseño en el código de sensado de tacto

Como ya hablamos en la subsección 4.1.3, en el código que se encarga del sensado de tacto hicimos un rediseño completo que incluye varias mejoras. Entre ellas, la inclusión de la programación orientada a objetos mediante una estructura de clases, que aporta claridad a su funcionamiento.

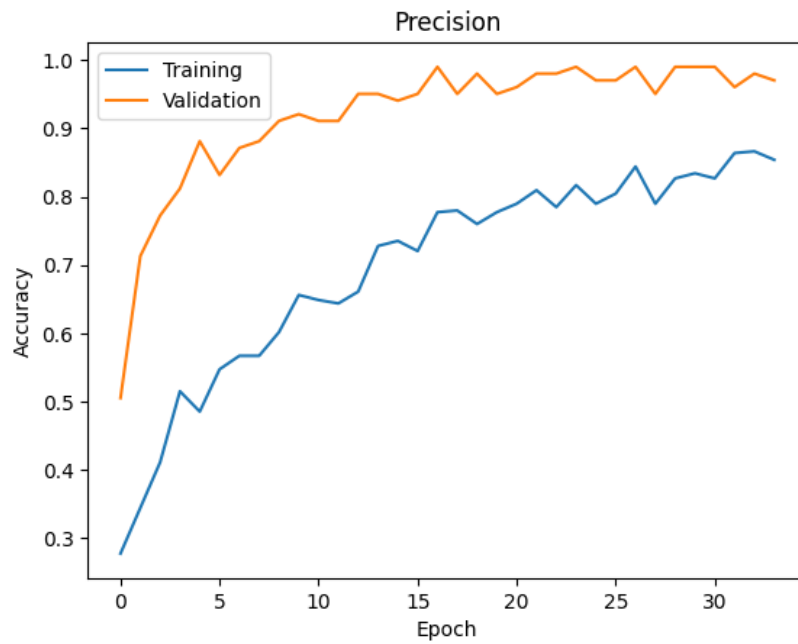


Figura 5.9: Gráfica de precisión del entrenamiento y validación de la red neuronal del subsistema de sensado EMG

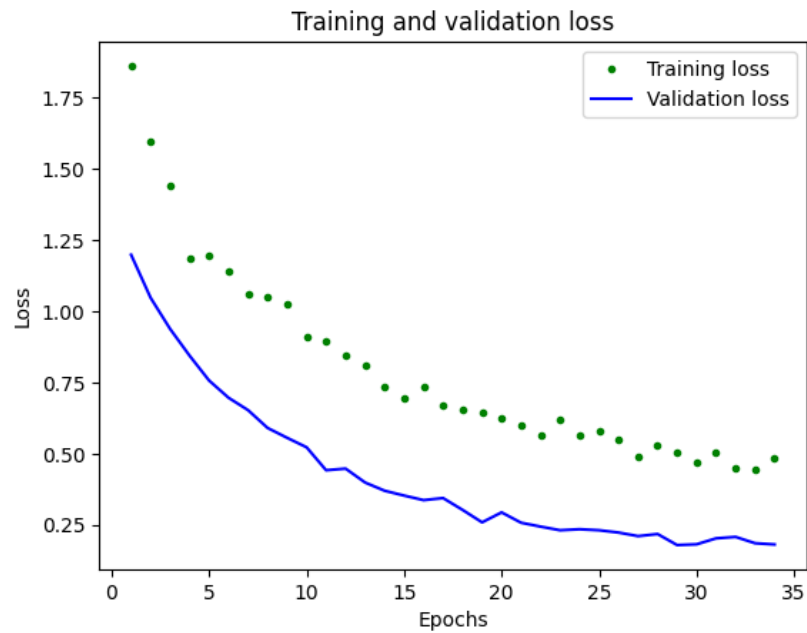


Figura 5.10: Gráfica de pérdida del entrenamiento y validación de la red neuronal del subsistema de sensado EMG

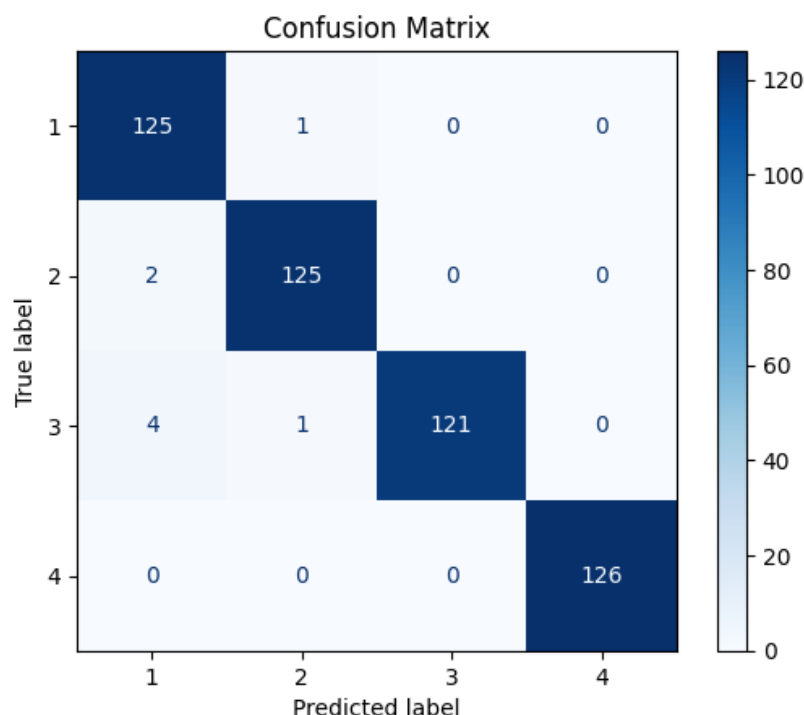


Figura 5.11: Matriz de confusión del entrenamiento del subsistema de sensado EMG

Por otro lado, una mejora importante ha sido la interpretación del funcionamiento del circuito integrado AD7147, que devuelve los valores leídos de cada sensor. El compañero que realizó el trabajo previo, simplemente leía los valores que devolvía y usaba cotas predefinidas para que, cuando se superasen, vibrasen los motores. Pero esto tenía un problema y era que los sensores no dan siempre la misma medida, sino que varía mucho dependiendo de las condiciones ambientales, como temperatura y humedad. Además, investigando más a fondo la hoja de datos del circuito integrado y haciendo pruebas, llegamos a la conclusión de que los valores que devolvía el integrado, por sí solos, no tenían sentido, sino que lo que tenía significado era la diferencia entre el valor leído actual y el valor del sensor en reposo (base). Eso otorgaba la variación de capacidad que había tenido el sensor. Además, esto permitía mejorar el funcionamiento al establecer como valor base el primer valor leído de cada sensor y así la cota no fuese de un valor absoluto que podría cambiar en un futuro y dejar de funcionar, sino una diferencia que, independientemente de las condiciones ambientales, siempre funcionase y tuviese el mismo significado. Las pruebas con las que llegamos a esta conclusión aparecen en la subsección [8.2.2](#).

5.3.7. Mejora en el uso de OpenCV para reconocimiento de objetos

En este caso, la mejora no ha sido tan sustancial como en los casos anteriores, pero sí que ha tenido impacto en el rendimiento del reconocimiento de objetos con OpenCV.

En primer lugar, el código anterior, cuando hacía un reconocimiento de la imagen que había tomado la cámara para obtener los objetos reconocidos, no devolvía el objeto reconocido con más calidad, sino el primero que había obtenido. Esto era un problema grande si había varios objetos en pantalla. La mejora que se ha realizado ha consistido en devolver el mejor objeto de todos los reconocidos.

En segundo lugar, otro problema era que cuando se hacía el reconocimiento de un objeto, se reconocía correctamente, pero al mover la cámara hacia otro y comenzar de nuevo el proceso, seguía devolviendo el objeto anterior durante varias iteraciones, afectando incluso al funcionamiento correcto debido a que había que esperar unos 20 segundos si se movía la cámara antes de que se pudiera reconocer el nuevo objeto. Esto era debido a que en el *buffer* que emplea OpenCV todavía había almacenadas capturas de la imagen anterior, así que cada vez que se quería reconocer, se hacían nuevas capturas pero se seguían procesando las que estaban antes en la cola, que eran del objeto anterior. Para evitar esto, se han descartado los 5 primeros frames del *buffer*, de tal manera que ahora, cada vez que se hace el reconocimiento, se obtiene el objeto actual, aunque se haya movido la cámara.

6. Implementación del Sistema

Este capítulo trata sobre todos los aspectos relacionados con la implementación del sistema en código, haciendo uso de un determinado entorno tecnológico. Primero se verá el entorno usado para la construcción y luego se hablará del código fuente, tanto de su organización en directorios como de la implementación de cada subsistema, explicando aquellos aspectos que se consideran más interesantes de cada uno.

6.1. Entorno de Construcción

Como ya hemos hablado anteriormente, nuestro proyecto se divide en la programación de microcontroladores empleando C/C++ y la programación en la Raspberry usando Python. Teniendo estas 2 variantes, explicamos a continuación el entorno de cada uno:

- Microcontroladores: Al emplear microcontroladores STM32, utilizamos como entorno de desarrollo STM32CubeIDE, el IDE que proporciona la misma empresa y que tiene multitud de utilidades para la programación y depuración de sus microcontroladores en lenguaje C y C++, como generación de código base según los periféricos empleados y su configuración, o *middlewares*, como STM32Cube.AI, para la generación de código para implementar en C un modelo de red neuronal preentrenado.
- Raspberry: Para los códigos en lenguaje Python de la Raspberry se ha usado *Visual Studio Code* como editor de código, junto con las bibliotecas necesarias, como *RPi.GPIO*, para el manejo de los pines GPIO de la misma.

6.2. Código Fuente

En esta sección se hablará de la organización del código fuente, describiendo la utilidad de los diferentes ficheros y su distribución en paquetes o directorios. Asimismo, se incluirá algún extracto significativo de código fuente que sea de interés para ilustrar algún algoritmo o funcionalidad específica de algunos subsistemas.

6.2.1. Organización del código fuente

Todo el código fuente se encuentra en el directorio *DigitalAppendixV3*. Este código es accesible desde el siguiente enlace de *GitHub* <https://github.com/Praxngg/TFG-ManoRobotica>. En él encontraremos, a su vez, cuatro directorios: uno llamado *Microcontrollers*, con los códigos de cada microcontrolador, otro llamado *Raspberry*, con los códigos de la Raspberry, el tercero se llama *EMG_Training*, con lo necesario para

realizar el entrenamiento de la red neuronal del subsistema EMG y el último llamado *Voice_Training*, con lo necesario para el entrenamiento del subsistema de reconocimiento de voz.

Los subdirectorios del directorio *Microcontrollers* son proyectos de STM listos para importar en su IDE. Los ficheros de código fuente y de cabecera se encuentran en *Core/Src* y *Core/Inc*, respectivamente. Estos proyectos son:

- *BlinkG4*: Proyecto de prueba para que el microcontrolador encargado del reconocimiento de voz haga parpadeos en su LED.
- *BlinkL4*: Proyecto de prueba para que el microcontrolador encargado del control de motores haga parpadeos en su LED.
- *BlinkL4Cap*: Proyecto de prueba para que el microcontrolador encargado del sensado de tacto haga parpadeos en su LED.
- *CapacitiveControlMicro*: Proyecto con el que se debe programar el microcontrolador encargado del sensado de tacto para el funcionamiento correcto de su subsistema.
- *L431KCU6_WristbandAmp*: Proyecto con el que se debe programar cada microcontrolador auxiliar del subsistema de sensado EMG para el proceso de obtener el *dataset* con el que posteriormente se puede entrenar la red neuronal. **Importante:** Todos los microcontroladores auxiliares se programan usando este mismo proyecto. Solo debe cambiar en *Core/Src/main.c* en la función *MX_I2C1_Init()*, en *hi2c1.Init.OwnAddress1*, la dirección I²C hexadecimal del microcontrolador (teniendo en cuenta que el 1 tiene la dirección 0x01, el 2 la dirección 0x02, etc.). Se puede ignorar el desplazamiento lógico a la izquierda de 1 bit, ya que es algo interno de STM. También se debería cambiar en *Core/Src/FeatureCalculation.c*, en la función *remove_offset()*, descomentar o comentar la línea indicada en los comentarios depende del microcontrolador que estemos programando.
- *L431KCU6_WristbandAmp_Deploy*: Proyecto con el que se debe programar cada microcontrolador auxiliar del subsistema de sensado EMG su correcto funcionamiento. **Importante:** Al igual que en el proyecto anterior, todos los microcontroladores auxiliares se programan usando este mismo proyecto, con las mismas consideraciones explicadas en el punto anterior.
- *L496RGT6_EMG_NeuralNetwork_Deploy*: Proyecto con el que se debe programar el microcontrolador principal del subsistema de sensado EMG para su correcto funcionamiento.
- *L496RGT6_PrincipalMCU*: Proyecto con el que se debe programar el microcontrolador principal del subsistema de sensado EMG para obtener el *dataset* a usar en el posterior entrenamiento de la red neuronal.
- *MotorControlMicro*: Proyecto con el que se debe programar el microcontrolador encargado del control de motores para el funcionamiento correcto de su subsistema.
- *VoiceControlMicro*: Proyecto con el que se debe programar el microcontrolador

encargado del reconocimiento de voz para el funcionamiento correcto de su subsistema.

En el caso del directorio *Raspberry*, encontramos los ficheros fuente de Python que son todos usados por el fichero *controlV3_1.py*, que será el que debamos ejecutar en la Raspberry. También encontramos el fichero *config.ini*, que tiene parámetros de configuración editables, como la dirección I²C de cada esclavo, cotas del sensado de tacto, etc.

En el directorio *EMG_Training* tenemos lo siguiente:

- *EMG_DataAcquisitionFlow.json*: Flujo para importar en el CubeMonitor para realizar todo el proceso de adquisición de muestras que forman el *dataset* para el entrenamiento.
- *EMG_NN_Feature_Training_mod.ipynb*: Cuaderno de *jupyter* con el *script* de python que permite realizar el entrenamiento de la red neuronal y obtener el fichero *.tflite* para su implementación en el microcontrolador.

En el directorio *Voice_Training* encontramos los siguientes directorios:

- *KWS CNN Trained Net*: En él está el fichero *kws_stm32_cnn.onnx* que nos permite implementar en el microcontrolador el modelo de red neuronal de reconocimiento de voz entrenado.
- *Script de entrenamiento en MATLAB*: Como su nombre indica, tenemos el *script* de matlab usado para entrenar la red neuronal y obtener el fichero *.onnx* anterior.

6.2.2. Implementación del subsistema de control de motores

Lo primero que podemos destacar de la implementación es el fichero *defines.h*. En él encontramos directivas importantes, como *TEST_ON*, que si la comentamos eliminaremos todas las partes del código relacionadas con el *debugging*, entre ellas la creación del objeto de la clase *testHandControl*, los delays y las impresiones a través de la UART. Otro apartado importante de este fichero son las directivas *FINGER_MIN* y *FINGER_MAX*, que tienen las posiciones mínima y máxima de cada dedo y que, a su vez, deben coincidir con las posiciones mínima y máxima declaradas en el apartado *minMaxValues* del fichero *config.ini* del directorio *Raspberry* que comentamos anteriormente.

En cuanto a la comunicación con el sistema central, se decidió emplear I²C por interrupciones. Esto es así porque el funcionamiento principal de este subsistema era comunicarse con el driver de motores para mover algún dedo e ir leyendo constantemente los potenciómetros para ello. Si en este proceso llega un nuevo mensaje para mover la mano de otra manera, el microcontrolador interrumpirá lo que está haciendo para leer este nuevo mensaje y mover los motores en consecuencia con el menor tiempo de reacción posible. Por otro lado, si usásemos *polling* en vez de interrupciones, no podríamos estar leyendo los potenciómetros en bucle mientras se mueven los motores. Además, las interrupciones I²C en STM traen consigo una interrupción extra que es cuando se ha completado la lectura o envío de datos. Esta función la usamos para

activar la bandera de que hay un nuevo comando disponible para decodificar, porque así ya estamos seguros de que la información se ha recibido completamente.

Para manejar el objeto de la clase *HandControl* que controla el movimiento de la mano, se ha utilizado un registro de estado en lugar de variables individuales como banderas, y que podemos encontrar en su fichero de cabecera. Este registro es una variable de 8 bits con el siguiente formato:

- Bit 0: bandera de nuevo comando disponible.
- Bit 1: bandera de pin de interrupción activo.
- Bit 2: bandera que indica que, al menos, un dedo se está moviendo.

Lo mismo ocurre con la clase *Finger*, pero el formato es distinto:

- Bit 0: bandera que indica si una nueva posición límite está disponible.
- Bit 1: bandera que indica si se alcanzó el límite.
- Bit 2: bandera que indica si se está abriendo el dedo.
- Bit 3: bandera que indica si se está cerrando el dedo.
- Bit 4: bandera que indica si se ha cumplido una condición de interrupción en la última comprobación.

En esta clase también definimos en el fichero de cabecera la velocidad de los dedos, en el rango $[-255, 255]$ y la máxima diferencia de valor entre nuestra posición actual y la posición que se nos envía desde el maestro, para evitar aceptarla si es muy parecida.

Lo último destacable de esta implementación es la función *checkLimitReached()*, de la clase *Finger*, que se encarga de parar el dedo si se ha llegado al límite. Debido a que los motores que tenemos actualmente son demasiado rápidos, el dedo no se para cuando se llega al límite establecido, sino cuando le queda un valor de 500 para llegar, porque los motores actuales son muy veloces y se mueven más rápido de lo que somos capaces de leer los potenciómetros y, si no se hiciera este ajuste, se pasarían de su posición límite.

6.2.3. Implementación del subsistema de sensado de tacto

Al igual que en el caso anterior, también tenemos en el fichero *defines.h* una directiva para habilitar o deshabilitar en el código la depuración.

Algo que destacar en la implementación es la configuración de cada *stage* empleado del AD7147 que se hace usando el método *ConfigStage()*. Se emplea la configuración de conectar cada *stage* a una entrada positiva, para que la lectura cuando se presiona el sensor capacitivo dé un valor mayor. Por otro lado, tenemos los *offset* iniciales, que se emplean para aumentar o disminuir el valor leído en reposo. Lo ideal es que el valor en reposo estuviera en torno a 32700, que es la mitad del rango $[0, 65535]$, para evitar que si es un valor muy grande o muy pequeño, al presionar se salga del rango, pero como el

valor en reposo va cambiando con cada uso y hemos probado en distintas condiciones y nunca se sale el valor de rango, hemos dejado estos *offset* con valor 0.

En cuanto al modo de funcionamiento del AD7147, podía funcionar en *full power* o en *low power*. En un principio se pensó configurarlo como *low power*, que lo que hace es que cuando ningún sensor está siendo tocado, permanece en un estado de bajo consumo. Esto es muy práctico, pero tiene una desventaja y es que este estado de bajo consumo se consigue reduciendo la frecuencia de conversión de los sensores, siendo el menor *delay* entre ellas de 200 ms. En una aplicación en la que el tacto es crucial para la seguridad del sistema, por el funcionamiento de agarre adaptativo, no podíamos permitirnos una espera tan alta entre conversiones. Por ello, se decidió emplear el modo *full power*, que permite una conversión en el peor de los casos de unos 37 ms. Aún así, para comprobar si el ahorro de energía era significativo, realizamos pruebas y el consumo de este subsistema (microcontrolador y AD7147) era de 6,65 mA constantemente en modo *full power*, mientras que en el modo de bajo consumo, cuando no se tocaba ningún sensor, el consumo era de entre 5,79 y 6,12 mA, por lo que el ahorro era de unos 0,86 mA en el mejor caso. Teniendo en cuenta que nuestro sistema en reposo consume unos 300 mA, el ahorro no es significativo comparado con la importancia del correcto funcionamiento del agarre adaptativo.

Por otro lado, en el fichero de cabecera de la clase *CapacitiveControl* encontramos otros aspectos interesantes para comentar. En primer lugar, tenemos definida la cota de cada sensor. El significado de esta cota es distinto al de la cota que tenemos en el fichero *config.ini* en la Raspberry. Estas cotas indican cuál es el valor límite de diferencia que, superado, significa que ha ocurrido una presión suficiente como para considerar que ha habido un inicio de tacto y, por tanto, debe avisar al maestro a través de la interrupción. También tenemos aquí un registro de estado que indica si la interrupción al maestro ha sido activada y los restantes bits indican si la cota de algún sensor ha sido alcanzada.

6.2.4. Implementación del subsistema de reconocimiento de voz

Una parte importante de esta implementación es el empleo del modo de bajo consumo que ya se comentó. La peculiaridad de este modo es que prácticamente apaga el microcontrolador y, tras despertarlo, ejecuta el código desde el principio, como si se le hubiera aplicado un *reset*. Para despertarlo del letargo, lo que empleamos es un pin de *wake up* que se activa en alto cuando se pulsa el botón que activa su modo. Un problema que se podía derivar de esto era no poder diferenciar si se había despertado por el pin de *wake up* o por el primer inicio del sistema, ya que en este último caso, se debía dormir de nuevo. Sin embargo, simplemente leyendo el pin somos capaces de hacer esta diferenciación, ya que en el primer inicio del sistema, el pin estaría en nivel bajo.

Un cambio importante respecto a las implementaciones anteriores es el modo de funcionamiento del I²C. Antes, empleábamos I²C por interrupciones, mientras que en esta implementación lo usamos por *polling*. Esto es así porque la función de este microcontrolador es que, una vez que se pulse el botón, hacer el reconocimiento, enviar el resultado y dormirse, y no debe hacer otra cosa mientras espera a que el maestro

le lea. Por ello, es suficiente con que se quede bloqueado, esperando esto tras avisar previamente al maestro de que ya tiene sus datos listos.

Por último, como ya se comentó en la subsección 5.3.4, es destacable el cambio en el uso y configuración de los periféricos utilizados, como el DMA, el I²C o el I2S. En el código actual se han traducido todas las funciones previas que los usaban o configuraban a funciones de alto nivel, empleando la biblioteca HAL de STM en lugar de los registros específicos de esos periféricos.

6.2.5. Implementación del subsistema de reconocimiento de objetos

La parte más importante en cuanto a implementación de este subsistema es la función *update_status()* (Figura 6.1), junto con *getObjects()* (Figura 6.2). Estas funciones son las que emplean la biblioteca de OpenCV para capturar y procesar imágenes. Como comentamos en su diagrama de flujo, las ejecutamos en un hilo separado, porque así se hacen muchas más detecciones en los 2 segundos que estamos actualizando el estado. Las 2 funciones las explicamos a continuación:

- *update_status()*: Primero, realiza varias capturas con la cámara para desechar los primeros frames y así tener el *buffer* de imágenes actualizado con las imágenes del lugar a donde está apuntando la cámara en ese momento. Si no ha habido ningún problema, la función *getObjects()* devuelve todos los objetos que ha detectado en la imagen y la probabilidad de cada uno de ellos. De todos estos objetos, nos quedaremos con el de mayor probabilidad y con ese actualizaremos el *status* de nuestro subsistema.
- *getObjects()*: En este caso, con la captura que nos ha pasado la función *update_status()*, haremos la detección de objetos y añadiremos a *objectInfo*, cada objeto con su probabilidad, en el caso de que estuviera en los objetos que aceptamos. Algo importante que se ha añadido es el *False* como valor del parámetro *draw* cuando llamamos a esta función desde *update_status()*, ya que esto servía para mostrar por pantalla las imágenes que estaba viendo la cámara y pintar rectángulos alrededor de los objetos detectados, junto con su nombre y probabilidad. Como esto no nos es necesario, ha sido deshabilitado, con lo que hemos ganado en velocidad de procesamiento.

6.2.6. Implementación del subsistema de sensado EMG y retroalimentación de tacto

Recordemos que la implementación de este subsistema se divide en la implementación en los microcontroladores auxiliares que mueven los motores vibradores y recogen los datos EMG para calcular sus características y el microcontrolador principal que recibe las características y ejecuta la red neuronal. También, por supuesto, está la implementación en un *script* de Python del entrenamiento del modelo de red neuronal.

```

def update_status(self):
    while self.__running:
        # Read various frames without processing, to discard the first ones and get the image updated and not with the previous image
        for _ in range(5):
            self.cap.read()
        success, img = self.cap.read()
        if success:
            result, objectInfo = self.getObjects(img, 0.45, 0.2, False, objects=self.__accepted_objects)
            if objectInfo:
                objectInfo.sort(key=lambda x: x[2], reverse=True)
                self.__set_current_to_last_status()
                self.__status["object"] = objectInfo[0][1]
                self.__status["detection_quality"] = objectInfo[0][2] * 100
            else:
                self.__set_current_to_last_status()
                self.__status = {"object": None, "detection_quality": None}
            #cv2.imshow("Output", img)
            #cv2.waitKey(1)
        else:
            break

```

Figura 6.1: Función *update_status()* del subsistema de reconocimiento de objetos

```

def getObjects(self, img, thres, nms, draw=True, objects=[]):
    classIds, confs, bbox = self.net.detect(img, confThreshold=thres, nmsThreshold=nms)
    if len(objects) == 0:
        objects = self.classNames
    objectInfo = []
    if len(classIds) != 0:
        for classId, confidence, box in zip(classIds.flatten(), confs.flatten(), bbox):
            className = self.classNames[classId - 1]
            if className in objects:
                objectInfo.append([box, className, confidence])
            if draw:
                cv2.rectangle(img, box, color=(0, 255, 0), thickness=2)
                cv2.putText(img, className.upper(), (box[0] + 10, box[1] + 30),
                            cv2.FONT_HERSHEY_COMPLEX, 1, (0, 255, 0), 2)
                cv2.putText(img, str(round(confidence * 100, 2)), (box[0] + 200, box[1] + 30),
                            cv2.FONT_HERSHEY_COMPLEX, 1, (0, 255, 0), 2)
    return img, objectInfo

```

Figura 6.2: Función *getObjects()* del subsistema de reconocimiento de objetos

Empezaremos con el entrenamiento. Lo primero destacable es el *resample* que realizamos mediante *SMOTE*, para equilibrar el conjunto de datos y aumentar artificialmente el número de muestras (Figura 6.3). Por otro lado, tenemos el escalado realizado previo al entrenamiento y que podemos ver su implementación empleando *StandardScaler* en la Figura 6.4. Los valores aplicados para hacer este escalado los guardaremos en un archivo *.csv* para aplicarlo posteriormente en el código de nuestro microcontrolador principal y así realizar el mismo escalado con los nuevos datos que con los de entrenamiento.

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
feature_data_resampled, labels_resampled = smote.fit_resample(feature_data, labels)
```

Figura 6.3: *Resample* en el entrenamiento de la red neuronal para el subsistema EMG

```
# Scaling the sets
scaler = StandardScaler()
#train_scaled = scaler.fit_transform(training_features)
scaler.fit(train_set)
train_scaled = scaler.transform(train_set)
test_scaled = scaler.transform(test_set)
```

Figura 6.4: Escalado en el entrenamiento de la red neuronal para el subsistema EMG

Para el caso de los microcontroladores auxiliares, lo más interesante es la implementación de cada una de las características que se extraen de la señal EMG original y que podemos ver en las Figuras 6.5, 6.6, 6.7, 6.8 y 6.9. Estas características, extraídas por cada microcontrolador auxiliar, son las que luego se emplean como entrada de la red neuronal del microcontrolador central. Cada una de ellas se usa siguiendo los criterios del artículo desarrollado por *Azhiri et al* [41]. A continuación, se explica cada una:

- IEMG (*Integrated EMG*): Suma de los valores absolutos de la señal EMG en una ventana de longitud N . Se encarga de medir la energía global acumulada de la señal y es muy sensible a la contracción muscular así que, cuanto mayor actividad muscular, mayor será el valor.
- MAV (*Mean Absolute Value*): Promedio de los valores absolutos de la señal. Es una versión normalizada del IEMG y representa la amplitud media de la señal EMG, por lo que se usa mucho en el reconocimiento de gestos musculares.
- SSI (*Simple Square Integral*): Suma de los cuadrados de las muestras. Se relaciona con la potencia total de la señal.
- RMS (*Root Mean Square*): Es la raíz cuadrada de la media de los cuadrados de las muestras. Mide la amplitud efectiva de la señal. Esta característica es muy útil, porque combina la potencia de la señal con la escala original de amplitud y es menos sensible al ruido que MAV.
- VAR (*Variance*): Varianza de la señal. En EMG se suele usar para ver la variabilidad de la contracción muscular.

- MYOP (*Myopulse Percentage Rate*): Calcula la proporción de muestras que superan un umbral T. Gracias a esto, se mide la actividad muscular activa frente al reposo. Si el valor es alto, el músculo está más tiempo activo.
- WL (*Waveform Length*): Suma de las diferencias absolutas entre muestras consecutivas. Representa la complejidad o la longitud acumulada de la señal. Cuanta más variabilidad, mayor el WL.
- DAMV (*Difference Absolute Mean Value*): Promedio de los incrementos absolutos entre muestras consecutivas y es una versión normalizada de WL. Refleja el ritmo medio de cambio en la señal.
- M2 (*Second Moment of Differences*): Suma de los cuadrados de las diferencias entre muestras consecutivas. Es similar a una energía, pero aplicada a los cambios instantáneos de la señal y representa la potencia de la variación de la señal.
- DVARV (*Difference Variance Value*): Versión normalizada del M2. Mide la variabilidad del cambio, útil para distinguir entre movimientos suaves y movimientos bruscos.
- DASDV (*Difference Absolute Standard Deviation Value*): Desviación estándar de las diferencias entre muestras consecutivas. Refleja la variabilidad instantánea de la señal.
- MAX (*Maximum Value*): Valor máximo de la señal. Refleja la mayor activación muscular alcanzada en el intervalo.
- MIN (*Minimum Value*): Valor mínimo de la señal. Junto con MAX, es útil para calcular el rango pico a pico.
- WAMP (*Willison Amplitude*): Cuenta cuántas veces la diferencia entre dos muestras consecutivas supera un umbral T. Mide la frecuencia de cambios significativos en la señal.
- IASD (*Integrated Absolute of the Second Difference of First Derivative*): Calcula el cambio absoluto entre valores consecutivos de la primera derivada de la señal. Es decir, mide cómo varía la pendiente de la señal.
- IATD (*Integrated Absolute of the Third Difference*): Se basa en la segunda derivada. Mide la intensidad y frecuencia de los cambios de aceleración de la señal muscular.
- IEAV (*Integrated Exponential Absolute Value*): Calcula la suma de exponenciales del valor absoluto normalizado de cada muestra. Resalta fuertemente las contracciones intensas (valores grandes en la señal). No se usa, por dar valores infinitos en algunas ocasiones.
- IALV (*Integrated Absolute Logarithmic Value*): Calcula la suma del valor absoluto del logaritmo de cada muestra. Da más importancia a valores pequeños. No se usa, por dar valores infinitos en algunas ocasiones.
- IE (*Integrated Exponencial*): Suma exponenciales de las muestras normalizadas. Sirve para capturar la asimetría de la señal.

Algo también importante es la implementación de la vibración de cada motor. Cada vez que un microcontrolador auxiliar recibe un mensaje I²C, comprueba el LSB para ver la información de vibración. Si esta información es distinta a la que ya tenía, la aplicará, haciendo que el motor comience a vibrar o deje de hacerlo.

```
float IEMG(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        //int value = x[i];
        sum += fabs((float)x[i]);
    }
    return sum;
}

float MAV(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        sum += fabs((float)x[i]); // Absolutwert jedes Elements addieren
    }

    return sum / N; // Mittelwert berechnen
}

float SSI(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        sum += (float)x[i] * (float)x[i];
    }
    return sum;
}

float RMS(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        sum += (float)x[i] * (float)x[i];
    }
    float rms = sqrt(sum/N);
    return rms;
}

float VAR(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        sum += (float)x[i] * (float)x[i];
    }
    return sum/(N-1);
}
```

Figura 6.5: Características IEMG, MAV, SSI, RMS y VAR

En cuanto al microcontrolador principal, algo importante son las comunicaciones I²C, puesto que gran parte del código se basa en comunicaciones. Además, tenemos dos buses funcionando independientemente. El I²C1, que es maestro de los microcontroladores auxiliares, y el I²C2, que es esclavo de la Raspberry. En el caso del I²C2, su funcionamiento es mediante interrupciones.

```

float MYOP(const int16_t *x, int N, uint16_t T) {
    float sum = 0.0;
    for (int i = 0; i < N; i++) {
        if(abs(x[i]) > T){
            sum += 1;
        }
    }
    return sum/N;
}

float WL(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N - 1; i++) {
        //sum += abs((double)x[i+1] - (double)x[i]);
        sum += fabs((float)(x[i+1] - x[i]));
    }
    return sum;
}

float DAMV(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N - 1; i++) {
        //sum += abs((double)x[i+1] - (double)x[i]);
        sum += fabs((float)(x[i+1] - x[i]));
    }
    return sum/(N-1);
}

float M2(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N - 1; i++) {
        sum += ((float)x[i+1] - (float)x[i]) * ((float)x[i+1] - (float)x[i]);
    }
    return sum;
}

float DVARV(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N - 1; i++) {
        sum += ((float)x[i+1] - (float)x[i]) * ((float)x[i+1] - (float)x[i]);
    }
    return sum/(N-2);
}

```

Figura 6.6: Características MYOP, WL, DAMV, M2 y DVARV

```

float DASDV(const int16_t *x, int N) {
    float sum = 0.0;
    for (int i = 0; i < N - 1; i++) {
        sum += ((float)x[i+1] - (float)x[i]) * ((float)x[i+1] - (float)x[i]);
    }
    return sqrt(sum/(N-1));
}

float MAX(const int16_t *x, int N) {
    float max = (float)x[0];
    for (int i = 0; i < N; i++) {
        if(max < x[i]){
            max = (float)x[i];
        }
    }
    return max;
}

float MIN(const int16_t *x, int N) {
    float min = (float)x[0];
    for (int i = 0; i < N; i++) {
        if(min > x[i]){
            min = (float)x[i];
        }
    }
    return min;
}

float WAMP(const int16_t *x, int N, uint16_t T) {
    float sum = 0.0;
    for (int i = 0; i < N - 1; i++) {
        if(abs(x[i+1]- x[i]) > T){
            sum += 1;
        }
    }
    return sum;
}

```

Figura 6.7: Características DASDV, MAX, MIN y WAMP

```

float IASD(const int16_t *x, int N) {
    float sum = 0.0;

    if (N < 3) {
        // Nicht genügend Elemente für die Berechnung
        return 0;
    }
    for (int i = 0; i < N - 2; i++) {
        // Ableitung  $x'[i] = x[i+1] - x[i]$ 
        int x_prime_i = (int)x[i + 1] - (int)x[i];
        int x_prime_i1 = (int)x[i + 2] - (int)x[i + 1];

        // Summe der Differenzen der Ableitungen
        sum += (float)(abs(x_prime_i1 - x_prime_i));
    }

    return sum;
}

float IATD(const int16_t *x, int N) {
    if (N < 4) {
        // Nicht genügend Elemente für die Berechnung
        return 0;
    }

    float sum = 0.0;

    for (int i = 0; i < N - 3; i++) {
        // Erste Ableitung  $x'[n] = x[n+1] - x[n]$ 
        int x_prime_i = (int)x[i + 1] - (int)x[i];
        int x_prime_i1 = (int)x[i + 2] - (int)x[i + 1];

        // Zweite Ableitung  $x''[n] = x'[n+1] - x'[n]$ 
        int x_double_prime_i = x_prime_i1 - x_prime_i;
        int x_double_prime_i1 = ((int)x[i + 3] - (int)x[i + 2]) - x_prime_i1;

        // Betrag der Differenz der zweiten Ableitung
        sum += (float)(abs(x_double_prime_i1 - x_double_prime_i));
    }

    return sum;
}

```

Figura 6.8: Características IASD y IATD

```

float IEAV(const int16_t *x, int N) {
    float sum = 0.0;

    for (int i = 0; i < N; i++) {
        sum += exp((float)(abs(x[i]/N))); // e^(|x[n]/N|) berechnen und zur Summe addieren (N wurde wegen inf dazu gedichtet)
    }

    return sum;
}

float IALV(const int16_t *x, int N, uint16_t T) {
    float sum = 0.0;

    for (int i = 0; i < N; i++) {
        sum += fabs(log((float)x[i])); // |log(x[n])| berechnen und zur Summe addieren
    }

    return sum;
}

float IE(const int16_t *x, int N) {
    float sum = 0.0;

    for (int i = 0; i < N; i++) {
        sum += exp((float)x[i]/N); // e^(x[n]/N) berechnen und zur Summe addieren (N wurde wegen inf dazu gedichtet)
    }

    return sum;
}

```

Figura 6.9: Características IEAV, IALV y IE

Por otro lado, con respecto a este código, cabe destacar el preprocesamiento realizado a las características recibidas antes de introducirlas en la red neuronal, para que la entrada sea la misma que la que tuvo el entrenamiento. Como se explicó anteriormente, obtuvimos durante el entrenamiento los valores de media y desviación típica en un archivo *.csv* y los pasamos al fichero *preprocessing_data.h*, representando cada uno con vectores. Gracias a ello, pudimos hacer el mismo escalado a cada uno de los valores recibidos de los distintos microcontroladores auxiliares, como podemos ver en la Figura 6.10. Tras esto, se realiza la ejecución de la red neuronal y se obtiene la salida para enviarla a la Raspberry.

```

// Apply the preprocessing of the data (normalice as in the training)
for (uint32_t i = 0; i < AI_EMG_NN_IN_1_SIZE/4; i++)
{
    float x = receivedValues_Amp1[i];
    float s = std[i] == 0.0f ? 1.0f : std[i]; // avoid dividing by 0
    ((ai_float *)in_data)[i] = (x - mean[i]) / s;
}

```

Figura 6.10: Escalado realizado en el microcontrolador principal del subsistema EMG para los datos del amplificador 1

6.2.7. Implementación de los subsistemas de iluminación RGB y salida de audio

Ambos se cubren en esta subsección.

Para el caso del subsistema de iluminación RGB, se emplea la biblioteca *neopixel*,

de *Adafruit*. El método más importante es *set_color()*, que emplea las funciones *fill()* y *show()* de esta biblioteca para configurar y mostrar el color que le pasamos en código RGB.

Con respecto al subsistema de sonido, usamos la herramienta *mpg*, que nos permite ejecutar un fichero de audio en formato .mp3. Al ser una herramienta que ejecutamos desde la terminal, empleamos la biblioteca *subprocess* para hacerlo desde nuestro código Python.

6.2.8. Implementación del sistema central

Aunque tenemos muchos módulos en el sistema central, aquí hablaremos del módulo principal que usa el resto, llamado *controlV3_1.py*. Los demás están orientados a la comunicación con cada subsistema y se cubrirán en la parte de integración. Utilizamos una clase de tipo *Enum* para activar o desactivar subsistemas que no tengamos conectados al sistema central y que, aún así, el código siga funcionando correctamente, y otra clase del mismo tipo para, según en el modo en el que nos encontremos de nuestra máquina de estados, saber qué entradas vamos a tener. Por ejemplo, en el Modo Voz, no tenemos como entrada al subsistema EMG, por lo que en ese caso no estará habilitado y no realizaremos funciones relacionadas con él. En este sistema, las tres funciones principales que ya vimos en el diagrama de flujo son *update_devices()* y *decide()*, junto con *map_command_and_move()*.

- *update_devices()* (Figura 6.11): Primero comprueba si cada subsistema está activado y luego si es una entrada en el modo en el que nos encontramos. Para el caso del subsistema de motores, leeremos su información por I²C en el caso de que nos haya notificado que tiene nueva información. Para los subsistemas EMG, reconocimiento de voz y reconocimiento de objetos, lo haremos en el caso de que se haya pulsado su botón correspondiente. Con respecto al subsistema de sensado de tacto, su información la leeremos directamente en su manenador de interrupción, ya que las funciones relacionadas con él, como son el agarre adaptativo y la vibración de motores, se deben ejecutar en cuanto se detecte tacto. Los subsistemas de salida de audio e iluminación RGB son actuadores, así que no requerimos hacer nada con ellos aquí.
- *decide()* (Figura 6.12): Dependiendo del modo en el que nos encontremos, comprobará si su subsistema asociado le ha proporcionado información nueva a la que previamente le había dado y, en caso afirmativo, obtendrá el comando que le ha mandado y moverá la mano acorde al *mapping* que tiene de a qué comando le corresponde qué tipo de movimiento.
- *map_command_and_move()*: Este método comprueba si el comando que recibe es válido y si el tipo de movimiento de mano que realiza es distinto al previo para, en este caso, abrir la mano en primer lugar y así soltar el objeto y luego, mover la mano con el tipo de movimiento indicado. Estos tipos de movimiento se encuentran definidos en el fichero *config.ini*.

Finalmente, algo importante de este sistema es la utilización del fichero *config.ini*, que tiene parámetros configurables de cada subsistema, como las posiciones máximas

y mínimas de cada motor, las cotas de los sensores de tacto, las direcciones I²C de los esclavos, etc.

```
def __update_devices(self):
    """
    Checks if new information is available from the connected devices and updates them if
    """
    if self.__active_parts[Input.MOTOR] == Status.ON:
        if self.__motor_cont_new_info == True:
            if self.__input_modes[Input.MOTOR] == Status.ON:
                self.__hand.update_status()
                self.__hand.print_status()
            self.__motor_cont_new_info = False

    # Different because the EMG device has not an interrupt line
    if self.__active_parts[Input.EMG] == Status.ON:
        if self.__but_hand_pressed == True: # if the button was pressed we require new info
            if self.__input_modes[Input.EMG] == Status.ON: # TODO revisar cuando acabe el emg
                self.__emg.check_start_recognition()
            self.__emg.send_command()
        #if self.__active_parts[Input.CAPACITIVE] == Status.ON:
        #self.__check_vibration_motors() # update vibration motors using the info of cap
        #self.__emg.send_command()
        #self.__emg.update()
        #self.__but_hand_pressed = False --> NO NECESARIO AQUI

    if self.__active_parts[Input.SPEECH] == Status.ON:
        if self.__speech_new_info == True: # We only have new info when we have previously p
            if self.__input_modes[Input.SPEECH] == Status.ON:
                self.__speech.update() # TODO may need to be inserted a second update reques
            #self.__speech_new_info = False

    if self.__active_parts[Input.OBJECT] == Status.ON:
        if self.__but_camera_pressed == True: # if the button was pressed we require new info
            if self.__input_modes[Input.OBJECT] == Status.ON:
                self.__object.detect_object()
            #self.__but_camera_pressed = False
```

Figura 6.11: Función *update_devices()* del sistema central

```

def __decide(self):
    """
    ...
    Decides dependent on the status of the corresponding mode which data to process.
    ...
    # TODO revisar cuando acabe el emgDevice.py
    if self.__current_state == State.HAND:
        if self.__active_parts[Input.EMG] == Status.ON:
            if self.__but_hand_pressed == True: # if the button was pressed we require new info from emg device
                sleep(4) # TODO adjust the time to wait for the emg to execute the neural network
                self.__emg.update()
                self.__emg.print_status()
                if self.__emg.new_data():
                    self.__map_command_and_move(self.__emg.get_command(), self.__position_mapping_emg)
                self.__but_hand_pressed = False # reset the flag
    elif self.__current_state == State.VOICE:
        if self.__active_parts[Input.SPEECH] == Status.ON:
            if self.__speech_new_info == True:
                if self.__speech.new_data():
                    self.__map_command_and_move(self.__speech.get_command(), self.__position_mapping_speech)
                self.__speech_new_info = False # reset the flag after processing the data
    elif self.__current_state == State.CAMERA:
        if self.__active_parts[Input.OBJECT] == Status.ON:
            if self.__but_camera_pressed == True:
                if self.__object.new_data():
                    self.__map_command_and_move(self.__object.get_best_object(), self.__position_mapping_objectrec)
                self.__but_camera_pressed = False # reset the flag after processing the data

```

Figura 6.12: Función *decide()* del sistema central

7. Integración del Sistema

Este capítulo recoge la integración de los distintos componentes con el sistema principal y entre ellos, tanto a nivel *hardware*, como *software*.

7.1. Integración Hardware

Para llevar a cabo la integración de todos los elementos *hardware*, mi compañero Miguel Ángel Arana, del grado de electrónica, realizó para su TFG el diseño y fabricación de una PCB con la forma y el tamaño de la palma de un niño, con el objetivo de introducirla dentro de la futura estructura mecánica de la prótesis robótica. Todos los esquemáticos de los diseños realizados por él aparecen en el Anexo C a título informativo. Aún así, también llevé a cabo labores relacionadas con la electrónica, como la soldadura de los motores vibradores en las PCB auxiliares del subsistema EMG, o la fabricación de la PCB donde se conectaban los sensores capacitivos. Asimismo, también colaboré en la fabricación de los prototipos de PCB que realizamos inicialmente para hacer pruebas.

La PCB principal incluiría los 3 microcontroladores STM, el micrófono MEMS para el reconocimiento de voz, el driver de motores TLE94112ES, un LED RGB, junto con la Raspberry Pi Zero 2W y todos los componentes electrónicos necesarios para su funcionamiento y conexionado, como reguladores de tensión, osciladores, resistencias, condensadores, LEDs, etc. La podemos ver en la Figura 7.1. Recordemos que en los proyectos en los que nos hemos basado, estos elementos estaban por separado, ocupando un mayor espacio.

La idea original era incluir la electrónica de todos los subsistemas en esta misma PCB por la otra cara, excepto el subsistema de sensado EMG, ya que debe ir en el antebrazo. No obstante, debido a que todavía no se tenía el diseño mecánico y, por tanto, no se sabía la posición de los motores y los dedos, no se pudo realizar. Para subsanar esto y tener un prototipo que permitiera probar toda la electrónica y la programación, se incluyeron conectores en la PCB que permitieran conectar las PCBs de estos subsistemas de manera externa. El conexionado y montaje final de nuestro proyecto se puede ver en la Figura 7.2.

7.2. Integración Software

Para explicar la integración del *software*, comentaremos las clases que hemos creado en Python para la comunicación del sistema central con cada uno de los subsistemas. Estas clases son las que vimos en el diagrama de la Figura 4.2.

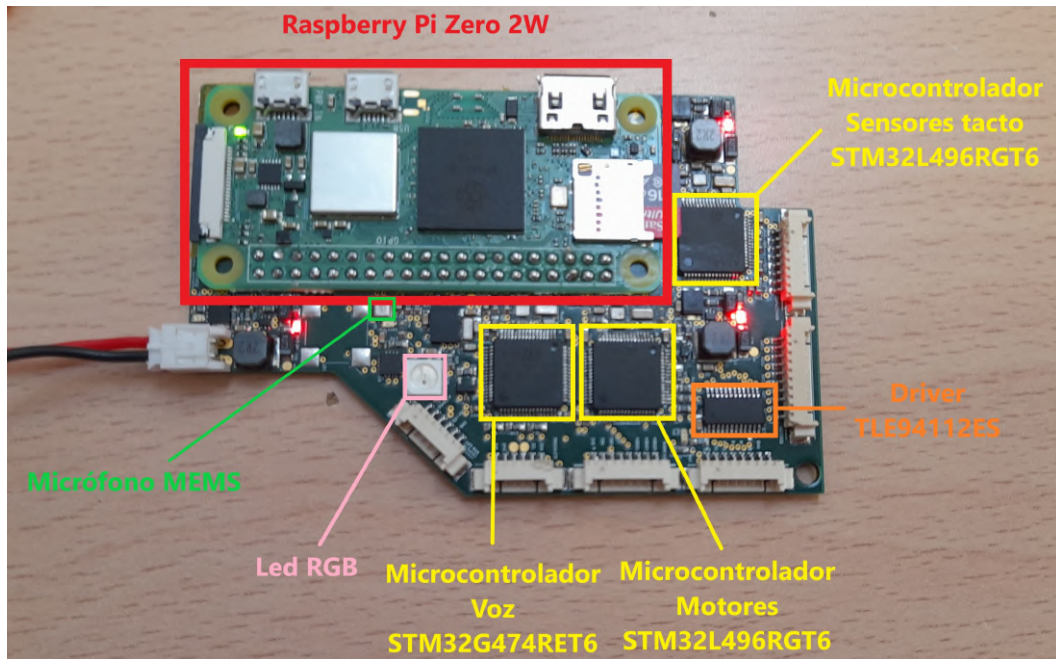


Figura 7.1: PCB integrada con forma de palma

7.2.1. Integración del subsistema de control de motores

Está representado por la clase *HandControl*. De ella, los atributos más importantes y que manejan casi todos los métodos son *status* y *target*. Ambos son dos diccionarios en los que cada clave es el nombre de un dedo y lo que cambia es el significado de los valores. En *status*, es la posición de cada dedo que ha enviado el microcontrolador hacia la Raspberry, mientras que en *target* es la posición límite que se le manda a cada dedo para que se mueva hacia ella. Ambos representan, por tanto, el mensaje I²C de ida y el de vuelta que vimos en la sección de diseño.

De los métodos implementados para esta función, los más importantes y que se usan en el sistema central para mover la mano en base a la información de otros sistemas son los siguientes:

- *move_open_close()*: Se emplea en la función *map_command_and_move()* de nuestro sistema principal para abrir la mano completamente antes de realizar el nuevo tipo de movimiento. Esta función está diseñada para recibir un diccionario siendo las claves los dedos que se quieren mover y los valores *open* o *close*. Con ella se manda a través de I²C los valores de abrir y cerrar al microcontrolador.
- *move_position()*: También se usa en la función *map_command_and_move()*, llamándola a través de la función *move_fix_position()*, que recibe el número identificador de un tipo de movimiento o agarre que hemos configurado en el fichero *config.ini*. Esta función es de las más importantes, ya que se encarga de casi todos los movimientos de la mano y decodifica los números con los tipos de movimientos o agarres que tenemos configurados, para enviarle por I²C al microcontrolador la posición límite de cada dedo. Actualmente tenemos: movimiento de apuntar con el dedo índice, agarre de un cilindro, agarre de una esfera, agarre de un rotulador, agarre de pinza (con dedo índice y pulgar), agarre lateral, agarre

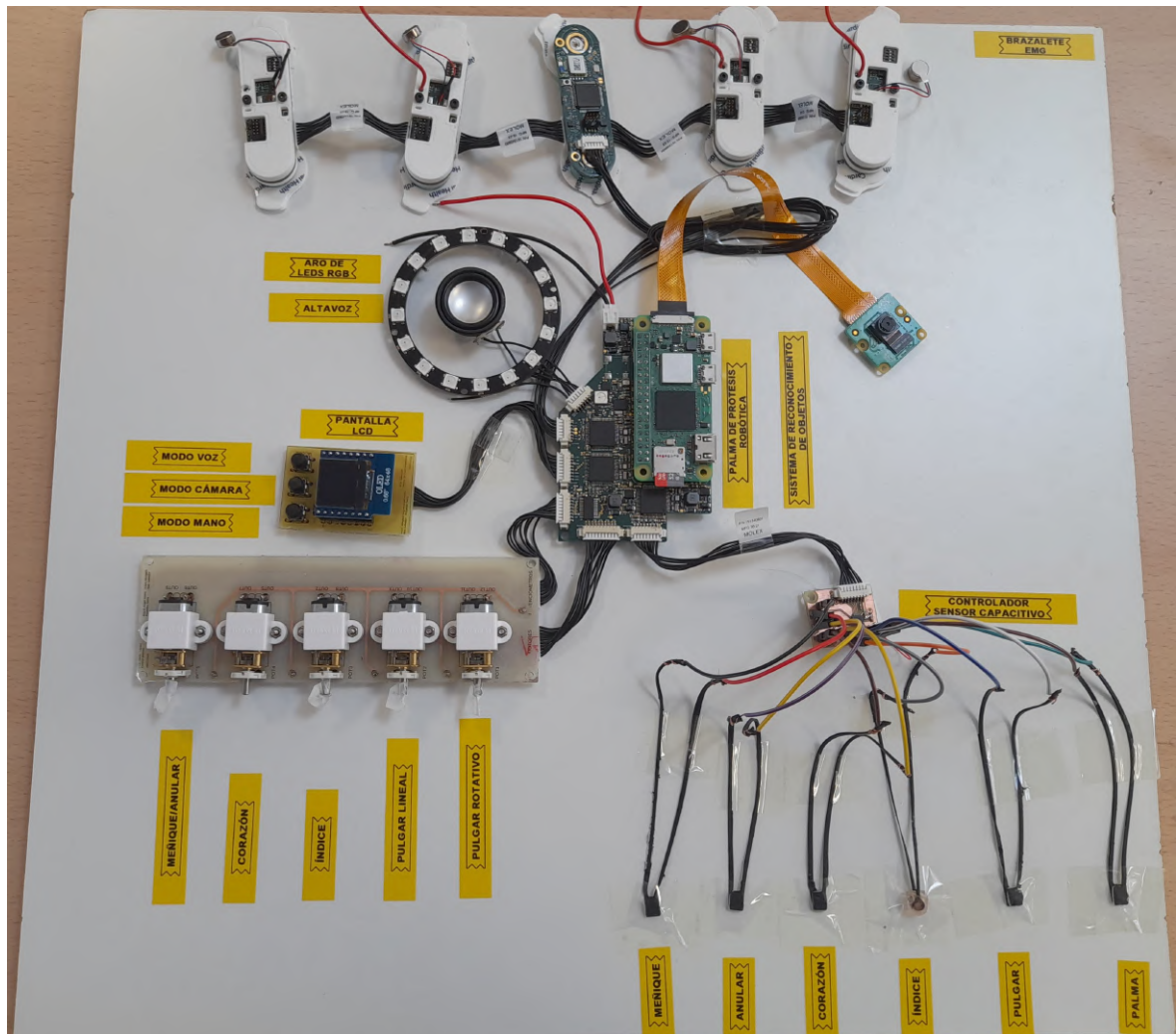


Figura 7.2: Integración de todos los subsistemas con la PCB de la palma

de gancho, agarre de trípode (como pinza pero añadiendo el dedo corazón) y símbolo de la paz. Algunos de ellos los podemos ver en la Figura 7.3. Simplemente, en el archivo de configuración, para cada tipo se pondría qué dedo se mueve y hasta dónde como límite, aunque gran parte de los movimientos son iguales, pero cambiando cuánto debe moverse el motor que dirige el movimiento de rotación del pulgar.

- *update_status()*: Es la función que se encarga de actualizar el *status* de la mano, consultándole datos por I²C a su microcontrolador. Normalmente se llama en el siguiente ciclo tras haber avisado el microcontrolador a la Raspberry por la línea de interrupción de que tiene nuevos datos, activándose su manejador de interrupción, que habilita una bandera.

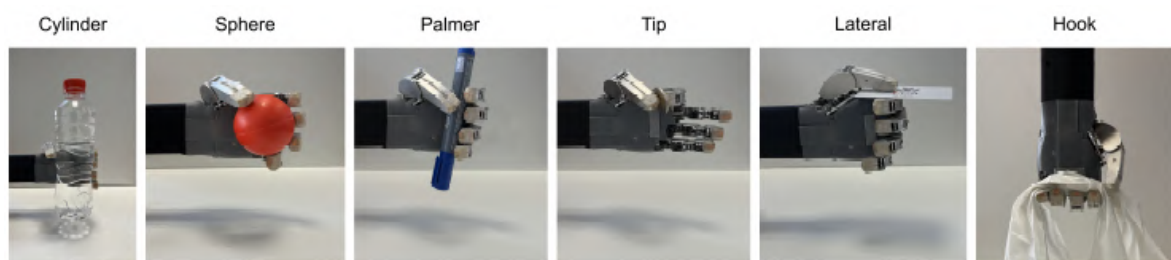


Figura 7.3: Algunos tipos de agarre configurados

7.2.2. Integración del subsistema de sensado de tacto

Al ser este subsistema solo de lectura, la principal estructura que tenemos ahora es *status*, que es un diccionario en el que cada clave identifica a un sensor y el valor es la diferencia de capacidad que nos ha reportado el microcontrolador con respecto a la capacidad en reposo. Por tanto, el método más importante que empleamos en nuestro sistema principal es *update_status()*, que se encarga de leer del microcontrolador la información anterior mediante I²C. Este método se llama cuando el microcontrolador nos activa la línea de interrupción, que hace que en la Raspberry se ejecute su manejador.

Tras ello, en la función *capacitive_heights_check()*, se comprueba la diferencia de capacidad de cada sensor y si alguno supera la cota que tenemos definida en *config.ini*, paramos el dedo asociado a ese sensor, o todos en el caso de que sea el sensor de la palma. Aquí es donde se integra este subsistema con el de control de motores para implementar la función de agarre adaptativo, usando la función *stop_movement()* de la clase *HandControl*, que recibe un diccionario con los dedos que quiera parar como clave y como valor *stop*.

La cota que hemos comentado que aparece en *config.ini* es distinta a la cota que implementamos en el microcontrolador del subsistema. La del microcontrolador era más pequeña, porque significaba que si se superaba, habría comenzado tacto en ese sensor y, por tanto, se debía informar al sistema central. En este caso no es así. Es una cota mayor que indica que si se supera, es porque hay presión suficiente y se debe parar el dedo, porque se considera que ya se está agarrando con la fuerza que debe hacerse.

También se integra este subsistema con el de sensado EMG para implementar la función de retroalimentación del tacto. Para ello, siguiendo en el manejador y tras la función anterior, se ejecuta la función `check_vibration_motors()`. Esta comprueba que el valor de cada sensor sea superior al 80 % de su cota definida para actualizar en ese caso el mensaje que se va a enviar posteriormente al subsistema EMG para que active los motores vibradores correspondientes.

7.2.3. Integración del subsistema de reconocimiento de voz

Como en el caso anterior, seguimos teniendo ahora un subsistema de solo lectura. Por ello, la principal estructura se encuentra en la clase `CmdDevice` y es `status`, pero que ahora está formada por las claves `command` y `detection_quality`. Este diccionario se actualiza con la función `update` que, al igual que antes, es la más importante por ser la encargada de recibir los datos del microcontrolador mediante I²C cuando realizamos la actualización de los dispositivos.

Para poner en marcha el reconocimiento de voz, se debe pulsar el botón, que ejecutará un manejador de interrupciones. En él, se pondrá un pin en alto para despertar al microcontrolador de la voz y que así empiece el reconocimiento. También, en el caso de que vengamos de otro modo, se limpia el `status` previo del subsistema, para evitar que si habíamos dicho el comando `x`, cambiamos a otro modo y volvemos a este modo a decir el comando `x` de nuevo, nos diría que es igual que el anterior y la mano no haría nada, porque su `status` no habría cambiado cuando realmente el estado de la mano sería distinto.

Al finalizar el reconocimiento, el microcontrolador nos activará la interrupción que pone en marcha la ejecución de su manejador para activar la bandera de nuevos datos, que hará que ejecutemos nuestra función `update`. En el caso de que los nuevos datos sean distintos a los anteriores, se moverá la mano en consecuencia con el comando `map_command_and_move()`. En este modo sigue activo el agarre adaptativo y la realimentación del tacto, es decir, la colaboración con el control de motores y el sensado de tacto.

7.2.4. Integración del subsistema de sensado EMG y realimentación del tacto

Volvemos ahora con un subsistema que tiene comunicación bidireccional, ya que manda por I²C si se debe comenzar el sensado EMG y si algún motor debe vibrar, y lee el resultado de la inferencia del sensado. Por ello, al igual que en el subsistema de control de motores, las principales estructuras son `status` y `target`, que se encuentran en la clase `EmgDevice`. La primera de ellas tiene el mismo significado que tenía en el subsistema de reconocimiento de voz, mientras que la segunda está formada por las claves:

- `start_emg_recognition`: Con valor `True` o `False`.
- `motor[6, 7, 1, 2]`: También con valores booleanos.

Una de las funciones empleadas será *update*, que funciona de la misma manera que en el reconocimiento de voz, pero que no se ejecuta tras activarse un manejador porque recordemos que en este subsistema no tenemos línea de interrupción. Por lo tanto, se ejecutará después de esperar unos segundos (suficientes para que le dé tiempo a la red neuronal a inferir y tenga los datos listos) tras activarse la interrupción de pulsar el botón del Modo Mano. En esta interrupción ocurría lo mismo que con el botón del Modo Voz, que se hacía una limpieza del *status*. Tras actualizar el estado, si la información es nueva, se moverá la mano en consecuencia con el comando *map_command_and_move()*.

Las otras funciones importantes y que trabajan en conjunto al ser encargadas de la comunicación hacia el microcontrolador son *update_vibration_motors()*, *send_command()* y *check_start_recognition()*:

- *update_vibration_motors()*: Actualiza la parte de cada motor del diccionario *target*. Se llama desde la función *check_vibration_motors()*, ejecutada por la activación de la interrupción del subsistema de sensado de tacto.
- *check_start_recognition()*: Pone en *True* la clave *start_emg_recognition* del diccionario *target*. Se ejecuta en el caso de que se pulse el botón de Modo Mano.
- *send_command()*: Con el diccionario *target*, forma la trama I²C que vimos en la sección de Diseño y envía el mensaje I²C al microcontrolador. Este mensaje se envía en cada ciclo del sistema principal, dependiendo de cómo se haya actualizado el mensaje con las funciones anteriores.

En este modo sigue activo el agarre adaptativo y, como se ha apreciado, la realimentación del tacto, es decir, la colaboración con el control de motores y el sensado de tacto.

7.2.5. Integración del subsistema de reconocimiento de objetos

Este subsistema es de los más sencillos de integrar, por no depender de comunicación I²C. Su funcionamiento es bastante similar a los anteriores y se encuentra desarrollado en la clase *ObjectRec*. En el caso del botón, actúa del mismo modo que el subsistema de sensado EMG.

Su estructura principal, además de *status*, ahora es *best_status*. Sus claves son el objeto detectado y la calidad de detección. La diferencia entre los dos diccionarios es que el primero se actualiza constantemente cuando se está haciendo el reconocimiento de objetos, mientras que el segundo se queda con el mejor de todos los *status*, es decir, con el de mayor calidad de detección. Por tanto, es el que se utiliza en el sistema principal.

Como funciones empleadas tenemos *detect_object()*, que inicia todo el reconocimiento tras pulsar el botón del modo hasta obtener el mejor objeto, y *get_best_object()*, que devuelve el mejor objeto.

Dicho esto, el funcionamiento es prácticamente igual que en los otros dos modos.

7.2.6. Integración de los subsistemas de iluminación RGB y salida de audio

Están implementados en las clases *RgbControl* y *SpeakerControl*.

La integración de estos subsistemas se lleva a cabo a la hora de pulsar el botón de cada modo. Según el botón pulsado, se activarán unas luces y sonará un audio *mp3* distinto, que indican que hemos entrado en ese modo.

Sus funciones no se mencionan por estar explicadas en la sección [6.2.7](#).

8. Pruebas del Sistema

En este capítulo se presentan los diferentes tipos de pruebas que se han llevado a cabo y sus resultados, para comprobar el correcto funcionamiento del sistema, tanto a nivel individual como integrado con el resto de subsistemas.

8.1. Estrategia

Debido a que tenemos un sistema físico, gran parte de las pruebas se han realizado de manera manual, observando que el funcionamiento del *hardware* era el correcto o comprobando mediante la terminal que los datos obtenidos eran los esperados. Por ello, la documentación de la mayoría de las pruebas se basa en capturas de pantalla de la terminal porque con ella somos capaces de demostrar que el comportamiento del sistema es el que se requiere.

Por supuesto, se han repetido algunas de las pruebas en el caso de que se modificase algún subsistema, para comprobar que el funcionamiento seguía siendo el correcto o que la interacción entre todos los elementos era la esperada.

El entorno en el que se han llevado a cabo las pruebas siempre ha sido real, no simulado, empleando prototipos *hardware* en un inicio que fuesen a funcionar como nuestro sistema final y luego con todos los elementos definitivos.

8.2. Pruebas de Funcionamiento

En esta sección se mostrarán las pruebas realizadas para comprobar que el funcionamiento del *hardware* era correcto y que no había errores en el mismo por soldadura, diseño, etcétera, así como pruebas para comprender el funcionamiento de algunos circuitos integrados.

8.2.1. Pruebas de la placa de pruebas del microcontrolador STM32L496RGT6

Como el diseño y fabricación de la PCB final eran algo que llevaban bastante tiempo y eran muy caros, decidimos diseñar y fabricar una placa de pruebas con uno de los elementos críticos, un microcontrolador STM32L496RGT6, como primer concepto y prueba de diseño *hardware*. Además, con esto se podría ir trabajando y probando los códigos de los distintos subsistemas. Esta placa ya la pudimos ver en la Figura 3.3.

La primera prueba y la más importante consistía en comprobar que la programación era correcta. Para ello, primero empleamos el *ST-LINK* de STM (Figura 8.1)

para probar la interfaz de programación *SWD*. La programación fue correcta a través del propio IDE de STM y empleamos un programa que hacía parpadear el LED que teníamos en la placa, lo cual fue un éxito. Luego pasamos a utilizar la Raspberry con OpenOCD instalado para comprobar que con ella podríamos programar el microcontrolador con el mismo programa. Esta prueba fue importante, porque si la programación de los microcontroladores a través de la Raspberry mediante OpenOCD no era correcta, habría que modificar el diseño de la PCB final para incluir interfaces físicas que permitieran realizar la programación mediante el ST-LINK, por ejemplo. Tras realizar la conexión física, conseguimos realizar la programación correcta mediante OpenOCD. El proceso de programación con este medio se puede ver en el Anexo [A.2.2](#).



Figura 8.1: Programador ST-Link

Otra comprobación que hicimos fue con el bus I²C. Conectamos esta placa a una NUCLEO-F446RE, para que actuara como maestro y luego como esclavo. Primero hicimos el envío de 1 byte y, cuando funcionó correctamente, se enviaron 11 bytes. También se probó la comunicación bidireccional y con modos de *polling* e interrupción, lo que ayudó a comprender el uso del bus I²C en distintas condiciones. Asimismo, se probaron distintas velocidades: *standard* a 100 KHz, *fast mode* a 400 KHz y *fast mode+* a 1 MHz. Para medir que estas velocidades eran correctas, usamos un osciloscopio, con el que obtuvimos los siguientes resultados:

- *standard mode*: velocidad de unos 100 KHz.
- *fast mode*: velocidad de unos 250 KHz.
- *fast mode+*: velocidad de unos 625 KHz.

A su vez, como la idea era que este microcontrolador fuese esclavo de la Raspberry, también hicimos estas pruebas pero utilizándola a ella como maestro. De este modo, podríamos realizar pruebas tanto del *hardware* que tendríamos, como del código final utilizado para la comunicación. Las pruebas en este caso también resultaron correctas.

8.2.2. Pruebas para comprender el funcionamiento del AD7147

Este circuito integrado, necesario para medir la capacidad de los sensores capacitivos, nos causó muchas dudas en un principio porque en el trabajo previo en el que se empleó no se explicaba su funcionamiento y el propio *datasheet* no nos dejaba

claro cómo interpretar los datos que devolvía de cada sensor. Es por esto que realizamos pruebas específicas simplemente para comprender cómo funcionaba, diseñando y fabricando la placa de pruebas que vimos en la Figura 3.4.

Tras el estudio de la hoja de datos, vimos que lo más lógico era que el valor que devolvía por sí solo no tenía sentido, más allá de un aumento o disminución de la capacidad del sensor leído ya que, entre otras cosas, el valor en reposo del sensor te recomendaba establecerlo siempre empleando unos *offsets* en torno a 32700 (el valor medio de 16 bits) para que no se saliese de rango. Así, si tú mismo decides el valor base, ¿cómo lo interpretas? Por ello, no se daba mucha más explicación en la hoja de datos, ya que este integrado (y prácticamente todos los del mercado usados para medir capacidad) se empleaba simplemente para saber si ha habido contacto de un dedo con el sensor, no para saber cuánta presión se ha ejercido. Un ejemplo típico en el que se aplica esto es en las vitrocerámicas táctiles o en las pantallas de los teléfonos móviles. Es por esto que pensamos que lo que tenía sentido era tomar la diferencia entre el valor base que tuviésemos en reposo y el nuevo valor apretando el sensor, lo que nos daría la diferencia de capacidad.

Sabiendo esto, junto con que la resolución del integrado era de 16 bits y el rango total de medida era de 16 pF (+8 y -8), tendríamos $\frac{16}{2^{16}}$ pF/valor. Es decir, si nos devuelve un 1 de diferencia, tendremos una variación de capacidad de $\frac{16}{2^{16}}$ pF. Si quisiéramos saber nuestra capacidad final con la variación aplicada, solo debemos sumar a la capacidad en reposo de nuestro sensor (medida con otro aparato específico) la variación de capacidad medida.

Para comprobar si esta idea era correcta, realizamos distintas pruebas midiendo, en primer lugar, la capacidad de un sensor capacitivo de prueba en reposo utilizando un instrumento de medida específico, para luego colocarle pesas distintas que ejerciesen constantemente la misma presión. Con esto, podíamos medir la capacidad bajo esa presión y calcular la diferencia para compararla con la diferencia medida por el AD7147 (Figura 8.2).

Como muestra, expondremos aquí algunas de las medidas.

Mediciones con aparato específico de medida de capacidad:

- Valor base: -7,6 pF.
- Con pesa de 50 gr: -6,6 pF. 1 pF de variación.
- Con pesa de 100 gr: -6,2 pF. 1,4 pF de variación.
- Con pesa de 200 gr: -6 pF. 1,6 pF de variación.
- Con pesa de 500 gr: -5,2 pF. 2,4 pF de variación.

Mediciones con el integrado AD7147:

- Valor base: 32300.
- Con pesa de 50 gr: 36300. Diferencia de 4000 con el valor base. Aplicando la fórmula, obtenemos 0,97 pF de variación.
- Con pesa de 100 gr: 37700. 1,32 pF de variación.

- Con pesa de 200 gr: 39800. 1,83 pF de variación.
- Con pesa de 500 gr: 42000. 2,37 pF de variación.

Tal y como podemos observar, las mediciones con el integrado son prácticamente iguales a las mediciones específicas de capacidad, por lo que podemos decir que nuestra idea acerca de la interpretación de los valores obtenidos era correcta.

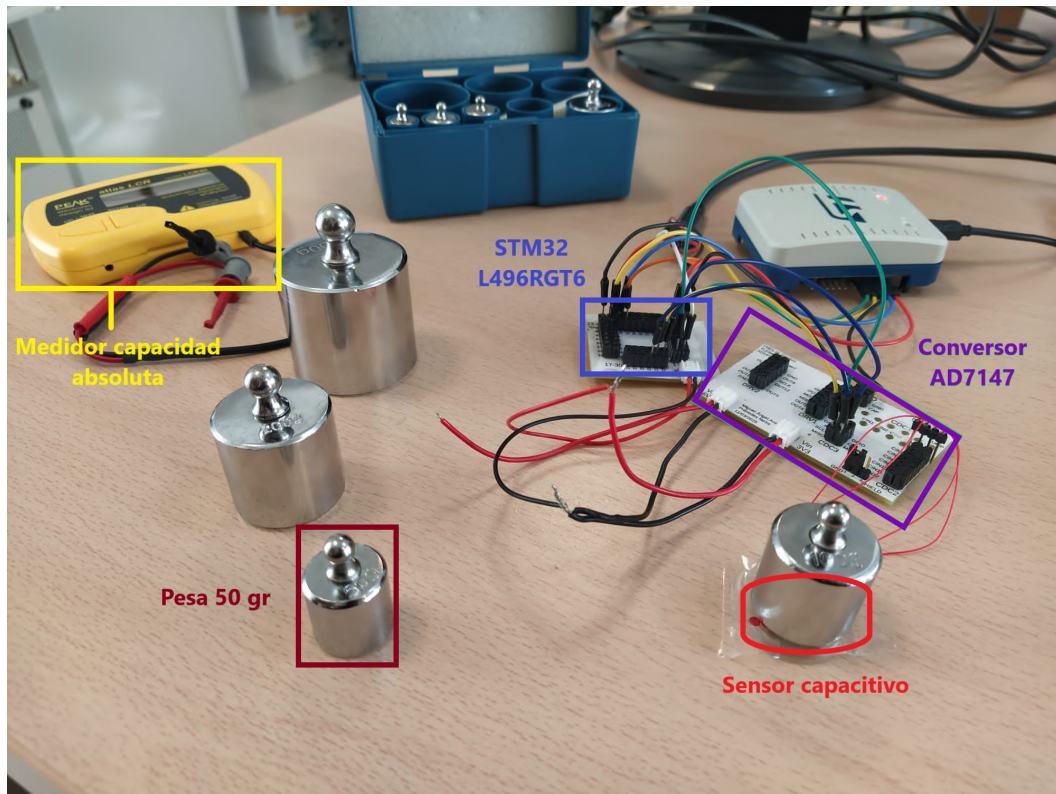


Figura 8.2: Pruebas con distintas pesas para medir las variaciones de capacidad

Por otro lado, algo que observamos con los sensores capacitivos es que la variación de capacidad al tocar con un dedo era muchísimo mayor a la variación obtenida al presionar con un objeto. Por ejemplo, si tocábamos la pesa mientras estaba puesta sobre el sensor, la variación era mucho mayor. Por ello, se intentó configurar esto a través de distintos registros del integrado, calibrando la sensibilidad. No obstante, este comportamiento era independiente de la configuración, ya que la propia naturaleza de un sensor capacitivo hace que el funcionamiento sea este.

8.3. Pruebas Unitarias

Este tipo de pruebas fue realizado para comprobar el correcto funcionamiento del código de cada subsistema de manera independiente, sin colaboración ni comunicación con la Raspberry.

8.3.1. Pruebas del subsistema de control de motores

Para realizar las pruebas de este subsistema empleamos la clase *TestHandControl*, que fue diseñada específicamente para *test*. El entorno de pruebas es el que podemos ver en la Figura 8.3, en el que tenemos 2 motores conectados y un potenciómetro que manejamos manualmente y que vamos conectando a un motor u otro.

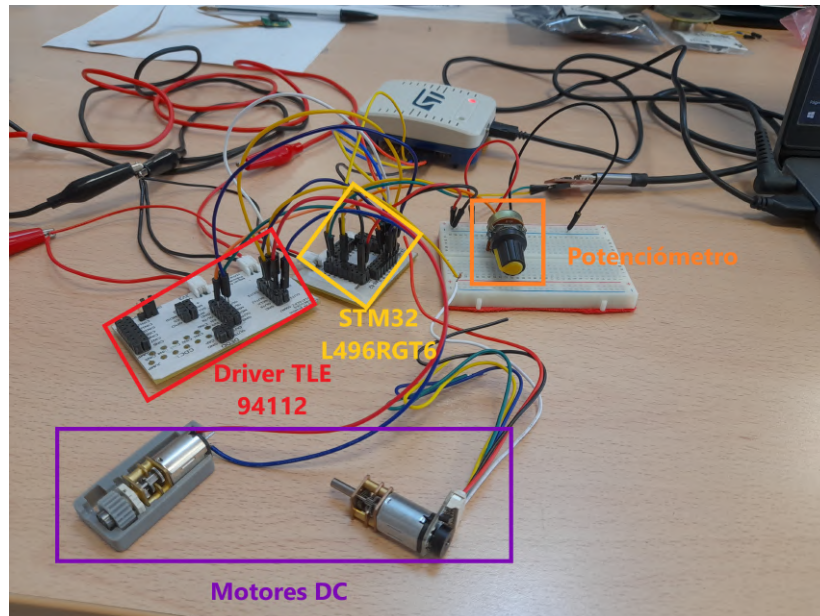


Figura 8.3: Montaje para las pruebas unitarias del subsistema de control de motores

Primero se realizaron pruebas de funcionamiento en las que no interviniese la comunicación I²C, para probar que los motores se movían correctamente y en la dirección debida en base a donde se encontrase su límite, se leía su posición y se paraban cuando llegaban a la posición límite que les habíamos establecido. Al tener un potenciómetro, íbamos girándolo a mano, comprobando que se iba actualizando su posición actual correctamente, hasta llegar o sobrepasar la posición indicada y en ese momento el motor al que estaba asociado se paraba y se activaba la bandera de límite alcanzado. También se activaba la bandera de apertura o cierre del dedo según el potenciómetro estuviese en un lado u otro del límite, moviéndose en el sentido correcto cada vez. Algunos resultados de estas pruebas los podemos ver impresos por pantalla en la Figura 8.4, donde los motores se asociaron uno al dedo *ring* y otro al *index*. Los elementos que vemos en cada línea de la imagen son:

- Nombre del motor.
- *Index*: Número asociado a ese motor.
- *Opening Flag*: Bandera que indica si el dedo se está abriendo o no. El valor 4 indica apertura.
- *Closing Flag*: Bandera que indica si el dedo se está cerrando o no. El valor 8 indica cierre.
- *Finger Position*: Valor leído del potenciómetro asociado al motor, que indica la última posición leída del dedo en un rango [0, 4095].

- *Limit*: Posición límite que tiene establecida el motor. Según la posición actual del motor, se moverá en un sentido u otro para alcanzar ese límite. En el momento en el que se compruebe que se ha superado ese límite (superior o inferiormente), el motor debe parar.
- *Limit reached*: Bandera que indica si se ha alcanzado el límite o no. El valor 2 indica que se ha alcanzado.
- *New Limit*: Bandera que indica si se ha recibido un nuevo límite para el dedo y, por tanto, se debe comenzar a mover hacia él.

En la Figura 8.4a se observa cómo el *index*, al tener la bandera de *limit reached* activada, no se está ni cerrando ni abriendo, lo que indica que se ha parado. Por el contrario, el *ring* no tiene esta bandera activada porque no hemos movido el potenciómetro hasta el límite y se ve cómo tiene la bandera de *opening* activada, al ser su posición actual menor que la límite, no como el *middle*, que al detectar una posición mayor, la bandera activada es la de *closing*. Además, en la Figura 8.4b, nos podemos fijar en que también aparece el mensaje I²C para el maestro actualizado, primero byte a byte (*raw values*) y luego formateado al estado seguido de las posiciones actuales.

Aunque no aparece en la imagen, también probamos con un LED la activación de la línea de interrupción en el momento en el que se cumpliera alguna de las condiciones de interrupción.

Finalmente, se probó la simulación de la recepción del comando de parada, tanto general como de un dedo específico (mediante la posición 5000 o el BCG 2), y se efectuó de manera correcta.

```
Finger Status-> ring, Index: 0, Opening Flag: 4, Closing Flag: 0, Finger Position: 721, Limit: 3000, limit reached: 0, New Limit: 1
Finger Status-> middle, Index: 1, Opening Flag: 0, Closing Flag: 8, Finger Position: 3129, Limit: 3000, limit reached: 0, New Limit: 1
Finger Status-> index, Index: 2, Opening Flag: 0, Closing Flag: 0, Finger Position: 2289, Limit: 3000, limit reached: 2, New Limit: 0
Finger Status-> thumb linear, Index: 3, Opening Flag: 4, Closing Flag: 0, Finger Position: 1663, Limit: 3000, limit reached: 0, New Limit: 1
Finger Status-> thumb rotate, Index: 4, Opening Flag: 4, Closing Flag: 0, Finger Position: 1381, Limit: 3000, limit reached: 0, New Limit: 1
```

(a) Impresión por pantalla del estado de cada motor

```
Finger Status-> ring, Index: 0, Opening Flag: 4, Closing Flag: 0, Finger Position: 719, Limit: 3000, limit reached: 0, New Limit: 1
Finger Status-> middle, Index: 1, Opening Flag: 0, Closing Flag: 8, Finger Position: 3128, Limit: 3000, limit reached: 0, New Limit: 1
Finger Status-> index, Index: 2, Opening Flag: 4, Closing Flag: 0, Finger Position: 2263, Limit: 3000, limit reached: 0, New Limit: 1
Finger Status-> thumb linear, Index: 3, Opening Flag: 4, Closing Flag: 0, Finger Position: 1686, Limit: 3000, limit reached: 0, New Limit: 1
Finger Status-> thumb rotate, Index: 4, Opening Flag: 4, Closing Flag: 0, Finger Position: 1381, Limit: 3000, limit reached: 0, New Limit: 1
I2C message
Raw values: 3 2 207 12 56 8 215 6 150 5 101
Formatted: 3 | Positions: 719 3128 2263 1686 1381
```

(b) Impresión por pantalla del estado de cada motor, junto con el mensaje I²C que tiene preparado para el maestro

Figura 8.4: Impresión por pantalla de la prueba de motores

8.3.2. Pruebas del subsistema de sensado de tacto

Para este subsistema también utilizamos su clase de prueba llamada *TestCapacitiveControl*. En este caso solo usaremos dos sensores, por ser los que teníamos disponibles. Uno de ellos estará conectado a la entrada 1 (*CIN7* del AD7147) y el otro a la 3 (*CIN9* del AD7147), simulando los dedos *ring* e *index*, respectivamente. El montaje para realizar las pruebas aparece en la Figura 8.5, donde un sensor está en el dedo impreso en 3D y el otro se encuentra en la mesa.

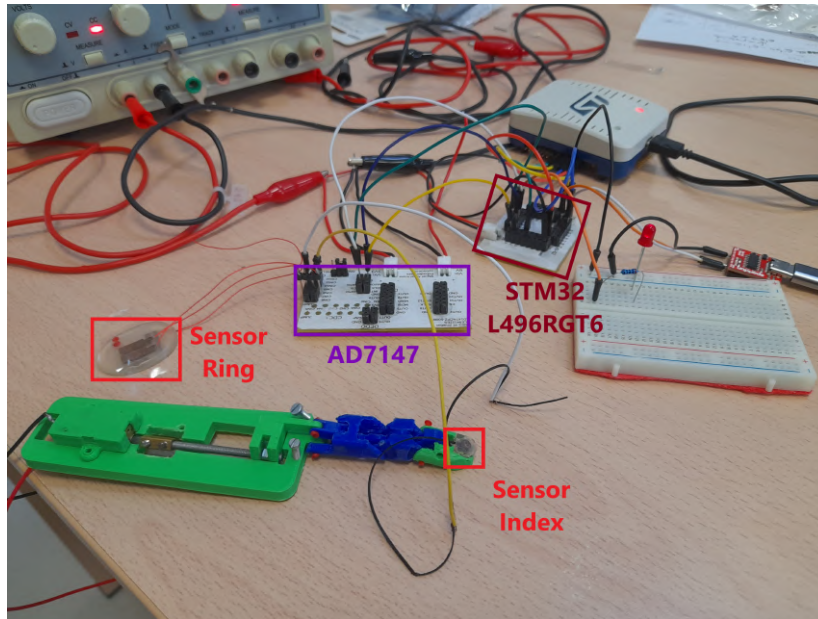


Figura 8.5: Montaje para las pruebas unitarias del subsistema de sensado de tacto

En primer lugar, se probó la función *testI2cMessage()*, que lee los sensores, calcula las diferencias e imprime el valor leído, el valor base, la diferencia calculada, las cotas y actualiza el mensaje I²C que envía al maestro I²C (que debe tener las diferencias) y lo imprime también. El funcionamiento es correcto y aparece en la Figura 8.6.

```
Sensors values: 0 = 65535, 1 = 32879, 2 = 34243, 3 = 33151, 4 = 34182, 5 = 34419,
Sensors differences: 0 = 32835, 1 = 79, 2 = 1543, 3 = 451, 4 = 1482, 5 = 1719,
Sensors base values: 0 = 32700, 1 = 32800, 2 = 32700, 3 = 32700, 4 = 32700, 5 = 32700,
Sensors heights values: 0 = 100, 1 = 1000, 2 = 100, 3 = 1000, 4 = 100, 5 = 100,
I2C message
Raw values: 128 67 0 79 6 7 1 195 5 202 6 183
Formatted (Differences): 32835 79 1543 451 1482 1719
```

Figura 8.6: Salida de la función de prueba *testI2cMessage()*

Luego, se probó la función *testI2cMessageInterrupt()*, que hace lo mismo que la *testI2cMessage()*, pero activando el pin de interrupción, para lo cual he colocado un LED para comprobar que se encendía, y así es.

A continuación, se llevó a cabo la prueba de la función *testHeightsInterrupt()* que hace lo mismo que la *testI2cMessage()* pero sin actualizar el mensaje I²C y añadiendo la comprobación de si se ha sobrepasado la cota de cada sensor o no, actualizando el *status_register* e imprimiéndolo. En el caso de que se haya sobrepasado alguna cota, se activa el pin de interrupción. Una prueba de la salida de esta función la podemos ver en la Figura 8.7. Al imprimir el *status_register* vemos que el bit LSB está a 1, que es el que indica que se ha activado la interrupción I²C al maestro, y el bit 2, que indica que en el sensor del dedo *ring*, la diferencia ha sobrepasado la cota. Para esta prueba, como solo tenemos dos sensores, se han puesto los demás con cotas muy altas para que ninguno pueda activar la interrupción, solo los dos mencionados.

Finalmente, a modo informativo, se ha probado la función *testI2cSensorControl()* que lee los sensores, calcula la diferencia y actualiza el mensaje I²C, midiendo el tiempo

```
Sensors values: 0 = 65535, 1 = 34865, 2 = 34231, 3 = 33099, 4 = 34180, 5 = 34424,
Sensors differences: 0 = 32835, 1 = 2065, 2 = 1531, 3 = 399, 4 = 1480, 5 = 1724,
Sensors base values: 0 = 32700, 1 = 32800, 2 = 32700, 3 = 32700, 4 = 32700, 5 = 32700,
Sensors heights values: 0 = 33000, 1 = 1000, 2 = 33000, 3 = 1000, 4 = 33000, 5 = 33000,
Status register binary = 0b00000101
```

Figura 8.7: Salida de la función de prueba *testHeightsInterrupt()*

que tarda en hacer todo esto. Además, imprime el mensaje I²C y el tiempo de ejecución mencionado. El valor de tiempo aparece en la Figura 8.8 y es de 375 μ s, un valor muy pequeño, que indica que la detección del tacto se hace lo suficientemente rápido.

```
Loop
I2C message
Raw values: 6 187 0 186 128 67 1 171 5 199 6 187
Formatted (Differences): 1723 186 32835 427 1479 1723
Time to execute = 375 us
```

Figura 8.8: Salida de la función de prueba *testI2cSensorControl()*

En general, los resultados de estas pruebas han sido los esperados, viendo un incremento en el valor de diferencia cuando tocamos un sensor y aumentando más cuanto más se presionaba.

8.3.3. Pruebas del subsistema de reconocimiento de voz

En este caso, debido a que parte del funcionamiento iba unido a la PCB final, con botones y la unión con la Raspberry, se probó un código muy parecido, para comprobar principalmente el funcionamiento correcto del modelo de red neuronal implementado en el microcontrolador y la recepción de audio utilizando una PCB de prueba en la que soldamos el micrófono con su electrónica. Además, como aquí ya teníamos una placa de pruebas NUCLEO-G474RE con el mismo microcontrolador que íbamos a tener en la PCB final, se probó sobre ella. La única diferencia con el código final es que en esta prueba despertamos al microcontrolador y, por tanto, comienza su funcionamiento tras pulsar el botón de usuario de la placa NUCLEO, mientras que en el final, se despierta a través de la Raspberry. El montaje empleado aparece en la Figura 8.9.

La idea era que tras pulsar el botón de la placa, se despertase el microcontrolador, se recogieran los datos del micrófono y se imprimiera por pantalla el resultado de la inferencia de la red neuronal. Estos resultados son los que aparecen en la Figura 8.10 para los números *uno* (0x11), *dos* (0x19) y *tres* (0x1d), siendo MSB el número reconocido y LSB el porcentaje de detección. Aún así, las pruebas se realizaron para todos los números, en diversas condiciones del entorno y funcionaron correctamente.

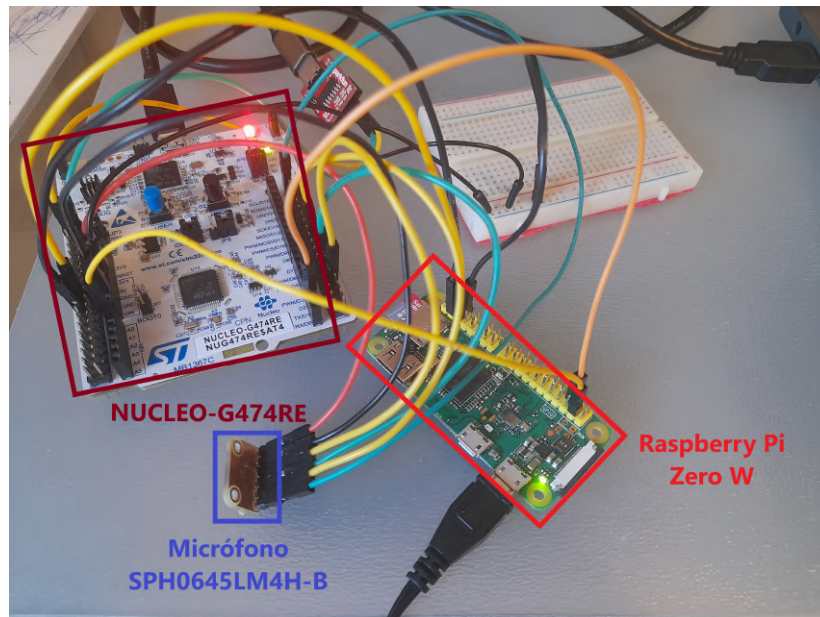


Figura 8.9: Montaje para las pruebas unitarias del subsistema de reconocimiento de voz. El micrófono está soldado a una PCB de pruebas fabricada por nosotros. El modelo de Raspberry usado es Zero W y no Zero 2W por ser con el que se contaba en ese momento.

```

Micro despertado
id_class: 0x11
MSB: 0x11
LSB: 0x4b
Datos antes de enviar
Datos enviados
Durmiendo micro
Micro despertado
id_class: 0x19
MSB: 0x19
LSB: 0x62
Datos antes de enviar
Datos enviados
Durmiendo micro
Micro despertado
id_class: 0x1d
MSB: 0x1d
LSB: 0x61
Datos antes de enviar
Datos enviados
Durmiendo micro
Micro despertado

```

Figura 8.10: Prueba del reconocimiento de 3 números: uno (0x11), dos (0x19) y tres (0x1d)

8.3.4. Pruebas del subsistema de sensado EMG y realimentación del tacto

Para realizar estas pruebas, colocaba el brazalete en mi antebrazo y conectaba el *ST-LINK* al microcontrolador principal de este subsistema, como se ve en la Figura 8.11, para usar el IDE de STM mediante el *debugger* y así poder ver los resultados obtenidos en las variables del propio programa.

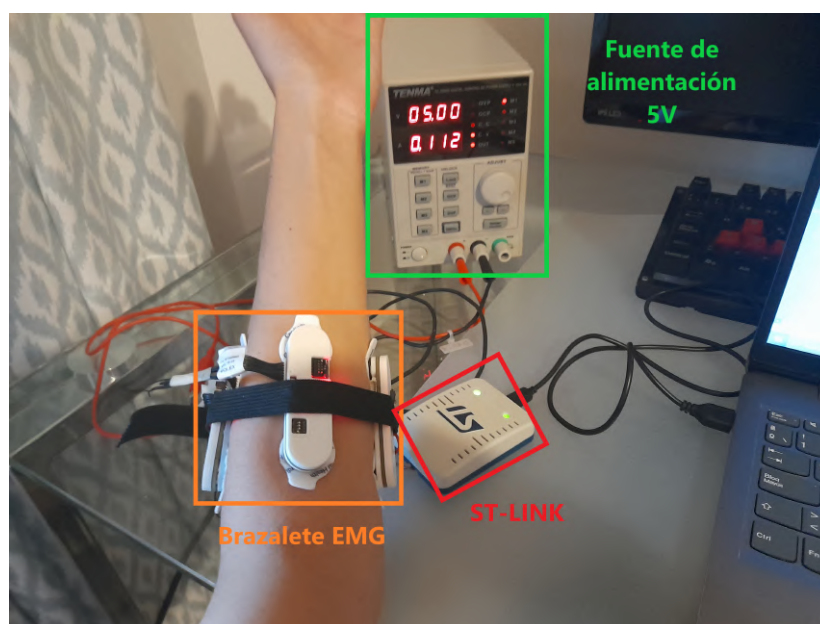


Figura 8.11: Montaje para las pruebas unitarias del subsistema de sensado EMG

Tras varias pruebas, comprobamos que, al hacer lecturas de las posiciones de mi brazo, seguíamos obteniendo una clase casi al 100 % de probabilidad, mientras que en las otras era muy baja. Además, la clase de salida siempre solía ser la misma (véase Figura 8.12) aunque cambiase de gesto con la mano, lo cual no era el funcionamiento esperado. La discusión de estos resultados aparece en la sección 8.5.

Expression	Type	Value
ai_output2	float [4]	0x2000249c <ai_output2>
ai_output2[0]	float	1
ai_output2[1]	float	1.09149162e-034
ai_output2[2]	float	4.16854826e-020
ai_output2[3]	float	2.65391548e-030

Figura 8.12: Predicción errónea del sensado EMG

Por otro lado, si como entrada de la red neuronal le proporcionábamos una muestra del *dataset* con el que realizamos el entrenamiento, la salida era igual que la que se obtuvo de predicción en el entrenamiento (véase Figura 8.13), por lo que podemos descartar cualquier error debido al paso a C del modelo de red neuronal, incluido el preprocesado.

En cuanto a los motores vibradores, utilizando el *debugger* del IDE de STM y conectando el *ST-LINK* al microcontrolador principal, probamos a enviarle desde I²C

```
Probability: [[0.152 0.165 0.484 0.199]]
Predicted class: [2]
```

(a) Predicción de una muestra durante el entrenamiento de la red EMG en Python

Expression	Type	Value
ai_output2	float [4]	0x2000249c <ai_output2>
ai_output2[0]	float	0.151813328
ai_output2[1]	float	0.16528444
ai_output2[2]	float	0.483835995
ai_output2[3]	float	0.199066266

(b) Predicción de la misma muestra en la implementación del microcontrolador en C

Figura 8.13: Predicción de la misma muestra durante el entrenamiento y en el microcontrolador

la trama necesaria para que vibrasen y lo hacían 2 de los 4 motores. Ya que el código es el mismo para todos los microcontroladores auxiliares y la comunicación I²C con ellos es correcta porque envían las características del sensado EMG, suponemos que es debido a algún problema *hardware*. No obstante, los que vibran lo hacían tal y como se esperaba de ellos.

8.3.5. Pruebas del subsistema de reconocimiento de objetos

En estas pruebas, simplemente usamos la cámara conectada a la Raspberry, tal y como aparece en la Figura 8.14.

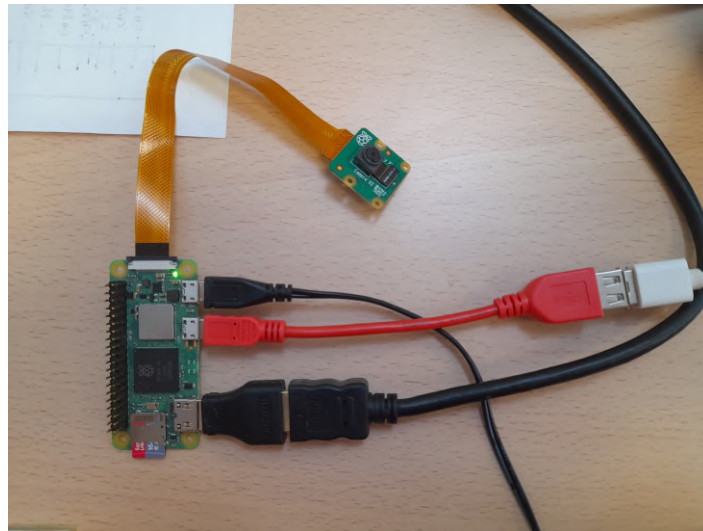


Figura 8.14: Montaje para las pruebas del subsistema de reconocimiento de objetos

El programa de prueba consistía en capturar imágenes durante 2 segundos y reconocer los objetos en ellas, devolviendo el mejor reconocido. Esta prueba era prácticamente igual que el funcionamiento del programa final de este subsistema, pero añadiendo que abriera una ventana nueva para ver las imágenes que estaba capturando la cámara y dibujara en ella los objetos reconocidos. Su funcionamiento era el esperado,

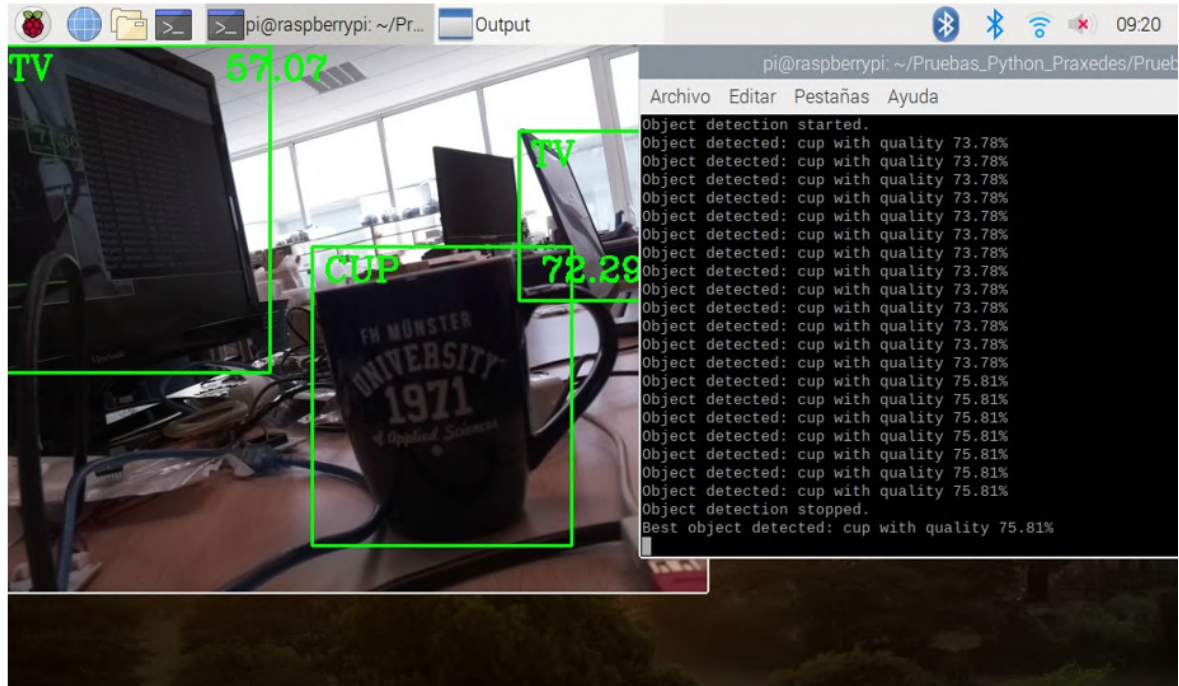


Figura 8.15: Prueba del subsistema de reconocimiento de objetos

con el único problema de que si se movía la cámara, se tardaba bastante en actualizar la imagen y se seguían obteniendo los objetos de la imagen anterior. Una parte de esta prueba se puede ver en la Figura 8.15.

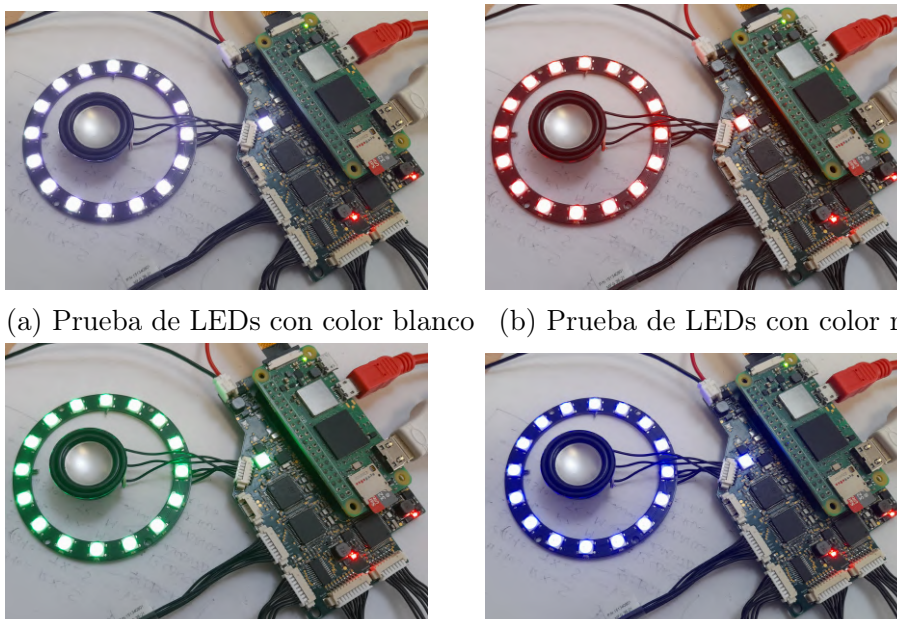
8.3.6. Pruebas de los subsistemas de iluminación RGB y salida de audio

Para el caso de la iluminación RGB, se probó primero el LED que está en la propia PCB final y, tras comprobar que funcionaba correctamente, se incluyó el aro RGB, cuyo funcionamiento también fue correcto. Una prueba de distintos colores se puede ver en la Figura 8.16.

En cuanto a las pruebas del subsistema de salida de audio, no tiene sentido incluir ninguna imagen, pero se realizaron pruebas con distintos archivos en formato *mp3* y el sonido era correcto con algún petardeo en algunas ocasiones, creemos que debido a algún problema del altavoz empleado.

8.4. Pruebas de Integración

Este tipo de pruebas se realizan para comprobar si la interacción entre varios artefactos *software* y, en nuestro caso *hardware*, se realiza de manera correcta.



(a) Prueba de LEDs con color blanco (b) Prueba de LEDs con color rojo

(c) Prueba de LEDs con color verde (d) Prueba de LEDs con color azul

Figura 8.16: Pruebas del aro RGB con distintos colores

8.4.1. Integración del subsistema de control de motores

Principalmente se ha llevado a cabo la comprobación de la comunicación entre la Raspberry como maestro que controla y lee la posición de cada motor y el microcontrolador, empleando para ello el módulo de Python encargado de la integración de este subsistema llamado *HandControl*. El montaje en este caso fue parecido al realizado para las pruebas unitarias, pero añadiendo un segundo potenciómetro y la Raspberry, que realizaba el envío de comandos y recibía el estado de cada motor (véase la Figura 8.17).

Las distintas comprobaciones que se realizaron fueron las siguientes:

- La comunicación es correcta, al inicio los límites de los dedos son sus posiciones iniciales y no se mueven.
- Los motores paran cuando llegan al límite o cuando desde la Raspberry se le dicen que paren.
- Aunque no hayan llegado los motores a su límite, si le mando otro movimiento lo hace y se reinicia la bandera de ese movimiento (*open* o *close*) para poner la otra. Cuando ha llegado a su límite, también se reinicia esa bandera, de modo que ambas están desactivadas. Cuando se le ordena parar, también se reinicia.
- Se transmiten los comandos y la información en ambos sentidos de manera correcta y se activa la bandera de limit reached.
- En la raspberry se lee la información de configuración del fichero *config.ini* de manera correcta.
- Funciona correctamente el hecho de parar un motor (con la función *stop_movement()*)

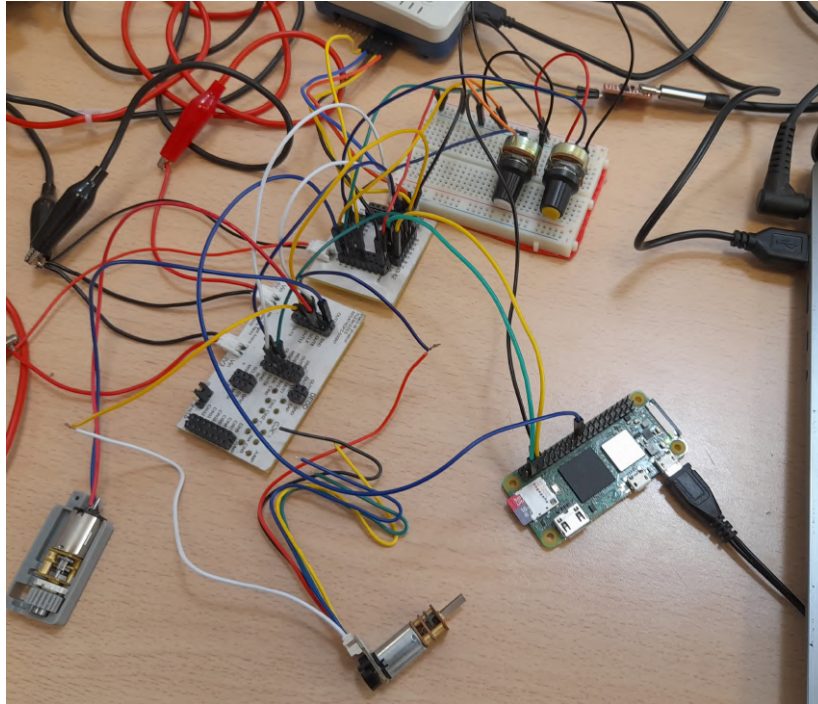


Figura 8.17: Montaje para las pruebas de integración del subsistema de control de motores

mientras que los demás siguen su movimiento normal si no se les dice nada.

- La función *move_position()*, que hace que se mueva de una de las formas predefinidas en el fichero *config.ini*, funciona como se espera.
- Si le mando a un motor el mismo límite que ya tenía antes, no hace nada nuevo ese dedo, sigue tal y como estaba.
- Cuando le llega un límite nuevo para todos los motores (peor caso), tarda en el procesado 111047 us. Si los límites que le mando son los mismos en todos con respecto a los anteriores (mejor caso), el tiempo pasa a ser 334 us, porque como comprueba que son iguales, no le manda por spi otra vez la misma información al *driver*, por eso esa bajada. Si no le mando nada por I²C, es de 310 us el tiempo. Ese extra de 24 us es por la información que se debe mandar por I²C al microcontrolador. Estos tiempos son lo suficientemente pequeños, ya que el tiempo que transcurre desde el envío de la orden hasta que actúan los motores es imperceptible.

Un ejemplo de la salida producida en la Raspberry lo podemos ver en la Figura 8.18. En ella vemos cómo se nos dice las posiciones actuales de cada motor y las posiciones límite que le enviamos a cada uno.

8.4.2. Integración del subsistema de sensado de tacto

Al igual que en el caso anterior, para probar la integración se ha empleado el módulo de Python *CapacitiveControl*. El montaje es el mismo que ya vimos en la

```

i@raspberrypi:~/codigo_mano_praxedes $ python3 handControl.py
handControl successfully initialised.
the commands are: read=0; stop=2; move=3;
the I2C communication parameters are: i2c addr=43; i2c offset=0; nr bytes per finger=2; nr cmd bytes=1; nr status bytes=1; nr used fingers/motors=5; nr bytes requested=11, nr bytes t
b writes=11
defined positions are: [{'nr': 0, 'ring': 3900, 'middle': 3900, 'index': 3900, 'thumb0': 3900, 'thumb1': 3900}, {'nr': 1, 'ring': 100, 'middle': 100, 'index': 3900, 'thumb0': 100, 't
thumb1': 100}, {'nr': 2, 'ring': 100, 'middle': 100, 'index': 100, 'thumb0': 100, 'thumb1': 100}, {'nr': 3, 'ring': 100, 'middle': 100, 'index': 100, 'thumb0': 100, 'thumb1': 100}, {'
nr': 4, 'ring': 100, 'middle': 100, 'index': 100, 'thumb0': 100, 'thumb1': 100}, {'nr': 5, 'ring': 3900, 'middle': 3900, 'index': 100, 'thumb0': 100, 'thumb1': 100}, {'nr': 6, 'ring'
: 100, 'middle': 100, 'index': 100, 'thumb0': 100, 'thumb1': 100}, {'nr': 7, 'ring': 100, 'middle': 100, 'index': 100, 'thumb0': 100, 'thumb1': 100}, {'nr': 8, 'ring': 3900, 'middle'
: 100, 'index': 3900, 'thumb0': 100, 'thumb1': 100}]
current status and positions are: {'status': 0, 'ring': 2141, 'middle': 1786, 'index': 1302, 'thumb0': 1614, 'thumb1': 1280}
the current target position is: {'command': None, 'ring': 2151, 'middle': 1767, 'index': 1299, 'thumb0': 1617, 'thumb1': 1278}
from hand control targets: {'command': 3, 'ring': 3900, 'middle': 100, 'index': 3900, 'thumb0': 100, 'thumb1': 100}
current status and positions are: {'status': 3, 'ring': 2151, 'middle': 1785, 'index': 1299, 'thumb0': 1589, 'thumb1': 1275}
from hand control targets: {'command': 3, 'ring': 3900, 'middle': 100, 'index': 3900, 'thumb0': 100, 'thumb1': 100}
current status and positions are: {'status': 3, 'ring': 2151, 'middle': 1786, 'index': 1304, 'thumb0': 1608, 'thumb1': 1280}
from hand control targets: {'command': 2, 'ring': 5000, 'middle': 5000, 'index': 5000, 'thumb0': 5000, 'thumb1': 5000}

```

Figura 8.18: Salida por terminal de Raspberry de la prueba de integración del subsistema de control de motores

Figura 8.19, pero añadiendo la Raspberry mediante I²C.

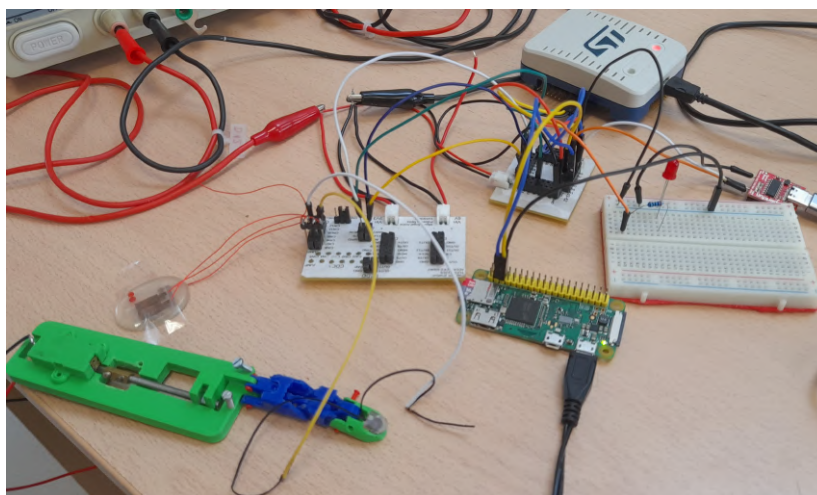


Figura 8.19: Montaje para las pruebas de integración del subsistema de sensado de tacto

Los resultados de las pruebas han sido los siguientes:

- La Raspberry lee correctamente la configuración del fichero *config.ini*.
- Los datos se reciben en la Raspberry correctamente. Estos datos son las diferencias de valores de cada sensor.
- Se activa la interrupción en el microcontrolador correctamente en el *status register* cuando se sobrepasa el valor de cota especificado en el microcontrolador y se desactiva la interrupción cuando lee de él la Raspberry. Si no lee, se queda activado el bit de interrupción. El bit de cota alcanzada solo se activa cuando se sobrepasa esa cota y en el momento en el que ya no se sobrepasa, se desactiva.

Un ejemplo de salida en la Raspberry es el de la Figura 8.20, mientras que en el microcontrolador aparece en la Figura 8.21. Hay que tener en cuenta que solo tenemos conectados los sensores llamados *ring* (segundo valor) e *index* (cuarto valor) y se pulsa el de *ring*. Se ve cómo se sobrepasa la cota (segundo valor de diferencias) y, por tanto, se activa el bit de interrupción (bit LSB que no aparece activado por haber leído el valor desde la Raspberry) y el bit del dedo *ring* (bit 2). El tiempo medido para ejecutar todo el ciclo en el microcontrolador es de 31001 us, pero contando con un *delay* que hay de 30 ms para que la comunicación I²C sea correcta. El primer valor es muy grande

porque no teníamos ningún sensor conectado y, al estar al aire, el comportamiento es indeterminado, así que para evitar problemas pusimos la cota de los sensores no conectados en valores muy altos, para que no activasen el bit de interrupción. De cualquier manera, los valores obtenidos son los esperados, porque el funcionamiento de aumentar la diferencia de capacidad cuando se pulsa ocurre, aumentando más cuando se pulsa con más fuerza.

```

isr was called on channel 20
Datos recibidos [128, 67, 58, 155, 5, 241, 5, 5, 5, 192, 6, 183]
Datos formateados [32835, 15003, 1521, 1285, 1472, 1719]

```

Figura 8.20: Salida por terminal de Raspberry de la prueba de integración del subsistema de sensado de tacto

```

Sensors values: 0 = 65535, 1 = 47803, 2 = 34221, 3 = 33985, 4 = 34172, 5 = 34419,
Sensors differences: 0 = 32835, 1 = 15003, 2 = 1521, 3 = 1285, 4 = 1472, 5 = 1719,
Sensors base values: 0 = 32700, 1 = 32800, 2 = 32700, 3 = 32700, 4 = 32700, 5 = 32700,
Sensors heights values: 0 = 33000, 1 = 2500, 2 = 33000, 3 = 2500, 4 = 33000, 5 = 33000,
I2C message
Raw values: 128 67 58 155 5 241 5 5 5 192 6 183
Formatted (Differences): 32835 15003 1521 1285 1472 1719
Status register binary = 0b00000100

```

Figura 8.21: Salida por terminal de microcontrolador de la prueba de integración del subsistema de sensado de tacto

8.4.3. Integración del subsistema de reconocimiento de voz

Para estas pruebas, el montaje fue mediante la PCB final y la prueba consistía en despertar al microcontrolador cada cierto tiempo para que hiciera el reconocimiento y luego que devolviese por I²C el comando detectado.

El microcontrolador se despertaba correctamente y el reconocimiento de todos los comandos fue el esperado. Lo podemos ver en la Figura 8.22.

8.4.4. Integración del subsistema de sensado EMG

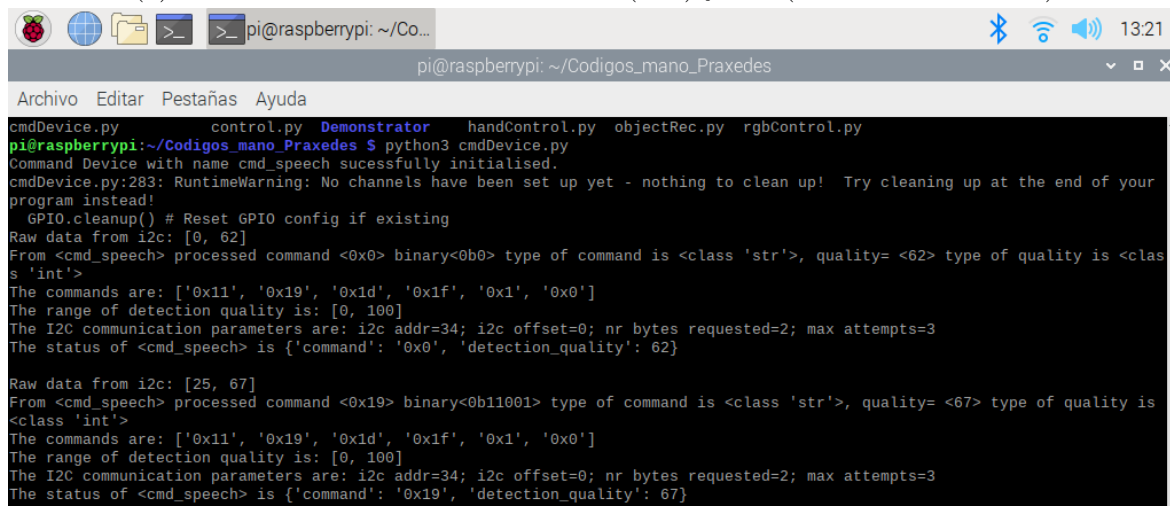
Principalmente se probó que la comunicación bidireccional I²C fuese correcta, es decir, que desde la Raspberry se pudiese enviar el comando de comienzo de sensado EMG y que la recepción del reconocimiento realizado fuese correcta. Para ello, se ha empleado el módulo *EmgDevice* de la Raspberry. El montaje es mediante la conexión de este subsistema al conector específico de nuestra PCB final.

En la Figura 8.23 podemos ver un ejemplo de salida. En ella, en la primera ejecución, el primer mensaje que enviamos no hace nada, mientras que en el segundo ya le decimos que realice el reconocimiento, para luego devolvernos que ha reconocido el 0, que es la posición de puño cerrado. En la segunda ejecución, por el contrario, vemos que nos ha reconocido el 2, que es el símbolo del pulgar.

```
Raw data from i2c: [17, 58]
From <cmd_speech> processed command <0x11> binary<0b10001> type of command is <class 'str'>, quality= <58> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x11', 'detection_quality': 58}

Raw data from i2c: [0, 76]
From <cmd_speech> processed command <0x0> binary<0b0> type of command is <class 'str'>, quality= <76> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x0', 'detection_quality': 76}
```

(a) Reconocimiento del comando 0x11 (uno) y 0x0 (ruido o desconocido)



```
cmdDevice.py control.py Demonstrator handControl.py objectRec.py rgbControl.py
pi@raspberrypi:~/Codigos_mano_Praxedes $ python3 cmdDevice.py
Command Device with name cmd_speech successfully initialised.
cmdDevice.py:283: RuntimeWarning: No channels have been set up yet - nothing to clean up! Try cleaning up at the end of your program instead!
  GPIO.cleanup() # Reset GPIO config if existing
Raw data from i2c: [0, 62]
From <cmd_speech> processed command <0x0> binary<0b0> type of command is <class 'str'>, quality= <62> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x0', 'detection_quality': 62}

Raw data from i2c: [25, 67]
From <cmd_speech> processed command <0x19> binary<0b11001> type of command is <class 'str'>, quality= <67> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x19', 'detection_quality': 67}
```

(b) Reconocimiento del comando 0x19 (dos)

```
Raw data from i2c: [29, 54]
From <cmd_speech> processed command <0x1d> binary<0b11101> type of command is <class 'str'>, quality= <54> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x1d', 'detection_quality': 54}
```

(c) Reconocimiento del comando 0x1d (tres)

```
Raw data from i2c: [31, 81]
From <cmd_speech> processed command <0x1f> binary<0b11111> type of command is <class 'str'>, quality= <81> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x1f', 'detection_quality': 81}

Raw data from i2c: [0, 58]
From <cmd_speech> processed command <0x0> binary<0b0> type of command is <class 'str'>, quality= <58> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x0', 'detection_quality': 58}

Raw data from i2c: [1, 97]
From <cmd_speech> processed command <0x1> binary<0b1> type of command is <class 'str'>, quality= <97> type of quality is <class 'int'>
The commands are: ['0x11', '0x19', '0x1d', '0x1f', '0x1', '0x0']
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=34; i2c offset=0; nr bytes requested=2; max attempts=3
The status of <cmd_speech> is {'command': '0x1', 'detection_quality': 97}
```

(d) Reconocimiento del comando 0x1f (cuatro) y 0x1 (cinco)

Figura 8.22: Salida por terminal de Raspberry de la prueba de integración del subsistema de sensado de tacto

Aunque no aparece en la imagen, también se probó la activación y desactivación de los motores vibradores mediante I²C y fue correcta, teniendo en cuenta, como ya comentamos en el apartado de pruebas unitarias, que solo funcionan 2 de los 4 motores.

```
pi@raspberrypi:~/Codigos_mano.Praxedes $ python3 emgDevice.py
Emg Device with name emg_device sucessfully initialised.
The commands are: [0, 1, 2, 3]
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=51; i2c offset=0; nr bytes requested=2; nr bytes sent=2; max attempts=3
The current target is: {'start_emg_recognition': False, 'motor6': True, 'motor7': False, 'motor1': False, 'motor2': False}
The current target is: {'start_emg_recognition': True, 'motor6': False, 'motor7': False, 'motor1': False, 'motor2': False}
The status of <emg_device> is {'command': 0, 'detection_quality': 100}
pi@raspberrypi:~/Codigos_mano.Praxedes $ python3 emgDevice.py
Emg Device with name emg_device sucessfully initialised.
The commands are: [0, 1, 2, 3]
The range of detection quality is: [0, 100]
The I2C communication parameters are: i2c addr=51; i2c offset=0; nr bytes requested=2; nr bytes sent=2; max attempts=3
The current target is: {'start_emg_recognition': False, 'motor6': True, 'motor7': False, 'motor1': False, 'motor2': False}
The current target is: {'start_emg_recognition': True, 'motor6': False, 'motor7': False, 'motor1': False, 'motor2': False}
The status of <emg_device> is {'command': 2, 'detection_quality': 100}
```

Figura 8.23: Salida por terminal de Raspberry de la prueba de integración del subsistema de sensado EMG

8.4.5. Integración del subsistema de reconocimiento de objetos

Como este subsistema se encontraba desde un principio en la propia Raspberry, las pruebas que se hicieron fueron más sobre el módulo que se encargará de integrarlo en el código final y que se llama *ObjectRec*.

El resultado fue muy parecido a las pruebas unitarias de este subsistema. El cambio principal ha sido la eliminación de la ventana que mostraba por pantalla la imagen con los objetos reconocidos, ya que en nuestro sistema no era necesario y mejoraba la eficiencia, actualizándose rápidamente el objeto que se estaba enfocando en el caso de que se moviese la cámara.

Como la implementación de la detección la hacíamos en un hilo separado, también se probó a eliminar este paralelismo y hacer la detección antes de la actualización, pero se hacían muchas menos detecciones, por lo que el sistema era menos preciso.

Una imagen de estas pruebas la podemos encontrar en la Figura 8.24. Vemos que se reconoce una televisión, lo cual era lo esperado.

```
pi@raspberrypi:~/codigo_mano_praxedes $ python3 objectRec.py
Object recognition with name object_rec sucessfully initialised.
The accepted objects are: ['person', 'cup', 'tv', 'chair', 'scissors', 'bottle', 'mouse', 'keyboard']
The range of acceptable detection quality is: [45, 100]
Object detection started.
Object detected: tv with quality 74.99%
Object detected: tv with quality 74.99%
Object detection stopped.
Best object detected: tv with quality 74.99%
```

Figura 8.24: Prueba de integración del subsistema de reconocimiento de objetos

8.4.6. Integración de los subsistemas de iluminación RGB y salida de audio

Para estos dos subsistemas, se probaron sus módulos llamados *RgbControl* y *SpeakerControl*, con pruebas iguales a las unitarias y el funcionamiento fue el esperado.

8.4.7. Integración completa

Aunque en las pruebas anteriores los montajes del subsistema de sensado EMG y reconocimiento de voz fueron con la PCB final, los demás se pudieron probar con los prototipos. Aún así, las pruebas para estos otros subsistemas se repitieron una vez que tuvimos la PCB final con cada uno de ellos conectado, como vemos en la Figura 8.25 para probar que todo el hardware estaba correcto y los resultados obtenidos fueron como los previos, con la excepción de los motores, porque al anclar los potenciómetros a los ejes de los motores, obteníamos que muchas veces los motores sobrepasaban la posición límite que les habíamos establecido en unas 400 unidades. Este resultado no era el esperado, y no nos dimos cuenta en las pruebas anteriores debido a que los potenciómetros los movíamos a mano, sin estar anclados a los motores. Este funcionamiento no del todo correcto creemos que es debido a que los motores empleados son muy veloces, de reductora 30:1, y se mueven más rápido de lo que el ADC es capaz de actualizar el valor de los potenciómetros. De cualquier manera, damos más detalles de esto en el apartado 8.5.

Sabiendo esto, queda probar que el módulo central, encargado de la coordinación y colaboración de todos los subsistemas anteriores, funciona correctamente en nuestra PCB final y esas serán las pruebas que se mostrarán en esta sección. Antes de comenzar, cabe destacar que el potenciómetro del motor *middle* siempre da unas medidas en el rango [3400, 3500] y no hemos llegado a averiguar por qué, así que para ese dedo dejamos el motor parado por ahora.

En primer lugar, probaremos el funcionamiento del Modo Voz. Lo que debe ocurrir es que, al pulsar el botón, se active la interrupción (ISR) del *speech*, se ponga el color de los LED y suene su audio, se comience el reconocimiento de voz y, cuando acabe, se reciba por I²C el comando reconocido. Tras esto, los motores se moverán en consecuencia hasta la posición asociada a ese comando. Como prueba de este funcionamiento correcto, tenemos las imágenes de la Figura 8.26. Por ejemplo, en la Figura 8.26a, vemos cómo se cambia del Modo Mano al Modo Voz, se escucha su sonido asociado, se recibe la interrupción por parte del microcontrolador de la voz, se leen sus datos que indican el comando 0 y se mueven los motores de las posiciones anteriores a la posición 0, obteniendo una interrupción por parte de su microcontrolador que nos dice dónde ha quedado cada motor. En ella vemos que cada motor queda en una posición cercana a donde se le manda como límite.

En segundo lugar, probaremos el Modo Cámara. El funcionamiento es bastante parecido al Modo Voz, cambiando el comando de voz recibido por el objeto reconocido y el movimiento de los motores hacia la posición asociada a ese objeto. Una prueba del funcionamiento la podemos ver en la Figura 8.27 que, al detectar una persona, la

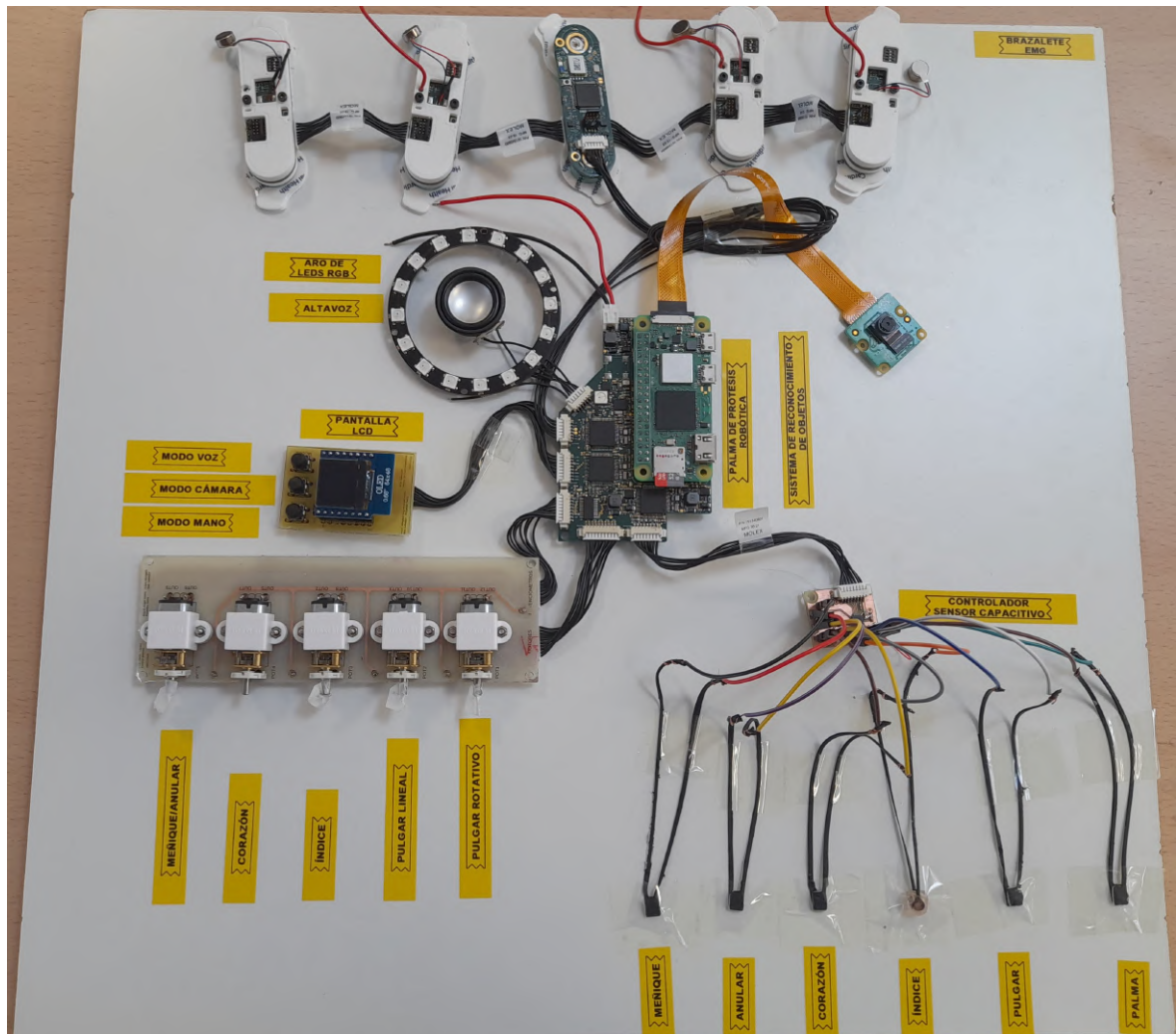


Figura 8.25: Integración de todos los subsistemas con la PCB de la palma

```

pi@raspberrypi: ~/Codigo_mano_Praxedes
Archivo Editar Pestañas Ayuda
Enter state 'State.INIT'
Switched from state 'State.INIT' to state 'State.HAND'.
ISR button speech was called
Switched from state 'State.HAND' to state 'State.VOICE'.
High Performance MPEG 1.0/2.0/2.5 Audio Player for Layers 1, 2 and 3
version 1.25.10; written and copyright by Michael Hipp and others
free software (LGPL) without any warranty but with best wishes

Directory: /home/pi/Music/

Terminal control enabled, press 'h' for listing of keys and functions.

Playing MPEG stream 1 of 1: voiceMode.mp3 ...

MPEG 1.0 L III vbr 44100 j-s

[0:03] Decoding of voiceMode.mp3 finished.
Status and last status of <cmd_speech> cleared.
ISR speech was called
Raw data from i2c: [0, 96]
From <cmd_speech> processed command <0x0> binary<0b0> type of command is <class 'str'>, quality= <96> type of quality is <class 'int'>
From hand control target= {'command': 3, 'ring': 200, 'middle': 3700, 'index': 3700, 'thumb0': 3700, 'thumb1': 3700}
Hand is opened.
Current status and positions are: {'status': 3, 'ring': 544, 'middle': 3368, 'index': 3200, 'thumb0': 3756, 'thumb1': 3824}
Enter move fix position with pos nr= 0
From hand control target= {'command': 3, 'ring': 600, 'middle': 3200, 'index': 3200, 'thumb0': 3200, 'thumb1': 3200}
ISR mc was called
Current status and positions are: {'status': 3, 'ring': 752, 'middle': 3455, 'index': 3200, 'thumb0': 3114, 'thumb1': 3584}

```

(a) Entrada al Modo Voz con su sonido asociado y movimiento de los motores a la posición relacionada con el comando 0

```

s 'int'>
From hand control target= {'command': 3, 'ring': 200, 'middle': 3700, 'index': 3700, 'thumb0': 3700, 'thumb1': 3700}
Hand is opened.
Current status and positions are: {'status': 3, 'ring': 544, 'middle': 3368, 'index': 3200, 'thumb0': 3756, 'thumb1': 3824}
Enter move fix position with pos nr= 0
From hand control target= {'command': 3, 'ring': 600, 'middle': 3200, 'index': 3200, 'thumb0': 3200, 'thumb1': 3200}
ISR mc was called
Current status and positions are: {'status': 3, 'ring': 752, 'middle': 3455, 'index': 3200, 'thumb0': 3114, 'thumb1': 3584}
ISR button speech was called
We are already in the voice state, so we require new info from the voice microcontroller.
ISR speech was called
Raw data from i2c: [0, 83]
From <cmd_speech> processed command <0x0> binary<0b0> type of command is <class 'str'>, quality= <83> type of quality is <class 'int'>
The last commands are the same. So, no command is send to the MC.
ISR button speech was called
We are already in the voice state, so we require new info from the voice microcontroller.
ISR speech was called
Raw data from i2c: [17, 54]
From <cmd_speech> processed command <0x11> binary<0b10001> type of command is <class 'str'>, quality= <54> type of quality is <class 'int'>
From hand control target= {'command': 3, 'ring': 200, 'middle': 3700, 'index': 3700, 'thumb0': 3700, 'thumb1': 3700}
ISR mc was called
Hand is opened.
Current status and positions are: {'status': 3, 'ring': 553, 'middle': 3416, 'index': 3247, 'thumb0': 3712, 'thumb1': 3832}
Enter move fix position with pos nr= 1
From hand control target= {'command': 3, 'ring': 3200, 'middle': 600, 'index': 3200, 'thumb0': 700, 'thumb1': 600}
Current status and positions are: {'status': 3, 'ring': 1184, 'middle': 3430, 'index': 3247, 'thumb0': 3455, 'thumb1': 3535}
ISR mc was called
Current status and positions are: {'status': 3, 'ring': 2816, 'middle': 3552, 'index': 3231, 'thumb0': 742, 'thumb1': 2359}

```

(b) Repetición del comando 0 y reconocimiento del comando 1 (0x11) con su movimiento asociado

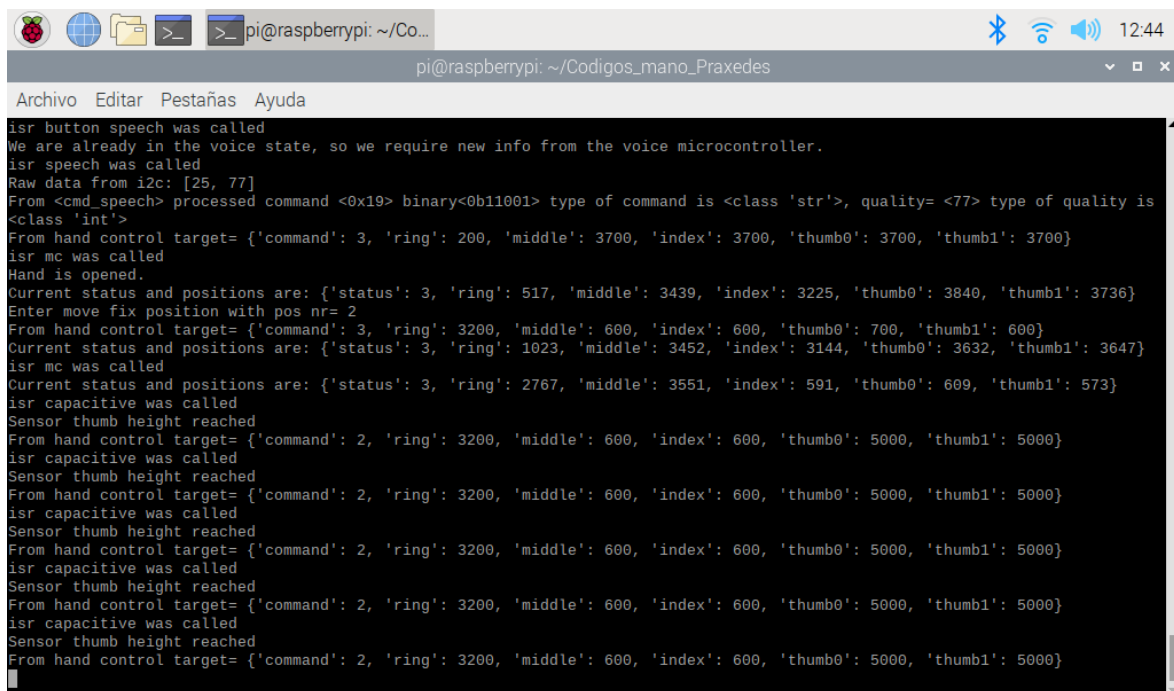
```

ISR button speech was called
We are already in the voice state, so we require new info from the voice microcontroller.
ISR speech was called
Raw data from i2c: [25, 95]
From <cmd_speech> processed command <0x19> binary<0b11001> type of command is <class 'str'>, quality= <95> type of quality is <class 'int'>
From hand control target= {'command': 3, 'ring': 200, 'middle': 3700, 'index': 3700, 'thumb0': 3700, 'thumb1': 3700}
Hand is opened.
Current status and positions are: {'status': 3, 'ring': 573, 'middle': 3380, 'index': 3215, 'thumb0': 3743, 'thumb1': 3785}
Enter move fix position with pos nr= 2
From hand control target= {'command': 3, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 700, 'thumb1': 600}
ISR mc was called
Current status and positions are: {'status': 3, 'ring': 2816, 'middle': 3528, 'index': 2615, 'thumb0': 1225, 'thumb1': 2928}

```

(c) Reconocimiento del comando 2 (0x19) con su movimiento asociado

Figura 8.26: Prueba del funcionamiento del Modo Voz integrado en el programa final

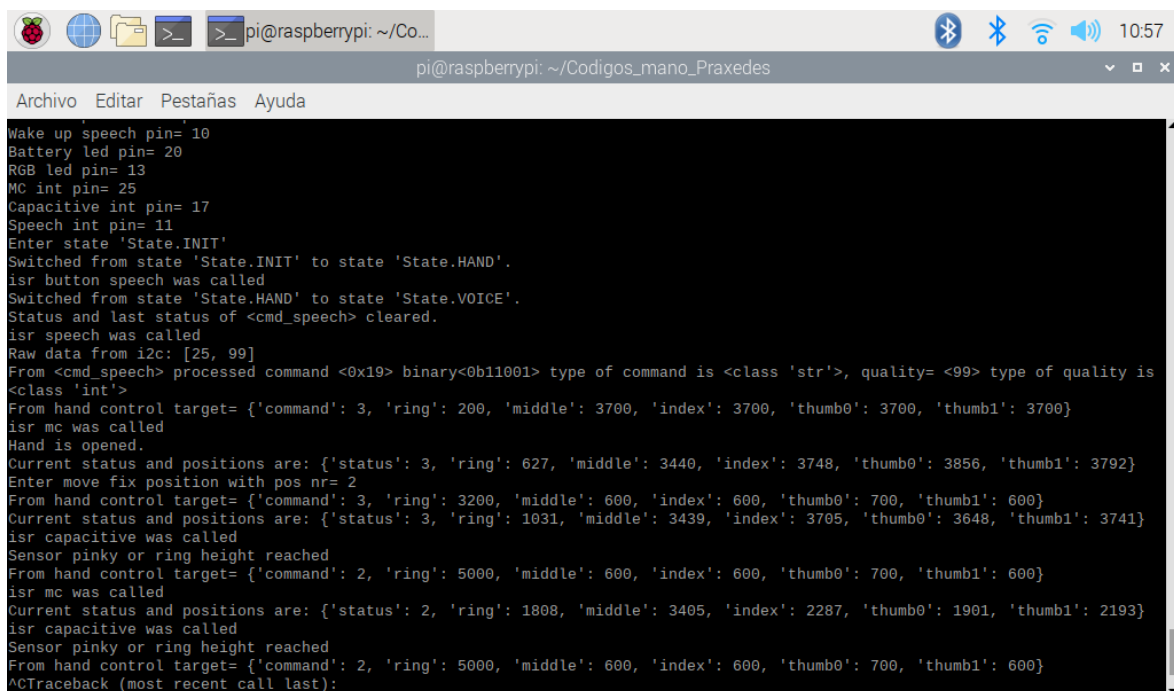


```
pi@raspberrypi: ~/Co...
pi@raspberrypi: ~/Codigos_mano_Praxedes

Archivo  Editar  Pestañas  Ayuda

isr button speech was called
We are already in the voice state, so we require new info from the voice microcontroller.
isr speech was called
Raw data from i2c: [25, 77]
From <cmd_speech> processed command <0x19> binary<0b11001> type of command is <class 'str'>, quality= <77> type of quality is <class 'int'>
From hand control target= {'command': 3, 'ring': 200, 'middle': 3700, 'index': 3700, 'thumb0': 3700, 'thumb1': 3700}
isr mc was called
Hand is opened.
Current status and positions are: {'status': 3, 'ring': 517, 'middle': 3439, 'index': 3225, 'thumb0': 3840, 'thumb1': 3736}
Enter move fix position with pos nr= 2
From hand control target= {'command': 3, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 700, 'thumb1': 600}
Current status and positions are: {'status': 3, 'ring': 1023, 'middle': 3452, 'index': 3144, 'thumb0': 3632, 'thumb1': 3647}
isr mc was called
Current status and positions are: {'status': 3, 'ring': 2767, 'middle': 3551, 'index': 591, 'thumb0': 609, 'thumb1': 573}
isr capacitive was called
Sensor thumb height reached
From hand control target= {'command': 2, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 5000, 'thumb1': 5000}
isr capacitive was called
Sensor thumb height reached
From hand control target= {'command': 2, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 5000, 'thumb1': 5000}
isr capacitive was called
Sensor thumb height reached
From hand control target= {'command': 2, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 5000, 'thumb1': 5000}
isr capacitive was called
Sensor thumb height reached
From hand control target= {'command': 2, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 5000, 'thumb1': 5000}
isr capacitive was called
Sensor thumb height reached
From hand control target= {'command': 2, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 5000, 'thumb1': 5000}
```

Figura 8.28: Prueba de activación de la ISR correspondiente a tocar un sensor de tacto y parada del motor asociado



```
pi@raspberrypi: ~/Co...
pi@raspberrypi: ~/Codigos_mano_Praxedes

Archivo  Editar  Pestañas  Ayuda

Wake up speech pin= 10
Battery led pin= 20
RGB led pin= 13
MC int pin= 25
Capacitive int pin= 17
Speech int pin= 11
Enter state 'State.INIT'
Switched from state 'State.INIT' to state 'State.HAND'.
isr button speech was called
Switched from state 'State.HAND' to state 'State.VOICE'.
Status and last status of <cmd_speech> cleared.
isr speech was called
Raw data from i2c: [25, 99]
From <cmd_speech> processed command <0x19> binary<0b11001> type of command is <class 'str'>, quality= <99> type of quality is <class 'int'>
From hand control target= {'command': 3, 'ring': 200, 'middle': 3700, 'index': 3700, 'thumb0': 3700, 'thumb1': 3700}
isr mc was called
Hand is opened.
Current status and positions are: {'status': 3, 'ring': 627, 'middle': 3440, 'index': 3748, 'thumb0': 3856, 'thumb1': 3792}
Enter move fix position with pos nr= 2
From hand control target= {'command': 3, 'ring': 3200, 'middle': 600, 'index': 600, 'thumb0': 700, 'thumb1': 600}
Current status and positions are: {'status': 3, 'ring': 1031, 'middle': 3439, 'index': 3705, 'thumb0': 3648, 'thumb1': 3741}
isr capacitive was called
Sensor pinky or ring height reached
From hand control target= {'command': 2, 'ring': 5000, 'middle': 600, 'index': 600, 'thumb0': 700, 'thumb1': 600}
isr mc was called
Current status and positions are: {'status': 2, 'ring': 1808, 'middle': 3405, 'index': 2287, 'thumb0': 1901, 'thumb1': 2193}
isr capacitive was called
Sensor pinky or ring height reached
From hand control target= {'command': 2, 'ring': 5000, 'middle': 600, 'index': 600, 'thumb0': 700, 'thumb1': 600}
^CTraceback (most recent call last):
```

Figura 8.29: Prueba de la funcionalidad de agarre adaptativo con Modo Voz

hasta dejar de tocar. El funcionamiento en la realidad no es tan correcto, debido a que tiene que viajar por muchos elementos el mensaje y por la propia construcción del sistema. Es cierto que cuando se toca un sensor la vibración comienza, pero a veces ocurre que si se separa demasiado rápido el dedo del sensor, no lo detecta y el motor sigue vibrando aunque no haya tacto. Esto es así porque la ISR solo se activa cuando se sobrepasa la cota del microcontrolador, pero cuando dejamos de tocar rápidamente, se pasa de estar por encima de la cota del microcontrolador a estar por debajo y el motor solo dejará de vibrar cuando se esté por encima de la cota del microcontrolador y por debajo del 80 % de la de la Raspberry. Al hacer el movimiento rápido, dejamos de estar en ese rango antes de que se active la interrupción, por lo que no se lee el sensor y no se desactiva el motor vibrador. Sin embargo, si lo hacemos suavemente, pasamos de estar por encima de la cota de la Raspberry a estar en el rango y sobre la cota del microcontrolador (con la interrupción activada, por tanto) a, finalmente, estar por debajo del microcontrolador.

8.5. Discusión de los resultados de las pruebas

Tras ver los resultados de las pruebas anteriores, en esta sección se llevará a cabo una discusión de los mismos y se obtendrán conclusiones y propuestas de mejora.

Comenzando con el control de motores, algo importante es que al hacer las pruebas con los potenciómetros unidos a los motores en su propia PCB, cuando se envían posiciones límites a las que ir, en muchas ocasiones dichas posiciones se sobrepasan. Para intentar subsanarlo, en el microcontrolador de motores se paraban estos antes de que llegaran a la posición establecida, pero aún así muchas veces se sobrepasaba. Esto es debido a que los motores que actualmente tenemos en la PCB son motores de reductora 30:1, muy rápidos, con lo que el código de leer los potenciómetros se hace de manera más lenta de lo que el potenciómetro gira. Además, al ser tan rápidos, la inercia hace que también sean muy complicados de controlar de manera precisa. Se sabe que la causa es esta y no *software*, porque cuando hicimos las pruebas con potenciómetros manuales, los motores se paraban en el momento en el que movíamos el potenciómetro a la posición límite. Estos motores se eligieron porque el mecanismo inicial que iban a tener los dedos requería de motores rápidos. No obstante, ese mecanismo presentaba otros problemas y recientemente se ha rediseñado, obteniendo uno nuevo que requiere de motores mucho más lentos. El uso de estos motores mejoraría enormemente este problema con el que actualmente nos encontramos. Otra posible mejora sería cambiar parte del código del microcontrolador para usar el DMA y así obtener una lectura mucho más rápida de los potenciómetros.

Con respecto a los sensores capacitivos, empleamos distintos tipos, siendo unos de PLA, incluido el dieléctrico, y otros de resina. En las pruebas ya vimos que la variación de capacidad era mucho mayor al tocar los sensores con los dedos que al tocarlos con una superficie; sin embargo, hemos observado que los de resina, al tener un aislamiento mayor gracias a la resina, se ven mucho menos afectados por el contacto humano que los de PLA, ya que estos entraban en contacto directo con los objetos. Es por esto que se obtiene un sensado mucho más real, cercano a la presión, con este tipo de sensores

y para una futura prótesis son los que se deberían emplear. Aún así, las medidas eran coherentes en todos los sensores.

Relacionado con el sensado de tacto, tenemos la realimentación del mismo. Se nos presentó el problema de que, al dejar de tocar el sensor, si lo hacíamos rápidamente, podía ocurrir que el motor siguiera vibrando. Una solución que se propone para esto es incluir los motores vibradores directamente conectados al microcontrolador encargado del sensado de tacto, en lugar de que se encuentren en los microcontroladores del subsistema de sensado EMG. Opino que esto es lo que más sentido tiene, ya que actualmente lo que se hace es obtener información del microcontrolador de sensado de tacto y esta información pasa hacia la Raspberry, que la procesa y la envía al microcontrolador principal del subsistema de sensado EMG para, finalmente, llegar a cada uno de sus microcontroladores auxiliares. Si este procesamiento se hiciera en el primer microcontrolador, nos ahorraríamos todo este flujo de mensajes y se obtendría una mayor eficiencia y mejor funcionamiento. Así, estos motores se podrían incluir en un brazalete que estuviese en la muñeca, en lugar de en los microcontroladores que se encargan de leer las señales EMG, puesto que la vibración puede hacer que estas señales se vean afectadas.

Por otro lado, encontramos que la red neuronal implementada en el subsistema de sensado EMG predice de manera incorrecta. En mi opinión, esto ocurre porque el *dataset* de entrenamiento se obtuvo poniendo los sensores EMG en una posición concreta del antebrazo y al ponértelo de nuevo, al no ser las posiciones exactamente las mismas, los datos de entrada nuevos no se parecen demasiado a los de entrenamiento. Una forma de arreglar esto es desarrollar una estructura que haga que siempre te pongas el brazalete en la misma posición exacta. Otra forma es tomar muchísimas muestras, con el brazalete colocado en distintas posiciones y entrenar con este conjunto de datos para que así, sin importar dónde te lo coloques, sepa hacer una predicción con sentido. No obstante, el proceso de obtener el conjunto de datos es lento y bastante tedioso.

Finalmente, el reconocimiento de objetos, de voz, iluminación y sonido funcionaron correctamente en todas las pruebas, así que no queda decir mucho más de ellos.

Parte III

Epílogo

9. Conclusiones

En este último capítulo se detallan los objetivos alcanzados, las lecciones aprendidas tras el desarrollo del presente proyecto y se identifican las posibles oportunidades de mejora sobre el *software* y *hardware* desarrollados.

9.1. Discusión y objetivos alcanzados

Después de todo lo redactado en este documento, se ha podido ver cómo se ha logrado el objetivo principal, al conseguir integrar todas y cada una de las partes *software* existentes previamente junto con las nuevas desarrolladas en nuestro sistema actual, de manera que han podido trabajar de manera colaborativa y emulando el comportamiento de una mano real, demostrándose esta afirmación al haber incorporado funcionalidades como el agarre adaptativo o la realimentación del tacto, además de añadir características lúdicas, como la iluminación RGB y salida de audio, mejorando la experiencia de un usuario infantil.

En cuanto a los objetivos específicos, todos han sido cumplidos por las razones expuestas a continuación.

En primer lugar, se ha llevado a cabo un análisis exhaustivo de los trabajos previos relacionados con la prótesis robótica, identificando sus fortalezas y áreas de mejora. Gracias a esto, se ha podido desarrollar una arquitectura, tanto *hardware*, como *software* que permite la colaboración entre los subsistemas empleando una Raspberry Pi como elemento de control central, consiguiendo establecer una comunicación entre esta y todos los microcontroladores STM32 mediante I²C, además de implementar la programación de los mismos desde la Raspberry Pi usando OpenOCD. Esto permite reprogramar los microcontroladores de manera sencilla e inalámbrica, a través de ella (mediante SSH, por ejemplo), lo que facilitaría mejoras o futuras actualizaciones del *software* sin tener que alterar el *hardware*.

Otro apartado que cabe destacar es la programación mediante el IDE de STM y su biblioteca HAL para manejar periféricos. Aunque parezca sencillo, este IDE, al ser tan completo, cuenta con numerosas funcionalidades que hacen que su aprendizaje sea complejo. Además, el uso y configuración de los periféricos mediante la biblioteca HAL conllevó un tiempo de aprendizaje importante, ya que cuentan con numerosos modos de funcionamiento y diversas configuraciones. No obstante, se consiguió aprender satisfactoriamente, aumentando mi conocimiento y experiencia con una familia de microcontroladores que cada vez es más usada en el entorno profesional.

Asimismo, se rediseñó, integró y mejoró el *software* para la lectura y configuración de sensores capacitivos. Esto se consiguió mediante la implementación y diseño de una estructura de clases que permitía el uso y configuración de los sensores de manera clara y sencilla. También se mejoró su uso al conseguir interpretar de manera correcta los valores devueltos por el circuito integrado AD7147. Esto llevó consigo un gran número

de pruebas, que aparecen en la subsección 8.2.2, con las que obtuvimos un funcionamiento correcto en cualquier entorno, a diferencia de cómo ocurría antes, puesto que con condiciones cambiantes del ambiente, se producían lecturas de valores distintos que llevaban a confusión y hacían que el código no se comportase como se esperaba.

Al párrafo anterior hay que añadir la integración completa del control de los motores, permitiendo la implementación de funcionalidades como el agarre adaptativo bajo la dirección de la Raspberry. Esta nueva funcionalidad hace que, aunque estén predefinidos los tipos de agarre en el código, en función de las dimensiones del objeto a coger, los dedos se muevan hasta una posición u otra, parando en el momento en el que detecten que hay tacto en el sensor capacitivo asociado a dicho dedo. Gracias a esta innovación, el movimiento de la mano se ajustará al objeto que se pretende manipular, pudiendo añadir nuevos tipos de agarre de manera sencilla y dinámica, aportando también seguridad, especialmente al estar enfocado a niños.

Por otro lado, se mejoró e integró el *software* para el reconocimiento de voz. Una de las mejoras fue reescribir el código para emplear las funciones de alto nivel de la biblioteca HAL de STM, que implementa el control de periféricos. Esto permitió ganar claridad y portabilidad en el mismo. Gracias a ello, este código se podría reutilizar sin prácticamente ajustes en cualquier microcontrolador de la familia STM. Además, se ha incluido un modo de bajo consumo para el microcontrolador encargado del reconocimiento, puesto que al ser el más potente que utilizamos y, a su vez, activarse solamente cuando se pulsa un botón, no tenía sentido que estuviese consumiendo tanta energía constantemente. Así, se reduce el consumo, siendo ideal para un sistema como nuestra prótesis, ideado para alimentarse con una batería.

En cuanto al reconocimiento de objetos, también se mejoró e integró su *software*. La mejora principal que se obtuvo fue un reconocimiento del objeto que se está enfocando en ese momento. Y es que anteriormente, si se reconocía un objeto y se enfocaba a otro, se seguía reconociendo el anterior durante un período de tiempo antes de obtener el nuevo. Con esto, el funcionamiento conseguido es mucho más intuitivo y correcto que el que se tenía anteriormente. Aún así, una de las mayores dificultades de este apartado fue la instalación de OpenCV en un dispositivo limitado como una Raspberry, para lo que fue necesario compilarlo en la misma, lo que conllevó varios días.

Tanto en el reconocimiento de objetos como en el de voz, se consiguió mover los motores correctamente en función del elemento reconocido.

Asimismo, se implementó e integró un *software* que permite el control de LEDs RGB y otro para la salida de audio, añadiendo así una experiencia lúdica del usuario infantil.

Por parte del subsistema de sensado EMG, aunque pese a los intentos realizados mediante la mejora de la red neuronal y el entrenamiento con el nuevo *dataset* no se consiguiera arreglar la inferencia implementada por los compañeros, se cumplió con el objetivo de comunicarlo con el sistema central para mover los dedos según el gesto inferido e implementar la nueva funcionalidad de realimentar el sensado de tacto a través de los motores vibradores. Aún así, se mejoró la estructura de la red neuronal y su entrenamiento, realizando un escalado de los datos para reducir el rango de los

mismos y un *resample* para equilibrar las muestras, además de medidas para reducir el *overfitting*, como regularización L2 y *dropout* del 50 %. Gracias a esto, se mejoró el *accuracy* de entrenamiento a 0,96, siendo superior al que obtuvieron los compañeros, que rondaba entre 0,7 y 0,85.

Sumado a todo lo anterior, se debe considerar que de los códigos que han servido como base, se han tenido no solo que readaptar y rediseñar la mayoría de ellos, sino también algo tan básico como cambiar el lenguaje de programación de Arduino a C y C++, consiguiendo una programación homogénea en todos los microcontroladores.

Finalmente, se logró que todos los subsistemas trabajen de forma colaborativa, emulando el comportamiento de una mano real y permitiendo un uso lúdico del sistema.

En resumen, el proyecto ha sentado las bases para una prótesis robótica funcional y adaptable, con un enfoque en la accesibilidad y usabilidad para niños, marcando un avance significativo en el campo de la robótica aplicada.

9.2. Lecciones aprendidas

Gracias a este proyecto he podido adquirir una gran cantidad de buenas prácticas, tanto a nivel tecnológico como procedimental, y que me han servido para mi desarrollo personal y profesional. A continuación, se resumen las lecciones más importantes.

A nivel tecnológico, uno de los aspectos más importantes ha sido la utilización de los protocolos I²C y SPI, completamente necesarios en un proyecto de integración donde la comunicación de los distintos elementos *hardware* es crucial para su correcto funcionamiento y colaboración. En cuanto a la programación, este proyecto me ha servido para aplicar muchos de los conocimientos adquiridos a lo largo del grado, como el paradigma de la programación orientada a objetos, la inteligencia artificial o el diseño e implementación de software para computadores empotrados, así como de una arquitectura que soportase los requisitos del sistema y que permitiera su reutilización *hardware* y *software* en el futuro. Además, el uso de todo el entorno STM me ha proporcionado una experiencia muy valiosa en esta familia de microcontroladores y toda su tecnología relacionada, en auge actualmente en proyectos de esta naturaleza. Asimismo, la integración de modelos de redes neuronales en *hardware* con recursos limitados (*TinyML*) ha supuesto un complemento perfecto a mis conocimientos previos, enseñándome a desarrollar sistemas *edge ai* que cada vez cobran más importancia en la robótica con el incremento de la potencia de los microcontroladores.

Por otro lado, también he aprendido mucho acerca de otra disciplina distinta a la informática, pero que encaja perfectamente con la especialización de ingeniería de computadores que he estudiado, como es la electrónica. En este ámbito, he podido formarme en soldadura y medición de circuitos electrónicos, así como en diseño y fabricación de PCBs. Gracias a este conocimiento, he comprendido y vivido de primera mano cómo es el desarrollo desde cero de prototipos en placas de circuito impreso para su posterior programación.

En cuanto al aspecto procedimental, la planificación ha supuesto un elemento fundamental para la consecución de todos los objetivos propuestos, no solo a nivel de

cumplir fechas, sino también en el cambio de planes según ciertos imprevistos. Por otra parte, al ser un proyecto de naturaleza robótica, de cuya parte electrónica se ha encargado mi compañero Miguel Ángel para su propio Trabajo de Fin de Grado, el entorno de trabajo ha sido multidisciplinar, una experiencia de gran valor y que me ha permitido crecer en el ámbito profesional y personal. En este contexto, el trabajo en equipo y la comunicación continua han sido un pilar clave que nos ha facilitado enormemente la finalización del trabajo.

Finalmente, aunque tenía una base de conocimientos sobre la que trabajar, gran parte de las tecnologías, protocolos o elementos utilizados eran del todo nuevos para mí, algo a lo que no estaba acostumbrado. Por ello, el aprendizaje autónomo ha sido una constante durante el desarrollo del trabajo que me ha proporcionado una evolución profesional enorme.

9.3. Trabajo futuro

Se ha visto que el sistema cumple con los objetivos que se definieron en un inicio. Aún así, hay diversos elementos que pueden ser mejorados:

- **Lectura de la posición de motores:** Aunque con la inclusión de motores más lentos es muy probable que se arregle el problema de que el motor se pare un poco más allá de la posición límite que se le establece, una mejora sería la inclusión del DMA en los potenciómetros para obtener una lectura más rápida de los mismos.
- **Realimentación del tacto:** Una mejora que se le puede hacer y para la que el formato de comunicación I²C está preparada es la vibración variable del motor en función de la cantidad de presión detectada en los sensores de tacto. Para ello, se podría usar PWM con 4 posibles intensidades, ya que en el mensaje I²C ya se incluyen 2 bits para cada motor. En cuanto a un problema que encontramos es la vibración de solo 2 de los 4 motores, seguramente debido a un problema *hardware* en las PCB de los microcontroladores auxiliares del subsistema de sensado EMG. También existe el problema actual de que si se deja de tocar rápidamente un sensor, es posible que el motor correspondiente siga vibrando. Todo esto se podría subsanar de manera simple si se incluyen estos motores vibradores en el microcontrolador encargado del sensado de tacto, lo que nos evitaría tener que mover información por el bus I²C de un extremo a otro, con los posibles fallos que esto puede acarrear, se podría incluir el funcionamiento PWM para muchas más intensidades de vibración y facilitaría el código del sistema central.
- **Reconocimiento de voz:** Se podría mejorar el entrenamiento de la red neuronal de este subsistema de tal manera que se incluyan muchas más palabras para el reconocimiento. Además, estas palabras podrían no solo mover la mano, sino también activar otros subsistemas, como cambiar el color de los LED o hacer sonar una música.
- **Sensado EMG:** El principal problema que presenta este subsistema es que no es capaz de reconocer correctamente el gesto que se hace con la mano y como resultado de la red neuronal se obtiene una salida prácticamente aleatoria. Por

ello, se propone mejorar el subsistema en varios aspectos. En primer lugar en la construcción, ya que para que el funcionamiento actual sea correcto, se debería de colocar siempre en la misma posición exacta, además de que solo serviría para la persona de la que se han extraído las muestras para realizar el entrenamiento. En segundo lugar, quizás una red neuronal no sea la mejor opción, por el problema anterior, ya que el entrenamiento es tedioso y poco eficiente. De esta manera se propone explorar otras alternativas. Finalmente, algo importante sería mejorar también el funcionamiento general, de tal manera que si estamos en el Modo Mano, se hagan detecciones constantemente para que en el momento en el que se cambie de gesto, la prótesis sea capaz de replicar exactamente la posición de los dedos que estamos poniendo, cosa que actualmente no hace, sino que si la posición es semiabierta, se abre completamente.

- **Sensado de temperatura:** Esta característica no sería demasiado difícil de añadir, pero sería muy útil, ya que ahora mismo si por ejemplo tocamos con la prótesis algo muy caliente, no nos avisa de ninguna manera, por lo que podríamos pensar en tocarlo con nuestra mano real y quemarnos o resultar en daños de la propia mano robótica
- **Pantalla LCD:** Actualmente se encuentra incluida una pequeña pantalla LCD, pero que no se utiliza. Se podría emplear como una manera más visual de cambiar entre modos o incluso dar más funcionalidades mediante un menú, como por ejemplo otros modos o visualizar datos relevantes de la prótesis, como la posición de cada dedo.
- **Bluetooth:** Para la comunicación remota con otros elementos, como el subsistema de sensado EMG, se podría incluir esta tecnología, para así eliminar el cableado actual, que puede resultar molesto.

Bibliografía

- [1] P. Medina-Coello, B. Salvador-Domínguez, F. J. Badesa, J. M. Rodríguez Corral, H. Plastrotmann, and A. Morgado-Estévez, “Anthropomorphic Robotic Hand Prosthesis Developed for Children,” *Biomimetics*, vol. 9, no. 7, p. 401, Jul. 2024, number: 7 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/2313-7673/9/7/401>
- [2] O. M. d. l. Salud, *Normas de ortoprotésica de la OMS*. Organización Mundial de la Salud, 2017, type: Publications.
- [3] M. Asghari Oskoei and H. Hu, “Myoelectric control systems—A survey,” *Biomedical Signal Processing and Control*, vol. 2, no. 4, pp. 275–294, Oct. 2007. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1746809407000547>
- [4] T. A. Kuiken, G. Li, B. A. Lock, R. D. Lipschutz, L. A. Miller, K. A. Stubblefield, and K. B. Englehart, “Targeted Muscle Reinnervation for Real-time Myoelectric Control of Multifunction Artificial Arms,” *JAMA*, vol. 301, no. 6, pp. 619–628, Feb. 2009. [Online]. Available: <https://doi.org/10.1001/jama.2009.116>
- [5] D. W. Tan, M. A. Schiefer, M. W. Keith, J. R. Anderson, J. Tyler, and D. J. Tyler, “A neural interface provides long-term stable natural touch perception,” *Science Translational Medicine*, vol. 6, no. 257, pp. 257ra138–257ra138, Oct. 2014, publisher: American Association for the Advancement of Science. [Online]. Available: <https://www.science.org/doi/abs/10.1126/scitranslmed.3008669>
- [6] G. Li, X. Liang, Y. Gao, T. Su, Z. Liu, and Z.-G. Hou, “A Linkage-Driven Underactuated Robotic Hand for Adaptive Grasping and In-Hand Manipulation,” *IEEE Transactions on Automation Science and Engineering*, vol. 21, no. 3, pp. 3039–3051, Jul. 2024. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10124882>
- [7] T. Laliberté, L. Birglen, and C. Gosselin, “Underactuation in robotic grasping hands,” *Machine Intelligence & Robotic Control*, vol. 4, no. 3, pp. 1–11, 2002.
- [8] J. Zuniga, D. Katsavelis, J. Peck, J. Stollberg, M. Petrykowski, A. Carson, and C. Fernandez, “Cyborg beast: a low-cost 3d-printed prosthetic hand for children with upper-limb differences,” *BMC Research Notes*, vol. 8, no. 1, p. 10, Jan. 2015. [Online]. Available: <https://doi.org/10.1186/s13104-015-0971-9>
- [9] C. Antfolk, M. D’Alonzo, M. Controzzi, G. Lundborg, B. Rosen, F. Sebelius, and C. Cipriani, “Artificial Redirection of Sensation From Prosthetic Fingers to the Phantom Hand Map on Transradial Amputees: Vibrotactile Versus Mechanotactile Sensory Feedback,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 21, no. 1, pp. 112–120, Jan. 2013. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/6316177>

- [10] F. Clemente, G. Valle, M. Controzzi, I. Strauss, F. Iberite, T. Stieglitz, G. Granata, P. M. Rossini, F. Petrini, S. Micera, and C. Cipriani, "Intraneural sensory feedback restores grip force control and motor coordination while using a prosthetic hand," *Journal of Neural Engineering*, vol. 16, no. 2, p. 026034, Feb. 2019, publisher: IOP Publishing. [Online]. Available: <https://dx.doi.org/10.1088/1741-2552/ab059b>
- [11] J. L. Finch, G. H. Heath, A. R. David, and J. Kulkarni, "Biomechanical Assessment of Two Artificial Big Toe Restorations From Ancient Egypt and Their Significance to the History of Prosthetics," *JPO: Journal of Prosthetics and Orthotics*, vol. 24, no. 4, p. 181, Oct. 2012. [Online]. Available: https://journals.lww.com/jpojournal/abstract/2012/10000/biomechanical_assessment_of_two_artificial_big_toe.4.aspx
- [12] A. J. Thurston, "Paré and Prosthetics: The Early History of Artificial Limbs," *ANZ Journal of Surgery*, vol. 77, no. 12, pp. 1114–1119, 2007, eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1445-2197.2007.04330.x>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1445-2197.2007.04330.x>
- [13] "Las prótesis a través del tiempo | NIH MedlinePlus Magazine." [Online]. Available: <https://magazine.medlineplus.gov/es/art%C3%ADculo/las-protesis-a-traves-del-tiempo>
- [14] D. S. Childress, "Historical Aspects of Powered Limb Prostheses | O&P Virtual Library." [Online]. Available: https://www.oandplibrary.org/cpo/1985_01_002.asp
- [15] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/6296526>
- [16] S. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 4, pp. 357–366, Aug. 1980. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/1163420>
- [17] S. Choi, S. Seo, B. Shin, H. Byun, M. Kersner, B. Kim, D. Kim, and S. Ha, "Temporal Convolution for Real-time Keyword Spotting on Mobile Devices," 2019. [Online]. Available: <https://arxiv.org/abs/1904.03814>
- [18] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello Edge: Keyword Spotting on Microcontrollers," Feb. 2018, arXiv:1711.07128 [cs]. [Online]. Available: <http://arxiv.org/abs/1711.07128>
- [19] C. Shi, D. Yang, J. Zhao, and H. Liu, "Computer Vision-Based Grasp Pattern Recognition With Application to Myoelectric Control of Dexterous

- Hand Prosthesis,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 28, no. 9, pp. 2090–2099, Sep. 2020. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9144480>
- [20] G. Bradski, “The OpenCV Library.” *Dr. Dobb’s Journal: Software Tools for the Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000. [Online]. Available: <https://www.elibrary.ru/item.asp?id=4934581>
- [21] “OpenCV: Deep Neural Networks (dnn module).” [Online]. Available: https://docs.opencv.org/4.x/d2/d58/tutorial_table_of_content_dnn.html
- [22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” Mar. 2016, arXiv:1603.04467 [cs]. [Online]. Available: <http://arxiv.org/abs/1603.04467>
- [23] F. Chollet, “Keras: Deep Learning for humans.” [Online]. Available: <https://keras.io/>
- [24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019. [Online]. Available: <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html>
- [25] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional Architecture for Fast Feature Embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia*, ser. MM ’14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 675–678. [Online]. Available: <https://dl.acm.org/doi/10.1145/2647868.2654889>
- [26] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” 2016, pp. 779–788. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Redmon_You_Only_Look_CVPR_2016_paper.html
- [27] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single Shot MultiBox Detector,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 21–37.

- [28] A. Harris, K. Katyal, M. Para, and J. Thomas, “Revolutionizing Prosthetics software technology,” in *2011 IEEE International Conference on Systems, Man, and Cybernetics*, Oct. 2011, pp. 2877–2884, iSSN: 1062-922X. [Online]. Available: <https://ieeexplore.ieee.org/document/6084102>
- [29] B. E. Lawson, J. Mitchell, D. Truex, A. Shultz, E. Ledoux, and M. Goldfarb, “A Robotic Leg Prosthesis: Design, Control, and Implementation,” *IEEE Robotics & Automation Magazine*, vol. 21, no. 4, pp. 70–81, Dec. 2014. [Online]. Available: <https://ieeexplore.ieee.org/document/6955797>
- [30] V. Artigue and G. Thomann, “Development of a prosthetic arm: Experimental validation with the user and an adapted software,” in *2009 IEEE International Conference on Robotics and Automation*, May 2009, pp. 3079–3084, iSSN: 1050-4729. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/5152373>
- [31] V. Nikolov, I. Georgiev, and I. Chavdarov, “Hardware and software of a 3D printed humanoid hand,” in *2023 XXXII International Scientific Conference Electronics (ET)*, Sep. 2023, pp. 1–6. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10278809>
- [32] F. Gaetani, P. Primiceri, G. Antonio Zappatore, and P. Visconti, “Hardware design and software development of a motion control and driving system for transradial prosthesis based on a wireless myoelectric armband,” *IET Science, Measurement & Technology*, vol. 13, no. 3, pp. 354–362, 2019, eprint: <https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-smt.2018.5108>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-smt.2018.5108>
- [33] STMicroelectronics, *STM32CubeIDE User Guide*, STMicroelectronics, 2025, accessed: 2025-08-03. [Online]. Available: https://www.st.com/resource/en/user_manual/um2609-stm32cubeide-user-guide-stmicroelectronics.pdf
- [34] —, *STM32CubeMX for STM32*, STMicroelectronics, 2025, accessed: 2025-08-03. [Online]. Available: <https://www.st.com>
- [35] R. P. Ltd, “Buy a Raspberry Pi Camera Module 2.” [Online]. Available: <https://www.raspberrypi.com/products/camera-module-v2/>
- [36] “DC Brush 6V Mini Gear Motor, Micro Turbo Worm Geared Motor Reductor Reductor Motores Dispositivos Electrónicos Lavadora a Presión Piezas Accesorios 1218GEN20 (DC6V 35RPM) : Amazon.es: Industria, empresas y ciencia.” [Online]. Available: <https://www.amazon.es>
- [37] “ingeniero informático Salario en España—Salario medio.” [Online]. Available: <https://es.talent.com/salary?job=ingeniero+informático>
- [38] r. berry, “Todo sobre Raspberry Pi Zero 2 W,” Oct. 2021. [Online]. Available: <https://picoockpit.com/raspberry-pi/es/todo-sobre-raspberry-pi-zero-2-w/>
- [39] “Precio de la tarifa de luz por horas HOY | 04 Agosto 2025.” [Online]. Available: <https://tarifaluzhora.es/>

- [40] I. López-Espejo, Z.-H. Tan, J. H. L. Hansen, and J. Jensen, “Deep spoken keyword spotting: An overview,” *IEEE Access*, vol. 10, pp. 4169–4199, 2022.
- [41] R. B. Azhiri, M. Esmaeili, and M. Nourani, “Real-Time EMG Signal Classification via Recurrent Neural Networks,” in *2021 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Dec. 2021, pp. 2628–2635. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9669872>

Parte IV

Anexos

A. Manual del desarrollador

A continuación se recogen las instrucciones necesarias para evolucionar el software. Este manual está dirigido a los desarrolladores que pretenden extender o modificar el código fuente, con el fin de incorporar nuevas funcionalidades o modificar las ya existentes.

A.1. Introducción

El presente manual incluye los pasos necesarios para instalar las herramientas necesarias para extender o modificar el *software* desarrollado para el control de la prótesis robótica de niños. Se incluye tanto el *software* de control de cada uno de los elementos individuales, como motores, sensores, cámara, etc., como de orquestación de todos estos subsistemas, de manera que puedan trabajar colaborativamente.

A.2. Preparación del entorno de trabajo

En esta sección detallaremos tanto el *hardware*, como el *software* necesario para poder comenzar la evolución de nuestro código.

A.2.1. Hardware

Lo primero que necesitaremos es la PCB principal, con la Raspberry conectada y una tarjeta SD de al menos 16 GB.

Por otro lado, necesitaremos un computador que usaremos para conectarnos a la Raspberry inalámbricamente mediante *ssh* y para poder trabajar sobre los códigos de cada uno de los microcontroladores. Si el acceso será mediante *ssh*, tendremos que tener una red a la que conectar la Raspberry y en la que estará también nuestro equipo.

En el caso en el que el acceso a la Raspberry no lo queramos hacer remotamente, o para la primera vez que se quiera acceder a ella y configurarle una red, será necesario conectarla a un monitor y, mediante un hub, emplear un teclado y un ratón para su manejo.

A.2.2. Software para la Raspberry

Esta sección se dedicará a la instalación de cada uno de los elementos *software* necesarios para la modificación del código fuente de la Raspberry y su correcta ejecución. Recordemos que todo este código está escrito en Python.

Imagen de la tarjeta SD de la Raspberry

Como sistema operativo de la Raspberry Pi, emplearemos *Raspberry Pi “Buster” OS*. Lo podremos instalar en la tarjeta micro SD mediante el programa oficial de Raspberry llamado *Raspberry Pi Imager*, disponible para su descarga en este [enlace](#). Por otro lado, la imagen oficial del sistema puede ser descargada [aquí](#) (ARCHIVO .ZIP). Tras ejecutar el instalador, veremos una imagen como la de la Figura A.1. En esta ventana se configurarán los distintos parámetros del sistema operativo “Buster” que se va a instalar.

En primer lugar, seleccionamos el modelo de Raspberry Pi Zero 2W, luego seleccionamos el sistema operativo que vamos a instalar, para ello pulsamos en *Use Custom*, descomprimos el .zip descargado anteriormente y cargamos el .img en el instalador. Después, seleccionamos como almacenamiento la tarjeta microSD que habremos introducido al PC a través de un adaptador USB.

Por último, se deben configurar unos parámetros avanzados. Para acceder a ellos se pulsa Control+Shift+X. En esta nueva ventana se configura el nombre de anfitrión para la conexión ssh, en este caso se deja por defecto “pi”, se configura el nombre de usuario, contraseña, conexión wifi, zona horaria y distribución de teclado. Luego, en la pestaña de “Servicios” activamos la comunicación ssh con autenticación por contraseña y pulsamos Guardar.

Con esto ya estaría lista la instalación de la imagen “Buster OS” en la microSD. Pulsamos siguiente y confirmamos la instalación con los parámetros introducidos. Cabe destacar que con la instalación de esta imagen sobre la microSD se borrará toda la información previa en la tarjeta.

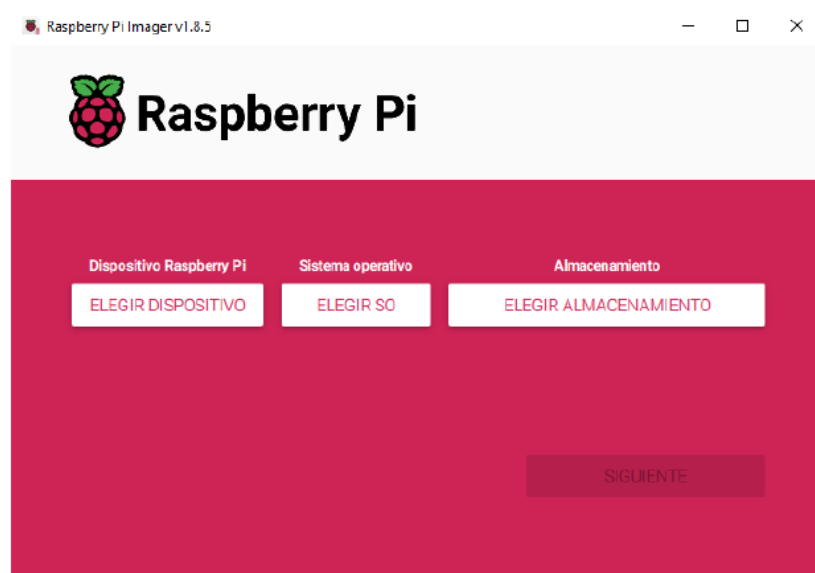


Figura A.1: Raspberry Pi Imager

Cuando iniciemos el sistema, lo primero que haremos será:

```
sudo apt-get update
```

Uso de del I²C de la Raspberry

Para usar el periférico I²C, lo primero que deberemos hacer es habilitarlo. Para asegurarnos de que está habilitado, en la terminal escribiremos:

```
sudo raspi-config
```

e iremos a *Interfacing options* > *I2C* > habilitar. Tras esto, reiniciaremos la Raspberry usando:

```
sudo reboot
```

Ahora instalaremos las dependencias básicas con:

```
sudo apt-get install -y python3-smbus i2c-tools
```

Para, finalmente, instalar la biblioteca *smbus2* que nos permita manejar el bus I²C:

```
python3 -m pip install --upgrade pip
python3 -m pip install smbus2
```

Instalación del software para manejo de LEDs RGB en la Raspberry

Para manejar los LEDs RGB de nuestra PCB a través de la Raspberry mediante Python, necesitaremos usar la biblioteca *NeoPixel* y *board* de Adafruit, para la cual deberemos instalar la biblioteca *Adafruit_Blinka*, que proporciona compatibilidad con CircuitPython en Python. Para ello, simplemente haremos:

```
sudo pip3 install rpi_ws281x adafruit-circuitpython-neopixel
sudo python3 -m pip install --force-reinstall adafruit-blinka
```

Esta instalación se basa en la explicación de Adafruit que se puede ver [aquí](#).

Instalación del software para manejo de altavoz en la Raspberry

Para emplear un altavoz, se decidió usar el amplificador MAX98357AETE+T, ya que Adafruit proporciona soporte para manejarlo.

Sabiendo esto, usamos el siguiente *script* de instalación para poder usarlo:

1. Primero, activamos un entorno virtual:

```
sudo apt install python3-venv
python -m venv env --system-site-packages
source env/bin/activate
```

2. Lo siguiente, fue ejecutar el *script* de instalación siguiente:

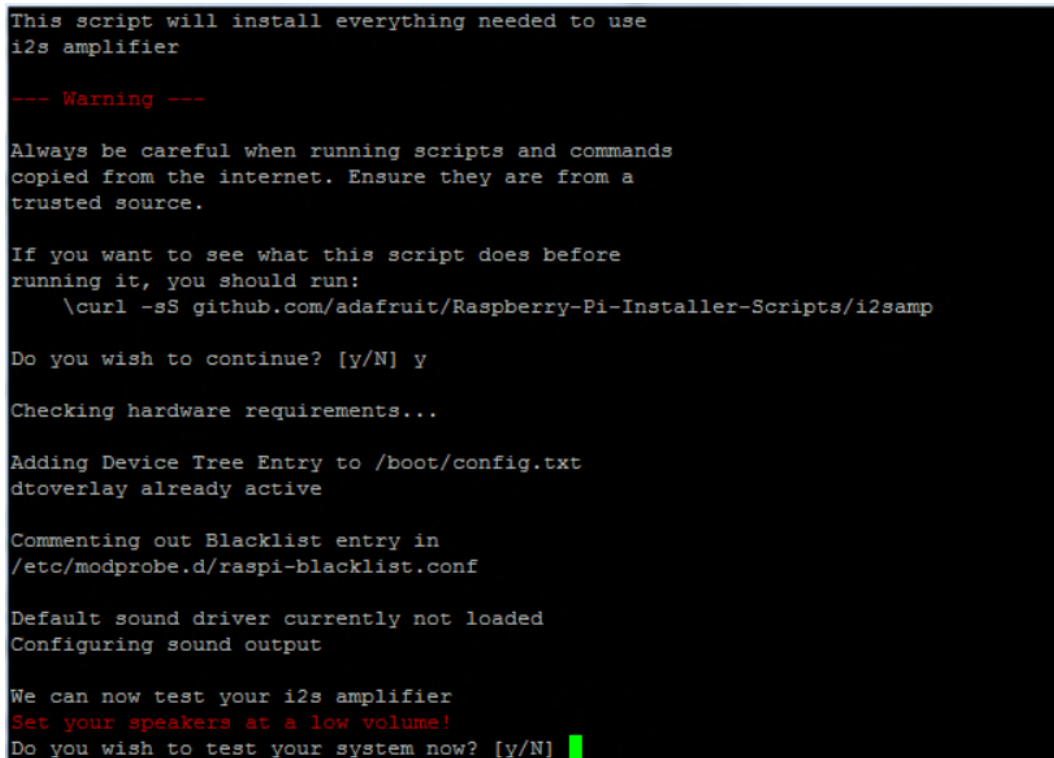
```
sudo apt install -y wget
pip3 install adafruit-python-shell
wget https://github.com/adafruit/
```

```
Raspberry-Pi-Installer-Scripts/raw/main/i2samp.py
sudo -E env PATH=$PATH python3 i2samp.py
```

3. Tras esto, nos aparecerá algo parecido a lo de la Figura A.2. Seleccionaremos 'y' en todo, hasta que nos diga si queremos reiniciar, lo cual también aceptaremos.
4. Una vez reiniciado, debemos ir a editar el fichero `/boot/config.txt` y buscar una línea que ponga `dtoverlay=audio=on`, la cual comentaremos escribiendo un `#` delante. En la siguiente línea escribiremos:

```
dtoverlay=hifiberry-dac
```

5. Guardaremos el archivo pulsando `Ctrl + X` y reiniciaremos la Raspberry de nuevo.



```
This script will install everything needed to use
i2s amplifier

--- Warning ---

Always be careful when running scripts and commands
copied from the internet. Ensure they are from a
trusted source.

If you want to see what this script does before
running it, you should run:
  \curl -sS github.com/adafruit/Raspberry-Pi-Installer-Scripts/i2samp

Do you wish to continue? [y/N] y

Checking hardware requirements...

Adding Device Tree Entry to /boot/config.txt
dtoverlay already active

Commenting out Blacklist entry in
/etc/modprobe.d/raspi-blacklist.conf

Default sound driver currently not loaded
Configuring sound output

We can now test your i2s amplifier
Set your speakers at a low volume!
Do you wish to test your system now? [y/N]
```

Figura A.2: Salida del *script* de instalación para manejar el altavoz

Cuando hayamos acabado, para poder probarlo, usaremos la herramienta *mpg123*, que se puede instalar del siguiente modo:

```
sudo apt-get install -y mpg123
```

Luego, podremos escuchar cualquier archivo mp3 *stereo* mediante esta herramienta o poniendo la URL de una radio de internet, como la siguiente:

```
mpg123 http://ice1.somafm.com/u80s-128-mp3
```

Esta instalación se basa en la explicación de Adafruit que se puede ver [aquí](#). Si no se consiguiera el funcionamiento correcto, probar la instalación manual que se indica en la misma página.

Nota: Tener en cuenta que ahora mismo solo se soporta audio *stereo*.

Instalación de OpenCV para el reconocimiento de objetos

Antes de comenzar la instalación se debe cambiar una línea de configuración en el archivo *Swapfile*, para ello se escribe en el terminal:

```
sudo nano /etc/dphys-swapfile
```

Cambiaremos *CONF_SWAPSIZE* = 100 a *CONF_SWAPSIZE* = 2048. Guardaremos los cambios presionando Ctrl+X, luego Y, y finalmente Enter. Este cambio es temporal y deberá ser revertido después.

Tras esto, continuaremos instalando las dependencias necesarias para OpenCV:

```
sudo apt-get install build-essential cmake pkg-config
sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev
sudo apt-get install libxvidcore-dev libx264-dev
sudo apt-get install libgtk2.0-dev libgtk-3-dev
sudo apt-get install libatlas-base-dev gfortran
sudo pip3 install numpy
```

Después, descargaremos y descomprimiremos OpenCV y los módulos contribuidos (segunda y tercera línea son la misma):

```
wget -O opencv.zip https://github.com/opencv/opencv/archive/4.4.0.zip
wget -O opencv_contrib.zip
https://github.com/opencv/opencv_contrib/archive/4.4.0.zip
unzip opencv.zip
unzip opencv_contrib.zip
```

Ahora, crearemos un directorio de compilación y configuraremos el proyecto:

```
cd ~/opencv-4.4.0/
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=RELEASE \
      -D CMAKE_INSTALL_PREFIX=/usr/local \
      -D INSTALL_PYTHON_EXAMPLES=ON \
      -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib-4.4.0/modules \
      -D BUILD_EXAMPLES=ON ..
```

Cuando acabemos, compilaremos OpenCV:

```
make -j $(nproc)
```

Importante: Esta parte puede tardar varias horas, dependiendo del modelo de Raspberry utilizado. Para la Pi Zero 2W puede tardar más de un día. Si se congela o da error, simplemente volvemos a escribir el comando anterior y continuará desde donde se quedó.

Cuando finalice, instalaremos OpenCV:

```
sudo make install && sudo ldconfig
sudo reboot
```

Después de reiniciar, el proceso de instalación estará casi completo. Ahora, se puede cambiar el archivo de intercambio de nuevo a `CONF_SWAPSIZE = 100`.

Finalmente, podemos descargar el código de ejemplo para realizar el reconocimiento de objetos y la librería COCO entrenada en el siguiente [enlace](#).

Nota: Debido a que no había ninguna guía para la instalación de OpenCV en nuestro modelo de Raspberry, se ha empleado la misma que desarrolló a modo de investigación Rafael Rodríguez Ortiz para su TFG, que usaba nuestro mismo modelo. Este TFG era el trabajo que tomamos de base para el subsistema de reconocimiento de objetos. A su vez, él se basó para la instalación en esta [guía](#).

Uso de varias UART virtuales en la Raspberry

Para poder usar la UART de la Raspberry (RX que tenemos conectado a los TX de los microcontroladores para poder imprimir por consola de la Raspberry lo que ellos mandan a la UART), debemos hacer una serie de configuraciones:

1. Activar la UART escribiendo el comando

```
sudo nano /boot/config.txt
```

buscando la línea o escribiéndola si no existe: `enable_uart=1`.

2. Deshabilitar la UART de la Raspberry como consola de login del sistema:

a) Editar `/boot/cmdline.txt` con

```
sudo nano /boot/cmdline.txt
```

y buscar y eliminar la parte donde dice `console=serial0,115200`. Notar que este archivo debe estar escrito en una sola línea.

b) Desactivar el servicio `serial-getty` usando:

```
sudo systemctl disable serial-getty@serial0.service
sudo systemctl stop serial-getty@serial0.service
```

3. Reiniciar la Raspberry usando:

```
sudo reboot
```

El paso 2. es necesario ya que por defecto la Raspberry usa la UART como consola de *login*, lo que permite acceder a la Raspberry por medio de la UART, como otra alternativa a SSH o a una pantalla con HDMI.

En el caso en el que se quiera activar de nuevo la UART como consola de *login* es necesario hacer los pasos opuestos:

1. Acceder a `/boot/cmdline.txt` y añadir donde estaba antes lo siguiente *console=serial0,115200*, quedando todo en una sola línea, sin saltos. Debe quedar así (en una sola línea):

```
dwc\otg.lpm\_enable=0 console=serial0,115200 console=tty1
root=PARTUUID=61170772-02 rootfstype=ext4 elevator=deadline
fsck.repair=yes rootwait quiet splash plymouth.ignore-serial-
consoles cfg80211.ieee80211\_regdom=GB
```

2. Volver a habilitar el servicio *serial-getty*:

```
sudo systemctl enable serial-getty@serial0.service
sudo systemctl start serial-getty@serial0.service
```

3. Reiniciar la Raspberry.

A continuación, lo que haremos será instalar la biblioteca *pigpio*, que nos permitirá leer la UART de pines que lo permitan. Para ello:

1. Instalar *pigpio*:

```
sudo apt update
sudo apt install pigpio python3-pigpio
```

2. Iniciar el servicio *pigpio* cada vez que queramos usar un programa Python para leer la UART:

```
sudo pigpiod
```

Ahora ya podemos ejecutar un código Python estableciendo un pin compatible con RX como nuestro RX de la UART.

Importante recalcar que esta UART es por software y no funciona tan bien como usar la UART normal, pero nos permite emplear cualquier pin de la Raspberry compatible como RX, permitiéndonos tener varias “UART virtuales” al mismo tiempo. La velocidad de recepción debe ser de 9600 baudios y no más alta para que la recepción sea correcta.

Se puede ver un código de ejemplo de uso en la Figura [A.3](#).

Instalación y uso de OpenOCD para la programación de microcontroladores desde la Raspberry

OpenOCD es la herramienta que nos permitirá programar microcontroladores desde la Raspberry, usando el fichero binario generado al compilar el programa, sin necesidad de programadores externos. Se recuerda que debemos tener abierta la terminal en la Raspberry Pi y tener acceso a Internet.

Los pasos son los siguientes:

```

import pigpio
import time

RX_GPIO = 17
BAUDRATE = 9600

pi = pigpio.pi() # Conecta al daemon
if not pi.connected:
    exit(1)

# Abre UART software en GPIO17
status = pi.bb_serial_read_open(RX_GPIO, BAUDRATE)
if status != 0:
    print("Error al abrir UART por software.Codigo:", status)
    pi.stop()
    exit(1)

print(f"Escuchando en GPIO {RX_GPIO} a {BAUDRATE} bps")

try:
    while True:
        count, data = pi.bb_serial_read(RX_GPIO)
        if count > 0:
            print("Recibido", data.decode('utf-8', errors='ignore'))
            time.sleep(0.1)
except KeyboardInterrupt:
    print("Interrumpido por el usuario")

# Limpieza
finally:
    pi.bb_serial_read_close(RX_GPIO)
    pi.stop()

```

Figura A.3: Programa de ejemplo para leer la UART de la Raspberry

1. Comenzaremos haciendo una actualización para tener los últimos paquetes y repositorios configurados:

```
sudo apt-get update
```

2. Ahora instalaremos todas las herramientas necesarias para compilar OpenOCD (una línea):

```
sudo apt-get install git autoconf libtool make pkg-config  
libusb-1.0-0 libusb-1.0-0-dev libjim-dev
```

3. A continuación, descargaremos el último código fuente de OpenOCD con:

```
git clone http://openocd.zylin.com/openocd
```

4. Seguidamente, cambiaremos al directorio del código y ejecutaremos el programa previo con:

```
cd openocd  
./bootstrap
```

5. Lo siguiente será compilar OpenOCD con el soporte nativo de *GPIO twiddling* de Raspberry Pi:

```
./configure --enable-sysfsgpio --enable-bcm2835gpio
```

6. Tras esta configuración es cuando haremos la compilación con:

```
make
```

7. Si la compilación tiene éxito, instalaremos con:

```
sudo make install
```

Esto sería todo para la instalación.

Veamos ahora los pasos para realizar la programación:

1. La forma más fácil es comenzar creando un nuevo directorio en el directorio *home*, por ejemplo llamándolo *programs*. Dentro de este directorio será donde pongamos los ficheros binarios o hexadecimales con los que queramos programar los microcontroladores:

```
cd ~  
mkdir programs  
cd programs
```

2. En este directorio, crearemos un nuevo fichero llamado *openocd.cfg*:

```
nano openocd.cfg
```

3. En este fichero escribiremos lo siguiente (esto es un ejemplo para programar el microcontrolador que controla los motores de los dedos):

```
source [find interface/raspberrypi-native.cfg]
transport select swd

source [find target/stm32l4x.cfg]
#source [find target/stm32g4x.cfg]

reset_config srst_nogate

# Pins uMot
adapter gpio srst 24
adapter gpio swclk 18
adapter gpio swdio 23

init
targets
reset halt

# Programar el uMot
program MotorControlMicro.bin 0x08000000 verify reset

shutdown
```

4. Tras esto, guardaremos los cambios y en el directorio ejecutaremos la siguiente orden que realizará la programación:

```
sudo openocd
```

La salida debería ser parecida a la obtenida en la Figura [A.4](#).

En el fichero *openocd.cfg* hay varias partes interesantes de comentar:

- En primer lugar, la línea *source [find target/stm32l4x.cfg]*. Aquí es donde debemos poner el microcontrolador que vamos a programar. Para el microcontrolador de los motores, al ser un STM32L496RGT6, es *stm32l4x.cfg*. Si fuera el de reconocimiento de voz, se cambiaría por la línea de abajo, que es *stm32g4x.cfg* al ser un STM32G474RET6. Los nombres que le asigna OpenOCD a los distintos microcontroladores (no solo de STM) o placas de desarrollo están en el fichero */usr/local/share/openocd/scripts/target*
- Lo siguiente importante son los pines de la Raspberry que utilizamos para programar. Deben ser 3 y deben estar conectados a sus homólogos en el microcontrolador. Necesitamos en la Raspberry un pin de reset (cualquier GPIO), un pin de datos (SWDIO) y un pin de señal de reloj (SWCLK), que son válidos el GPIO

18 (usado por nosotros), el 11, 12, 13 y 19.

- Finalmente, en la línea:

```
program MotorControlMicro.bin 0x08000000 verify reset
```

se debe escribir el nombre del fichero binario obtenido al compilar nuestro programa y con el que queremos programar el microcontrolador. En nuestro caso es *MotorControlMicro.bin*. Cabe destacar que este fichero binario debe haber sido obtenido tras realizar la compilación con el IDE de STM. El binario que puede generar el IDE de Arduino con STM32duino no será válido.

```
pi@raspberrypi:~/programs $ sudo openocd
Open On-Chip Debugger 0.12.0+dev-00801-g133dd9d66 (2024-12-08-03:20)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Warn : TMS/SWDIO moved to GPIO 8 (pin 24). Check the wiring please!
Info : BCM2835 GPIO JTAG/SWD bitbang driver
Info : clock speed 499 kHz
Info : SWD DPIDR 0x2ba01477
Info : [stm32l4x.cpu] Cortex-M4 r0p1 processor detected
Info : [stm32l4x.cpu] target has 6 breakpoints, 4 watchpoints
Info : [stm32l4x.cpu] Examination succeed
Info : [stm32l4x.cpu] starting gdb server on 3333
Info : Listening on port 3333 for gdb connections
[stm32l4x.cpu] halted due to breakpoint, current mode: Thread
xPSR: 0x01000000 pc: 0x080006ec msp: 0x20040000
[stm32l4x.cpu] halted due to breakpoint, current mode: Thread
xPSR: 0x01000000 pc: 0x080006ec msp: 0x20040000
** Programming Started **
Info : device idcode = 0x20006461 (STM32L49/L4Axx - Rev B : 0x2000)
Info : RDP level 0 (0xAA)
Info : flash size = 1024 KiB
Info : flash mode : dual-bank
Info : Padding image section 0 at 0x08004104 with 4 bytes (bank write end alignment)
Warn : Adding extra erase range, 0x08004108 .. 0x080047ff
** Programming Finished **
** Verify Started **
** Verified OK **
** Resetting Target **
shutdown command invoked
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
pi@raspberrypi:~/programs $ cd
```

Figura A.4: Salida obtenida al programar correctamente un microcontrolador con OpenOCD

Nota: Los pasos para la instalación y uso de OpenOCD los hemos obtenido del siguiente [enlace](#). Consultar para más información.

A.2.3. Software relacionado con los microcontroladores STM

En esta sección se hablará de las herramientas *software* empleadas para realizar los programas que ejecutan los microcontroladores STM32 de la prótesis.

STM32CubeIDE

Este es el IDE usado para crear los programas en C y C++ de cada microcontrolador. Lo podemos descargar desde [aquí](#).

Si se quiere importar cada uno de los proyectos de la prótesis en un ordenador personal, habría que seguir los siguientes pasos:

1. Ir a la sección *File > Import... > Existing Projects into Workspace*
2. Buscaremos el proyecto y lo importaremos, seleccionando la opción de *Copy projects into workspace* para no modificar el original.

Si ahora lo que queremos es generar el binario o hexadecimal para utilizarlo con OpenOCD y programar el microcontrolador desde la Raspberry, deberemos ir a *Project > Properties > C/C++ Build > Settings > Tool Settings > MCU/MPU Post build outputs* y seleccionaremos:

- *Convert to binary file (-O binary)*
- *Convert to Intel Hex file (-O ihex)*

Tras esto, aplicaremos los cambios y ya podremos compilar. Estos ficheros se encontrarán en el directorio *Debug* del proyecto.

STM32CubeProgrammer

Este es un programa alternativo que se puede emplear para programar los microcontroladores, pero también cambiarles la configuración con los *Option Bytes*. Cabe destacar que esto solo será necesario hacerlo la primera vez para programar microcontroladores nuevos de fábrica para el caso de que el pin *Boot0* se deje al aire. Para todos los microcontroladores que hemos usado en el proyecto ya se ha realizado, así que no hace falta hacerlo de nuevo. Se incluye a nivel curiosidad, ya que es algo que nos costó averiguar.

Lo primero que se debe hacer es usar el ST-LINK y conectarlo a los pines del microcontrolador a través de la interfaz SWD.

Luego, en el STM32CubeProgrammer, nos iremos a la opción de *Option Bytes* y debemos:

- Marcar la opción nBOOT1 y nBOOT0.
- Desmarcar nSWBOOT0.

Ya se podrá programar el microcontrolador fácilmente.

Obtener dataset para entrenamiento del subsistema de sensado EMG con STM32CubeMonitor

Aunque este programa se puede usar para muchas cosas, nosotros lo empleamos para desarrollar un flujo que comienza el proceso de muestreo y guarda las característi-

cas directamente en un .csv en el formato correcto, que formará el *dataset* para el *script* de entrenamiento de la red neuronal que se usará en el subsistema de sensado EMG. La explicación que se dará aquí será sencilla y simplemente para enseñar a utilizarla para formar un nuevo *dataset*. Para más información, recomendamos encarecidamente revisar el Trabajo de Fin de Máster de Henrik Plastrotmann y Benedikt Wülker, donde se comentan detalles importantes.

Para tomar datos, primero se debe tener colocado el brazalete EMG en el antebrazo, con el microcontrolador principal conectado a un ST-LINK. Luego, para abrir el flujo, abriremos el programa STM32CubeMonitor(descargable [aquí](#)) e importaremos el flujo llamado *EMG_DataAcquisitionFlow.json*. Tras esto, podremos darle al botón *Deploy* y veremos el *Dashboard* de la Figura 4.5.

Los pasos a realizar a continuación son los siguientes:

1. Empezar la adquisición pulsando el botón **START ACQUISITION**.
2. Seleccionar la clase de la que vamos a adquirir datos en el menú desplegable (*Fist*, *Flat*, *Thumb*, *Peace*). Aunque este menú puede mostrar una clase de una sesión anterior, es importante hacer clic en la clase deseada de todas maneras, al menos después de cerrar y reabrir o relanzar el *dashboard*. De otro modo, puede enseñar la clase, pero realmente no estará inicializada. Si no estás seguro, puedes comprobar el nodo del menú en el flujo. Si hay un número debajo, la clase está inicializada. Si no hay ninguno, deberás hacer clic en el menú del *dashboard* y seleccionar la clase. Esto añadirá la etiqueta en el .csv luego.
3. Poner con la mano el gesto que se haya seleccionado y mantenerlo.
4. Pulsar el botón **CAPTURE EMG SAMPLE**. El LED izquierdo se pondrá de color verde y algo de rxData (representando el estado actual de los cuatro amplificadores) se mostrará en el gráfico. Después de algunos segundos, el LED izquierdo se pondrá en color naranja y luego rojo. Finalmente, el LED derecho se pondrá verde, indicando que el proceso de muestreo se ha completado y los datos se pueden guardar.
5. Pulsa el botón **SAVE FEATURES**. Esto combinará los flujos de datos y la etiqueta de la clase en una línea del .csv y lo guardará en dicho archivo. De esta manera ya tenemos una muestra tomada de un gesto.
6. Si queremos seguir tomando muestras del mismo gesto, seguiremos teniendo en la mano el gesto seleccionado y seguiremos haciendo el proceso de capturar muestra y guardar.
7. Si queremos tomar una muestra de un gesto distinto, no hace falta parar la adquisición y comenzarla de nuevo. Simplemente cambiaremos el gesto en el menú desplegable, pondremos la mano con ese gesto haremos una captura y un guardado por tantas muestras como queramos tener en el .csv.

Nota: Si se quisieran añadir más amplificadores o tomar unos distintos al 1, 2, 6, 7, se deberían hacer ajustes al flujo.

Una vez que tengamos el *dataset*, podemos importar el cuaderno *.ipynb* en Google Colab. Este cuaderno contiene la implementación de la red neuronal y su entrenamiento

que, junto con el *dataset*, podemos entrenar y obtener el modelo *tflite* para importarlo en STM32Cube.AI e implementarlo en el microcontrolador.

STM32Cube.AI para implementar modelos de Deep Learning en microcontroladores STM32

Esta es una herramienta integrada en el STM32CubeIDE, que nos permitirá integrar modelos de redes neuronales en formatos ONNX, TFLITE, etc., en nuestro microcontrolador STM32. En concreto, lo hemos usado para el microcontrolador del subsistema de reconocimiento de voz y el subsistema de sensado EMG.

Para usarlo, iremos al fichero *.ioc* del proyecto y seleccionaremos la columna *Pinout & Configuration*. Seleccionaremos la categoría *Middleware and Software Packs* y buscaremos *X-CUBE-AI*. Añadiremos una nueva red neuronal seleccionando el símbolo *+* cerca de la pestaña *Main*. Asignaremos un nombre para la red neuronal y seleccionaremos el tipo de fichero en el que tenemos la red neuronal entrenada. En el apartado *model*, buscaremos el fichero del modelo en nuestro equipo. Después de ello, podremos clicar en el botón *Analyse*. El IDE analizará la red neuronal y, si es correcta, confirmará el modelo. Opcionalmente, se puede pulsar *Validate on desktop* después de analizar para que haga una prueba con nuestro equipo de la red neuronal. Finalmente, podremos generar el código y el IDE nos habrá generado ficheros de cabecera y fuentes con funciones que podremos emplear para ejecutar la red neuronal en nuestro programa. Todo esto se puede ver en la Figura A.5.

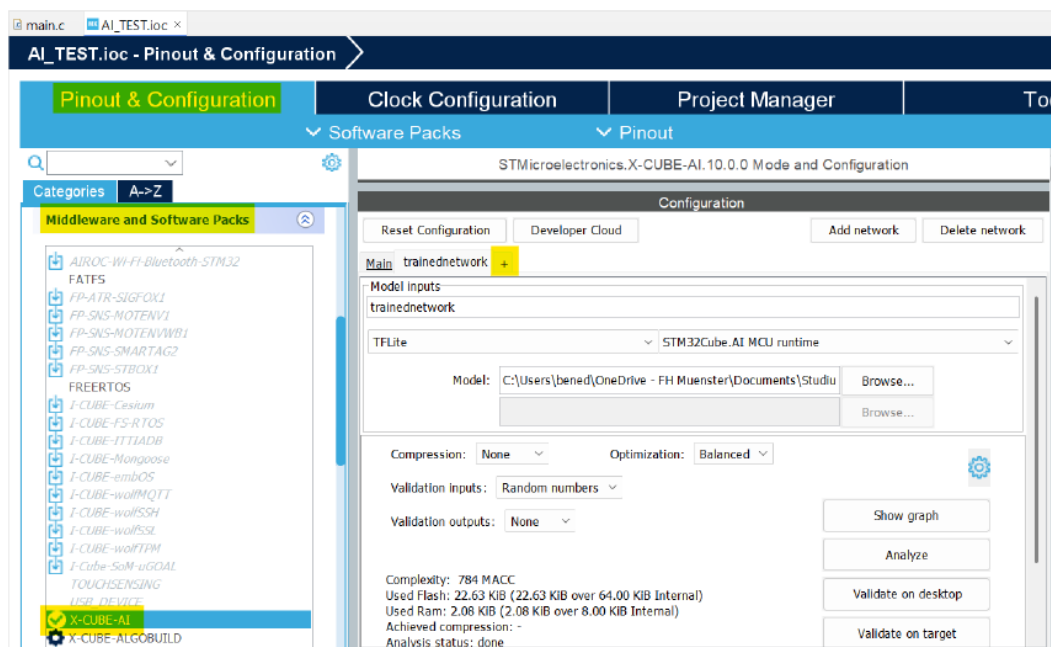


Figura A.5: STM32CubeIDE añadiendo una nueva red neuronal con STM32Cube.AI

B. Manual de usuario

Las instrucciones de uso del software se detallan a continuación. Este manual se dirige al usuario final del software objetivo de este proyecto.

B.1. Introducción

El software que se presenta está dirigido al control de los distintos elementos de una futura prótesis de mano robótica. Actualmente está desarrollado para un prototipo que no tiene dedos montados, pero que se puede adaptar de manera muy simple para el momento en que se incluyan. Una parte del mismo se encarga de controlar cada uno de los subsistemas individuales que son: motores que mueven los dedos, sensores que detectan tacto, cámara para reconocer objetos, micrófono para reconocer palabras, altavoz para reproducir audio y luces RGB; mientras que la otra orquesta el funcionamiento conjunto de todos ellos.

La idea es que pueda ser usado por un niño de manera sencilla y le aporte un funcionamiento parecido al de una mano real, con varios elementos lúdicos, como es el caso de los sonidos y las luces.

El ámbito es el Laboratorio D12, donde tenemos conexión a la red Celula.

B.2. Instalación

En esta sección se detallarán tanto los requisitos como los pasos necesarios para tener el sistema listo para su uso.

B.2.1. Requisitos

- **Hardware:** Lo principal es tener los elementos de la prótesis robótica conectados a la PCB. A continuación, si es la primera vez que vamos a usar el sistema, necesitaremos teclado, ratón y monitor mediante un *hub* que permita conectar estos periféricos a la Raspberry Pi del sistema, para acceder a ella y poderle configurar una red a la que conectarla. A partir de ahora, podremos seguir accediendo a ella para ejecutar el *software* de esta manera o, más cómodamente, de manera inalámbrica usando cualquier equipo que pueda conectarse a la misma red y accediendo a la Raspberry mediante *ssh*, sin necesidad de todos los periféricos anteriores. Para ello, deberemos habernos fijado en la IP que ha sido asignada a la misma.

- **Software:** Opcionalmente, tener una herramienta que permita conectarse a la *Raspberry* mediante *ssh*. En equipos con sistema operativo Windows 10 o Windows 11, viene instalado de forma predeterminada. También se pueden usar programas específicos como *Putty*, que puede ser descargado en este [enlace](#). El resto del *software* está preinstalado en la tarjeta SD de la Raspberry.

B.2.2. Procedimiento

1. Encender la PCB conectándola a 7,4 V, tras lo cual se encenderán 3 LEDs de la misma en color rojo (ahora mismo solo 2 de ellos, porque uno está fundido).
2. Esperar a que el LED de la Raspberry Pi quede fijo en verde.
3. Si nos vamos a conectar mediante *ssh* en Windows:
 - Pulsar las teclas Windows + R.
 - Escribir cmd.
 - Pulsar Enter.

Ahora, en la consola escribiremos:

```
ssh pi@ip_de_la_Raspberry
```

En el momento en el que se nos pida la contraseña, escribiremos *raspberry*.

4. Una vez estemos conectados, nos moveremos al directorio *Codigos_mano_Praxedes* del siguiente modo:

```
cd Codigos_mano_Praxedes
```

5. Ahora podremos ejecutar el programa que pone en marcha el sistema escribiendo:

```
sudo -E python3 controlV3_1.py
```

6. Si todo ha funcionado correctamente, habremos visto que los LED se han encendido en color blanco y luego en rojo. Tras esto, podremos usar el sistema.
7. En el momento en el que se quiera parar la ejecución del programa, podremos pulsar las teclas Ctrl + C. Si se quiere apagar el sistema, podremos escribir:

```
sudo shutdown -h now
```

y cuando el Led de la Raspberry se apague, podremos desconectar el sistema de su alimentación.

B.3. Uso del sistema

El uso del sistema es bastante simple, contando con 3 modos que describiremos a continuación, controlados por los 3 botones que podemos ver en la Figura B.1.



Figura B.1: Botonera con 3 botones, cada uno para activar un modo

B.3.1. Modo Voz

Para usar este modo, debemos pulsar el botón que lleva su nombre. En este momento, si estábamos en otro modo, se cambiará el color de los LEDs a azul y sonará una voz diciendo que se ha activado este modo.

Poco después, se encenderá un LED naranja en la PCB durante unos segundos. Durante este tiempo, deberemos decir un número del 1 al 5 para que los motores se muevan de una forma u otra. Si el sistema nos ha entendido, moverá los motores para abrir la mano y, posteriormente, moverá cada uno a la posición identificada con el número. Si no nos ha entendido o la posición es la misma que la que ya teníamos, los motores no se moverán.

B.3.2. Modo Cámara

Su uso se activa en este caso, pulsando el botón central. Si estábamos en otro modo, se cambiará el color de los LEDs a verde y sonará una voz diciendo que se ha activado.

Para que el reconocimiento de un objeto se haga correctamente, debemos situar la cámara mirándolo antes de pulsar el botón. Los objetos disponibles son: persona, taza, pantalla, silla, tijeras, botella, ratón y teclado.

Al igual que antes, si se ha reconocido correctamente el objeto, se moverán los motores en consecuencia. En el caso opuesto o si la posición que designa ese objeto es igual a la que ya tenía, no se hará ningún movimiento.

B.3.3. Modo Mano

Para usar este modo, debemos habernos colocado previamente en el antebrazo izquierdo el brazalete EMG. Tras esto, podremos pulsar el botón llamado *Modo Mano*, el cual cambiará el color de los LED a rojo y también sonará una voz avisando del cambio de modo.

Para que el reconocimiento de un gesto se haga correctamente, debemos poner en la mano el gesto antes de pulsar el botón. Los gestos disponibles son: puño cerrado, mano abierta, pulgar y símbolo de la paz.

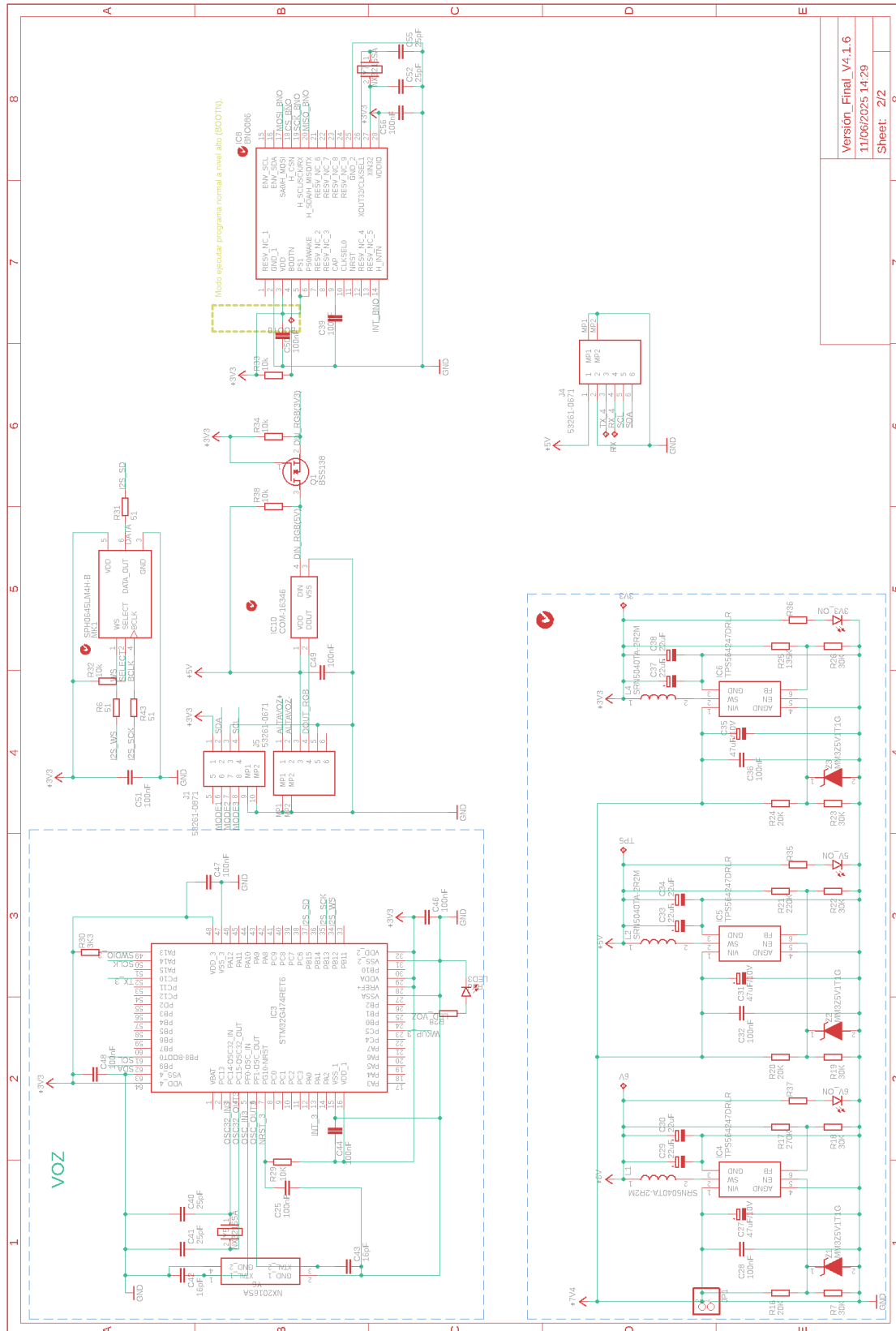
Del mismo modo que en los casos anteriores, si se ha reconocido correctamente el gesto, se moverán los motores en consecuencia. En el caso opuesto o si la posición que designa ese gesto es igual a la que ya tenía, no se hará ningún movimiento.

Nota: Para todos los modos, si queremos que se haga un nuevo reconocimiento, simplemente debemos pulsar su botón de nuevo.

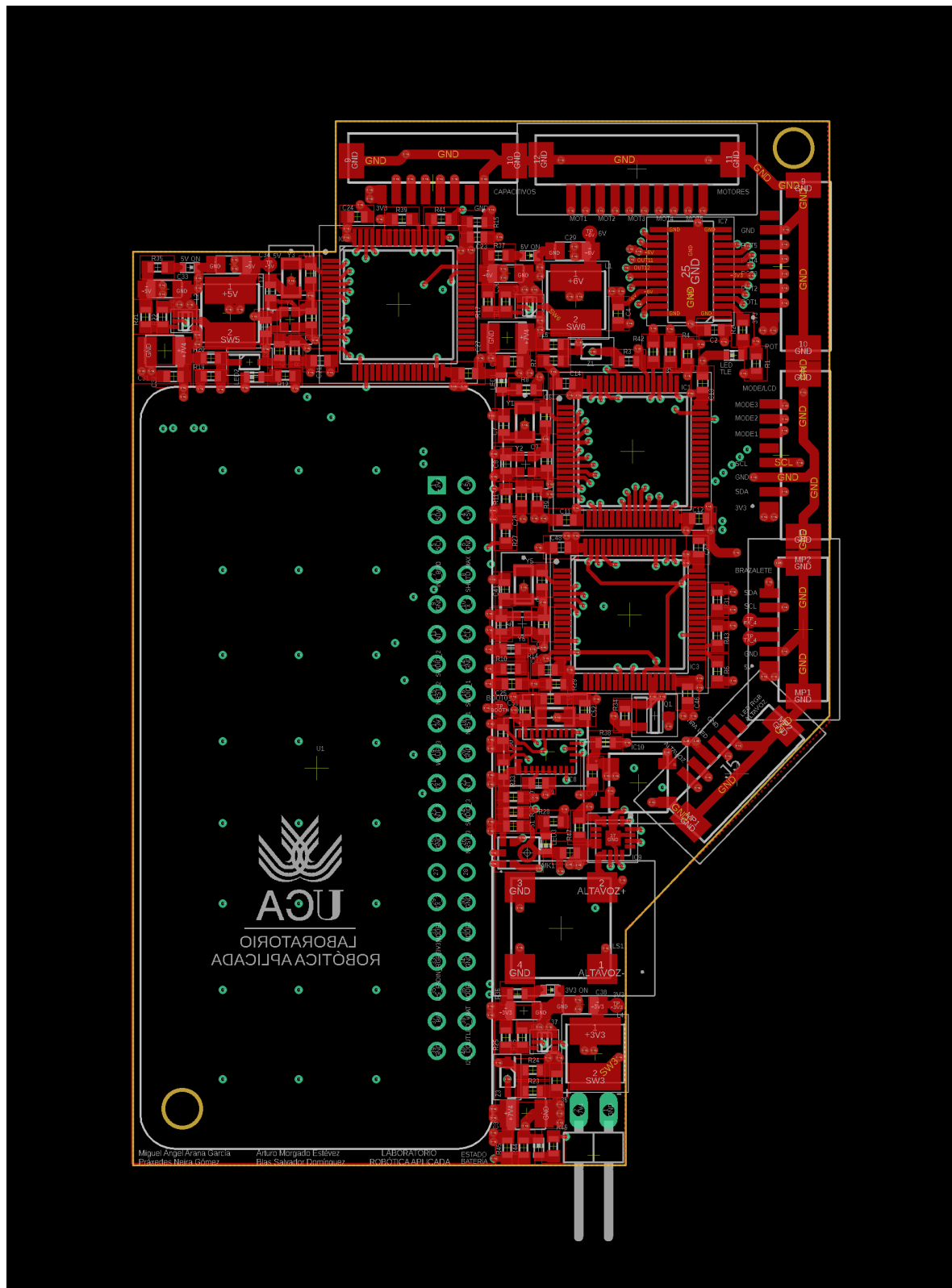
B.3.4. Tacto

Independientemente del modo en el que nos encontremos, el sensado de tacto estará activado. De esta manera, si durante el movimiento de un motor (que no sea de apertura de la mano, sino de cierre), se produce contacto con su sensor asociado, dicho motor se parará, para producir un agarre adaptativo. Además, dependiendo del sensor de tacto tocado, se activará un motor vibrador u otro, para informar que se está produciendo tacto.

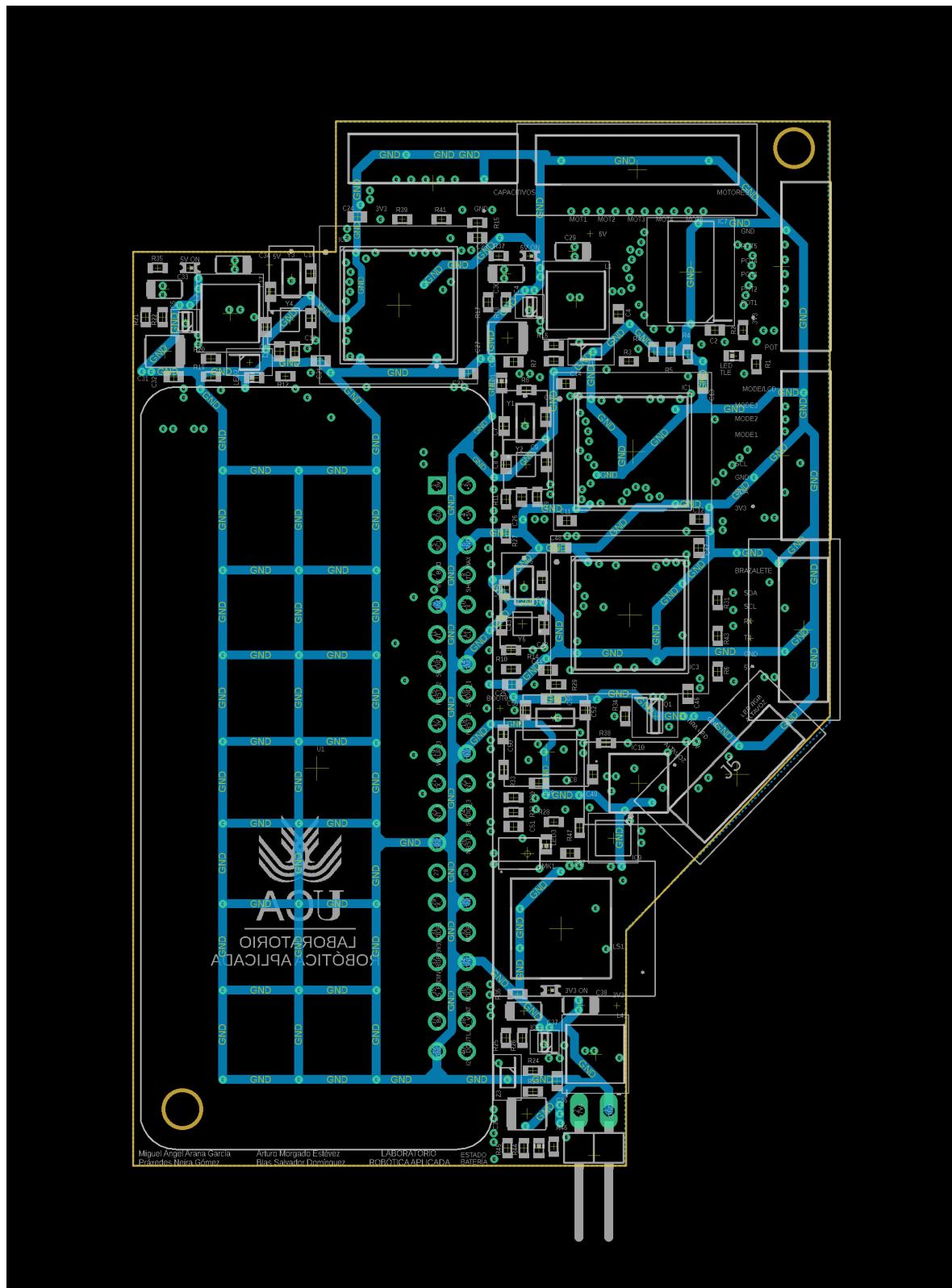
C.1.2. Esquemático 2 PCB principal



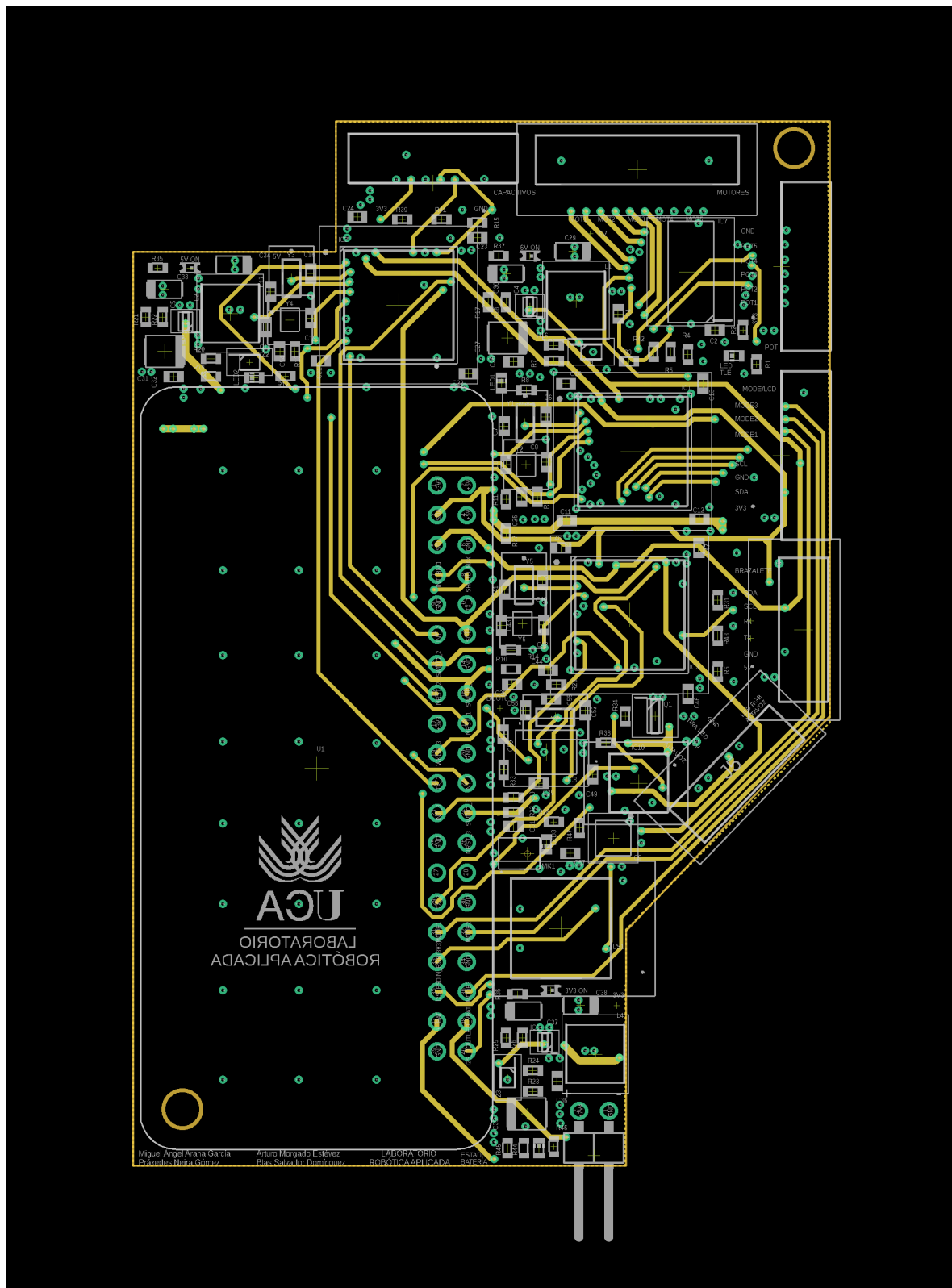
C.1.3. Board Top PCB principal



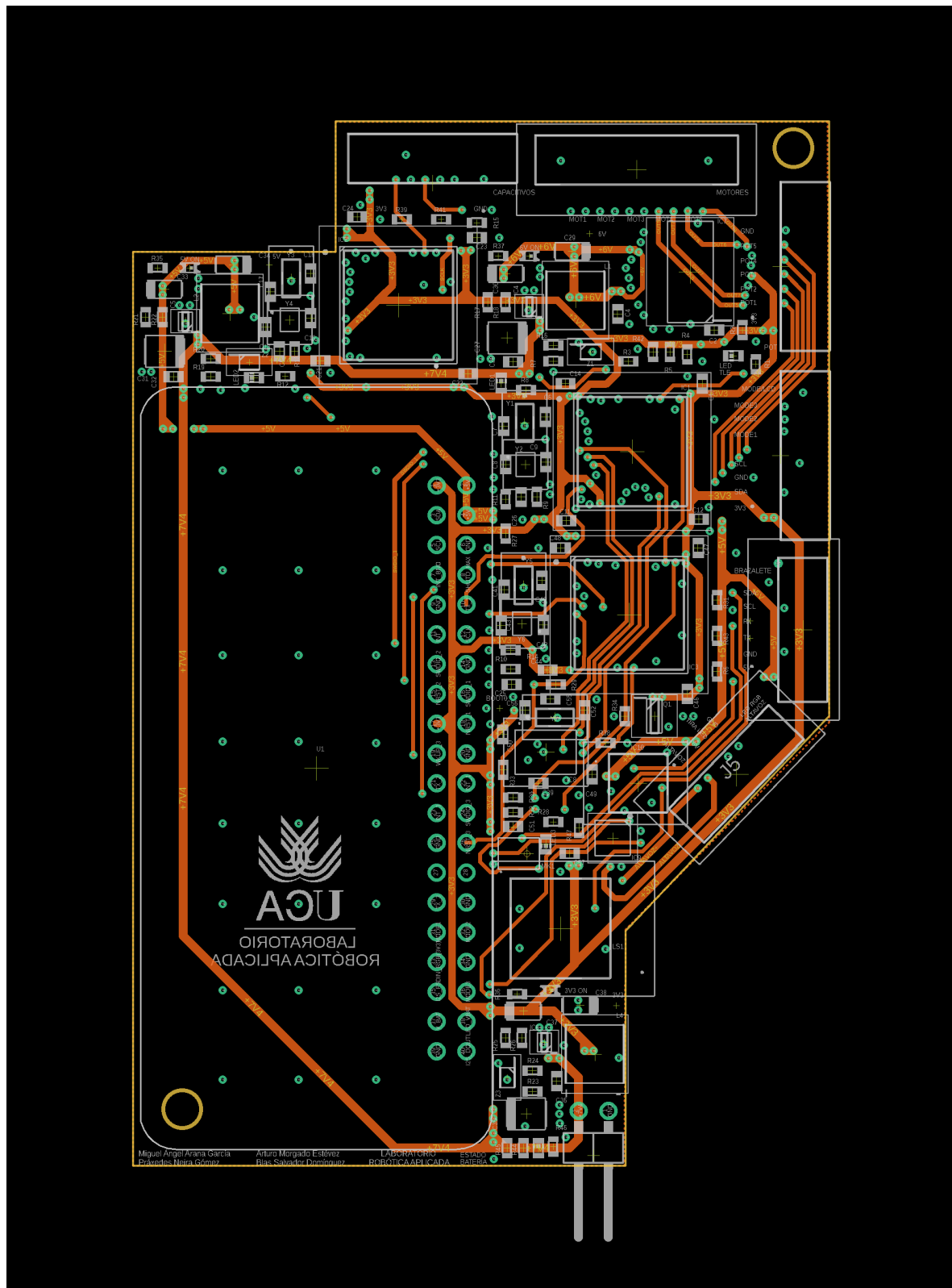
C.1.4. Board Bottom PCB principal



C.1.5. Board Capa 1 PCB principal

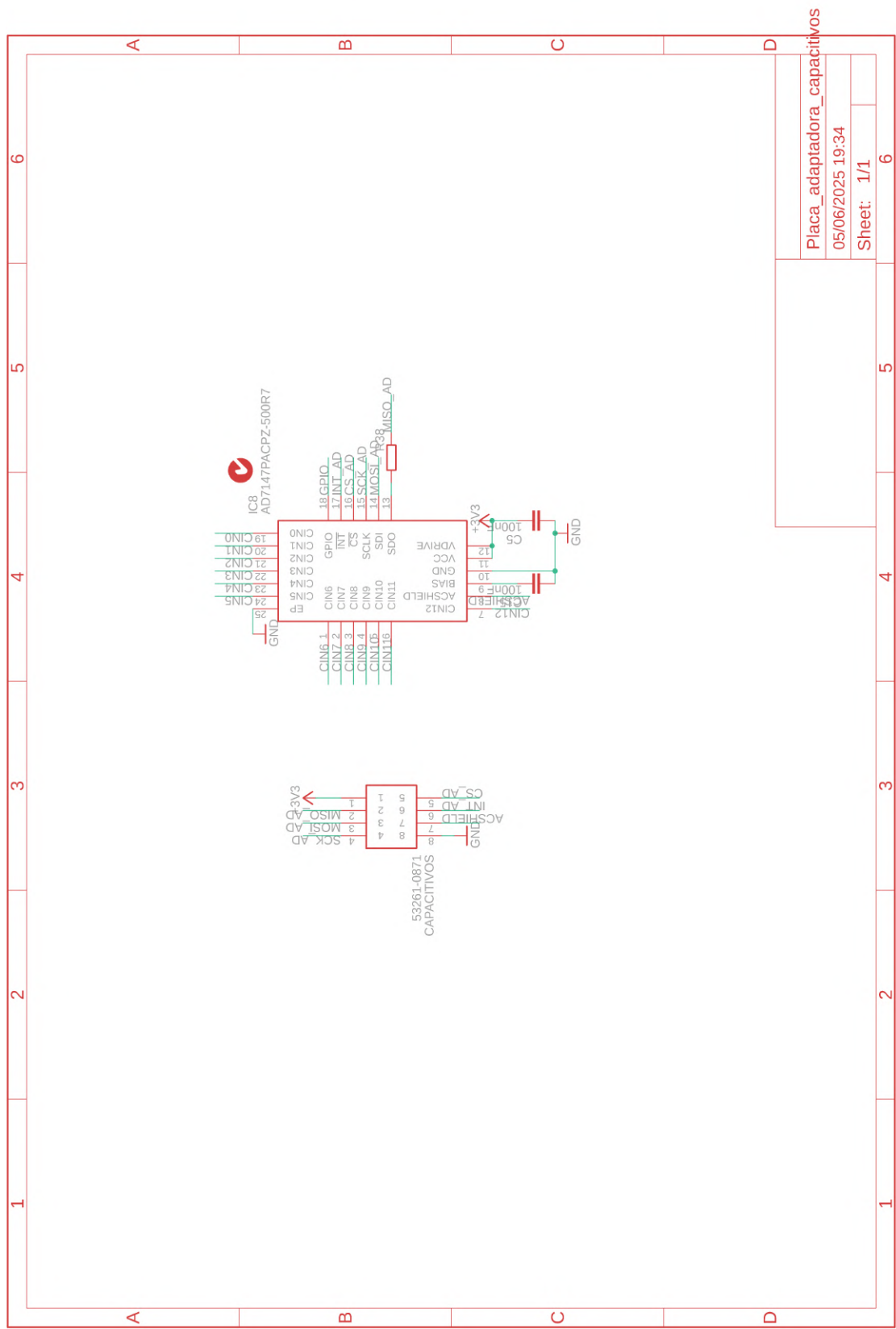


C.1.6. Board Capa 2 PCB principal

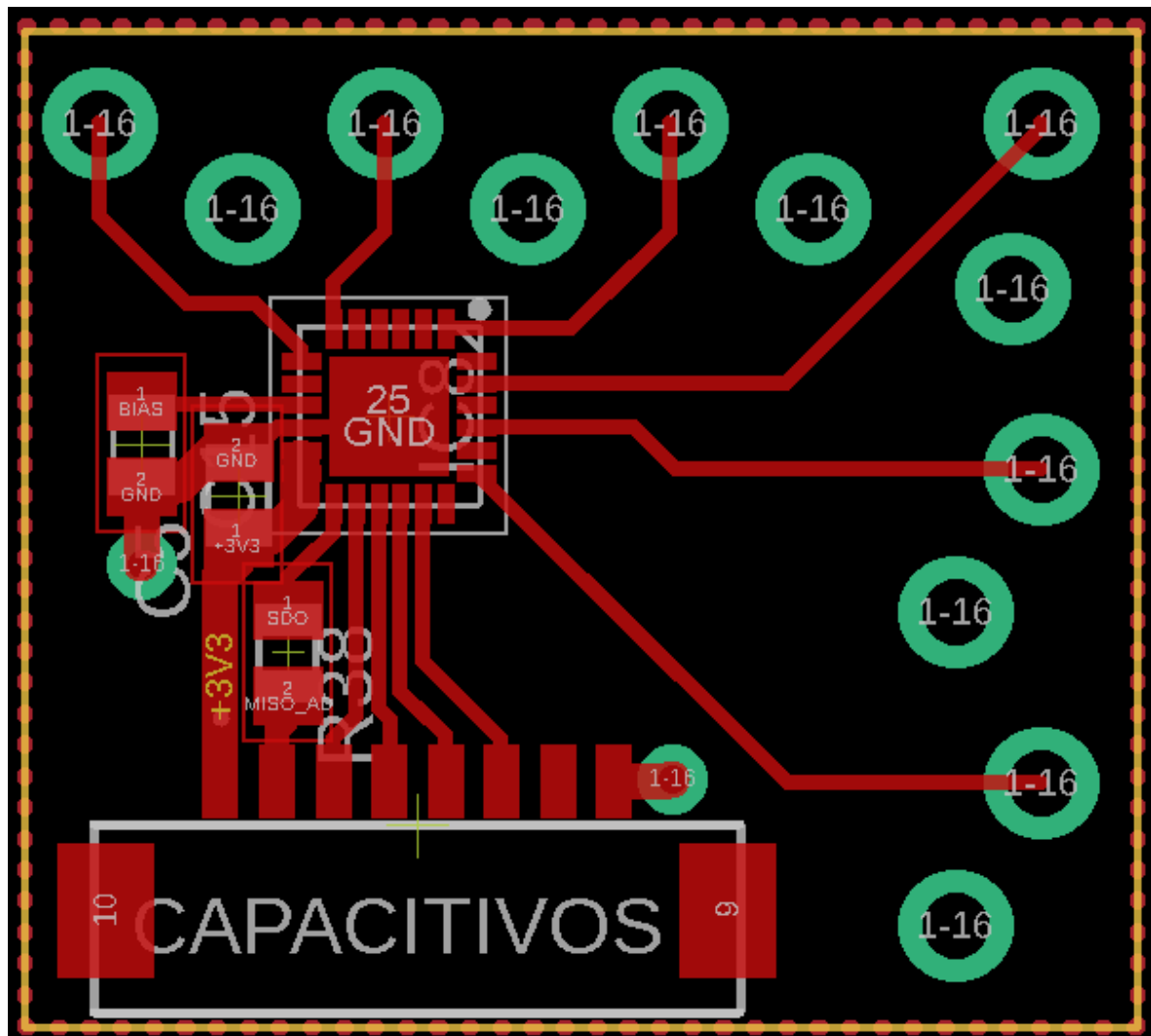


C.2. PCB IC sensores capacitivos

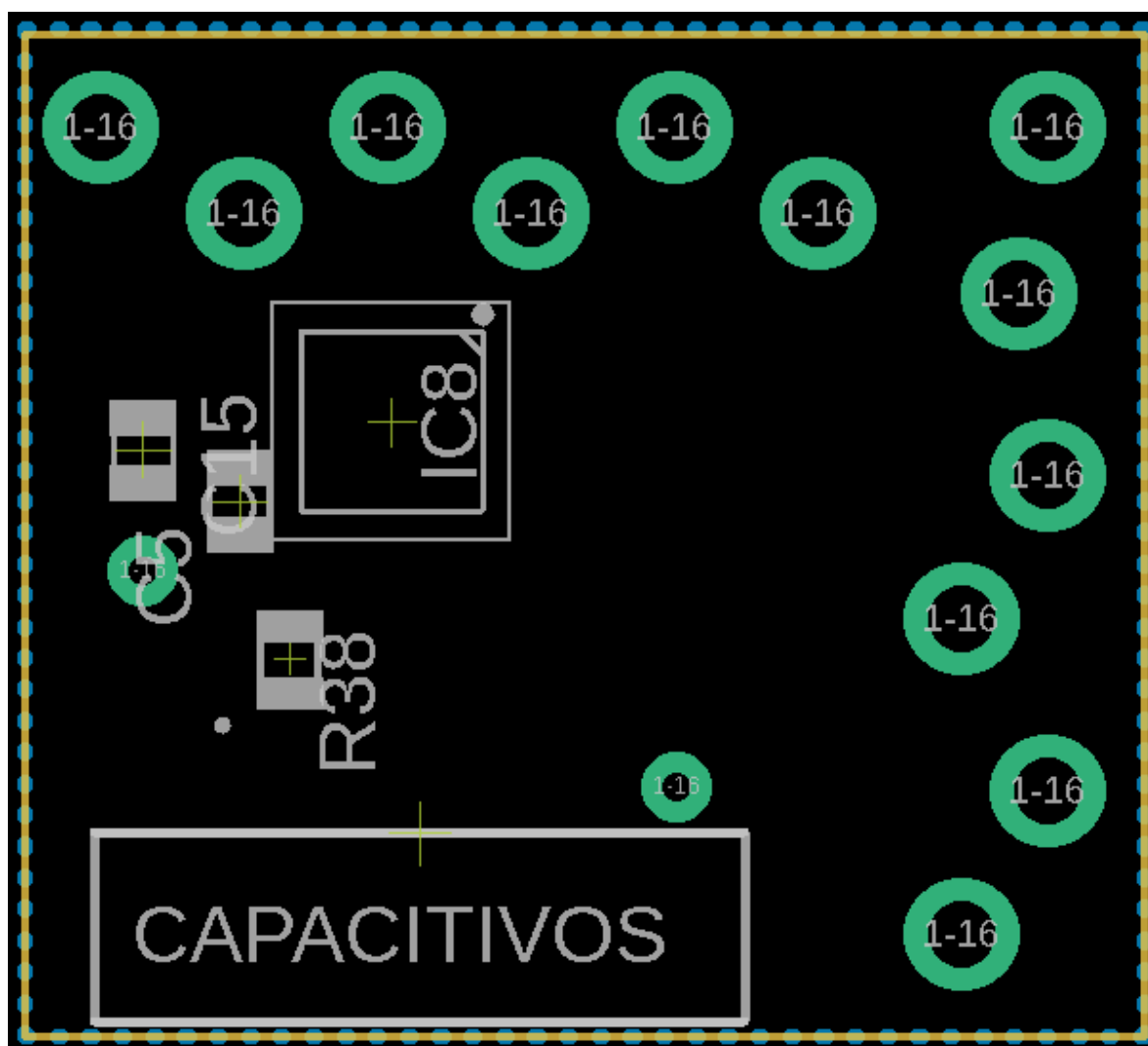
C.2.1. Esquemático PCB sensores capacitivos



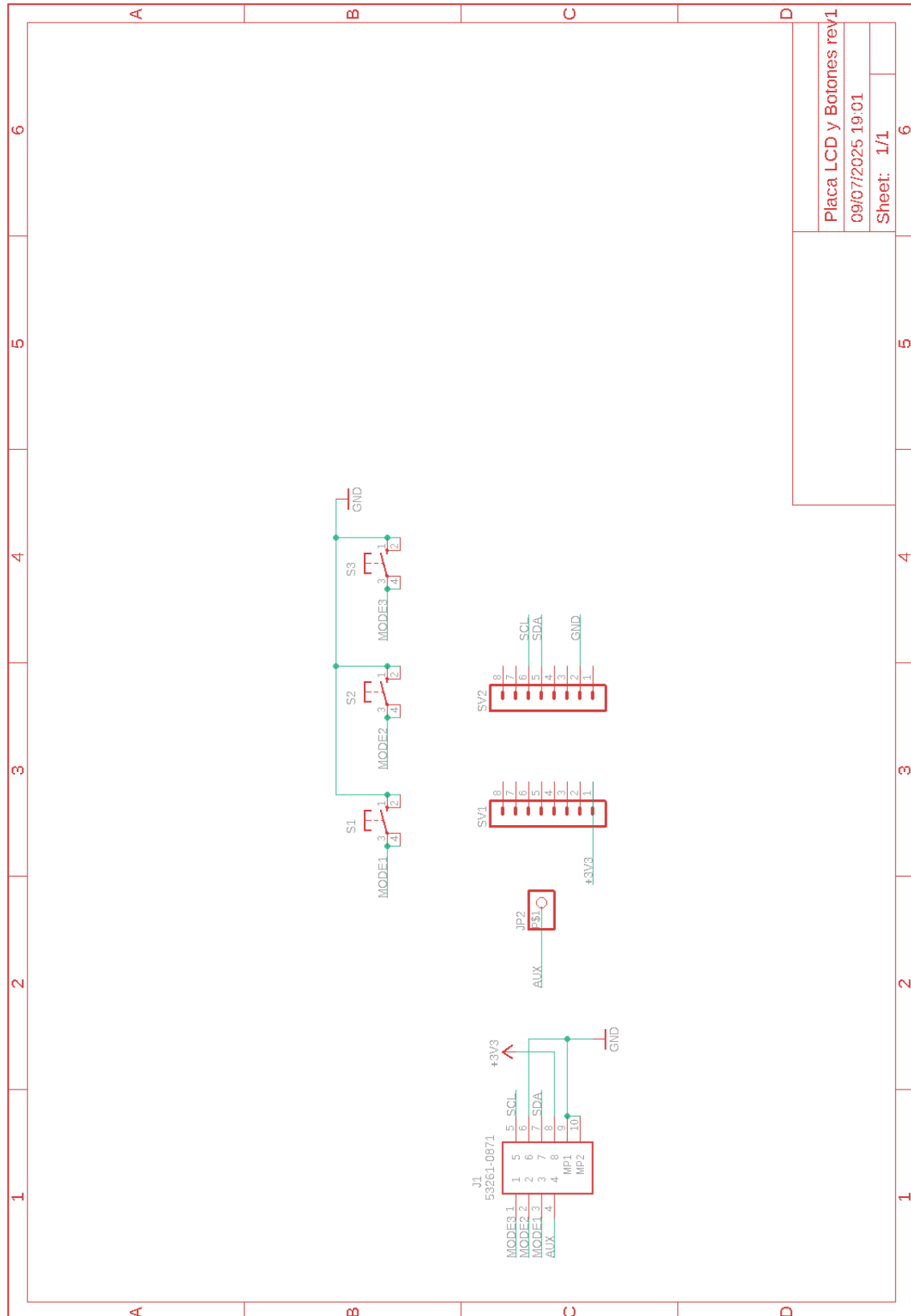
C.2.2. Board Top PCB capacitivos



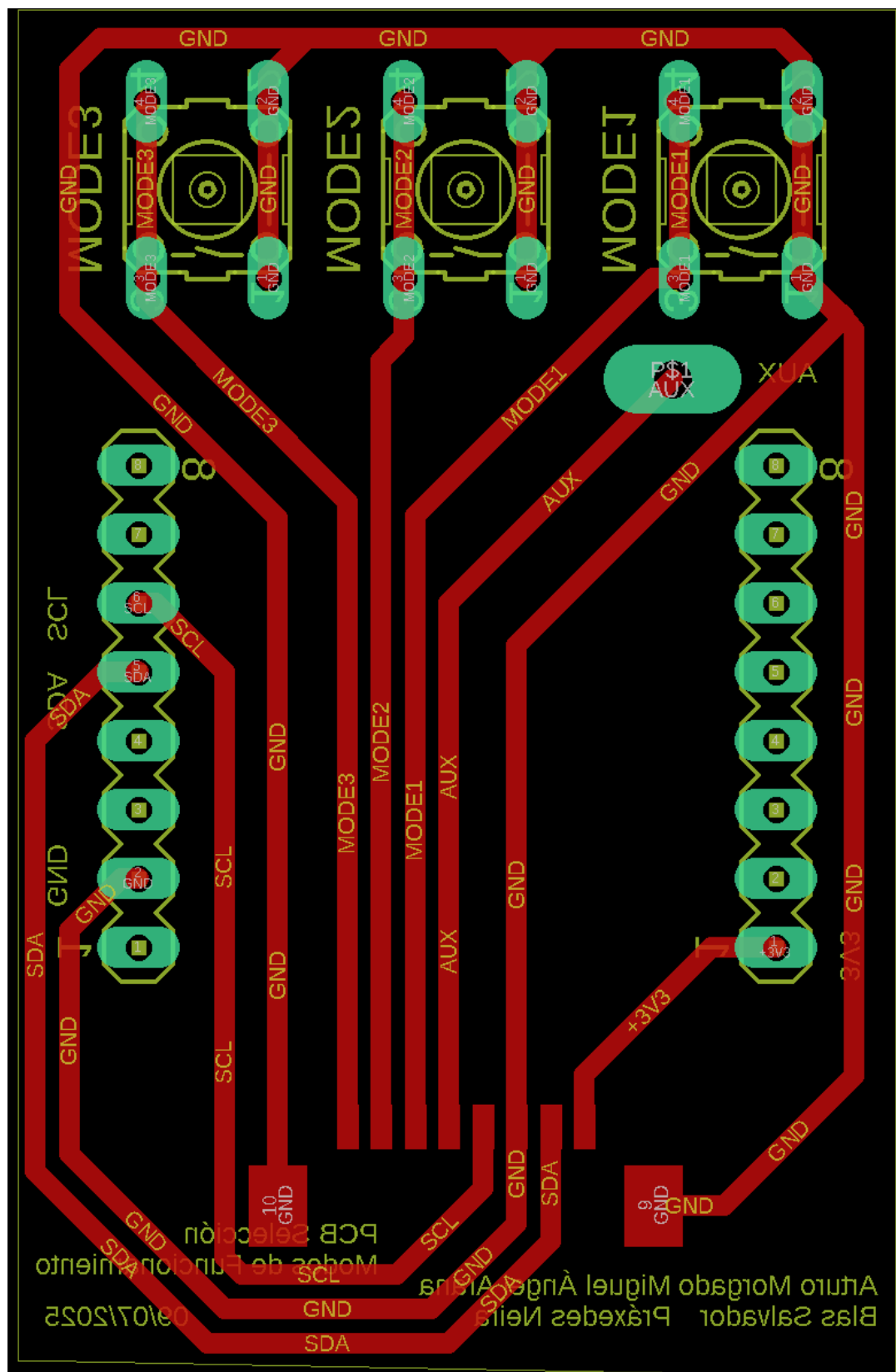
C.2.3. Board Bottom PCB capacitivos



C.3.1. Esquemático PCB botones y LCD

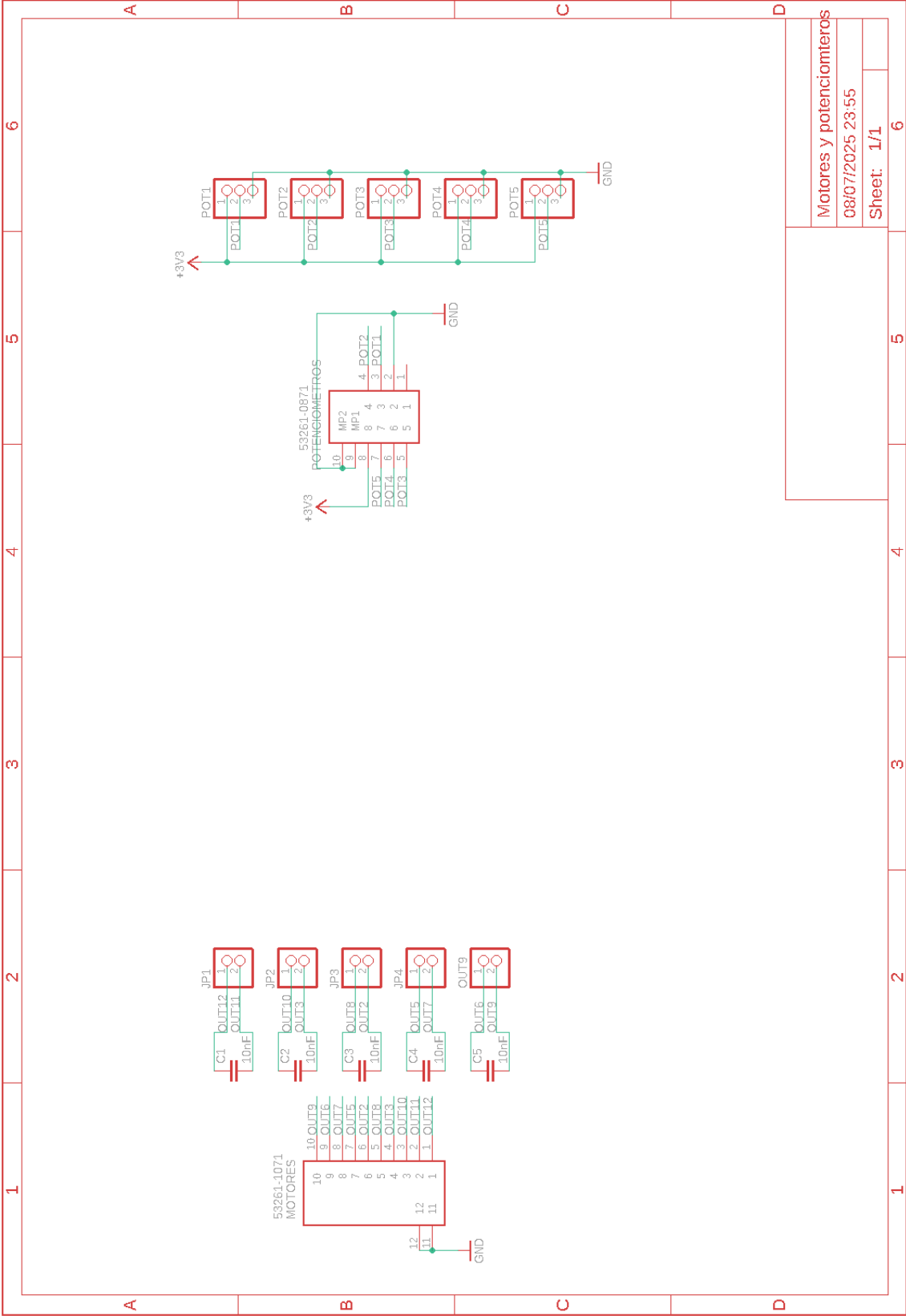


C.3.2. Board PCB botones y LCD

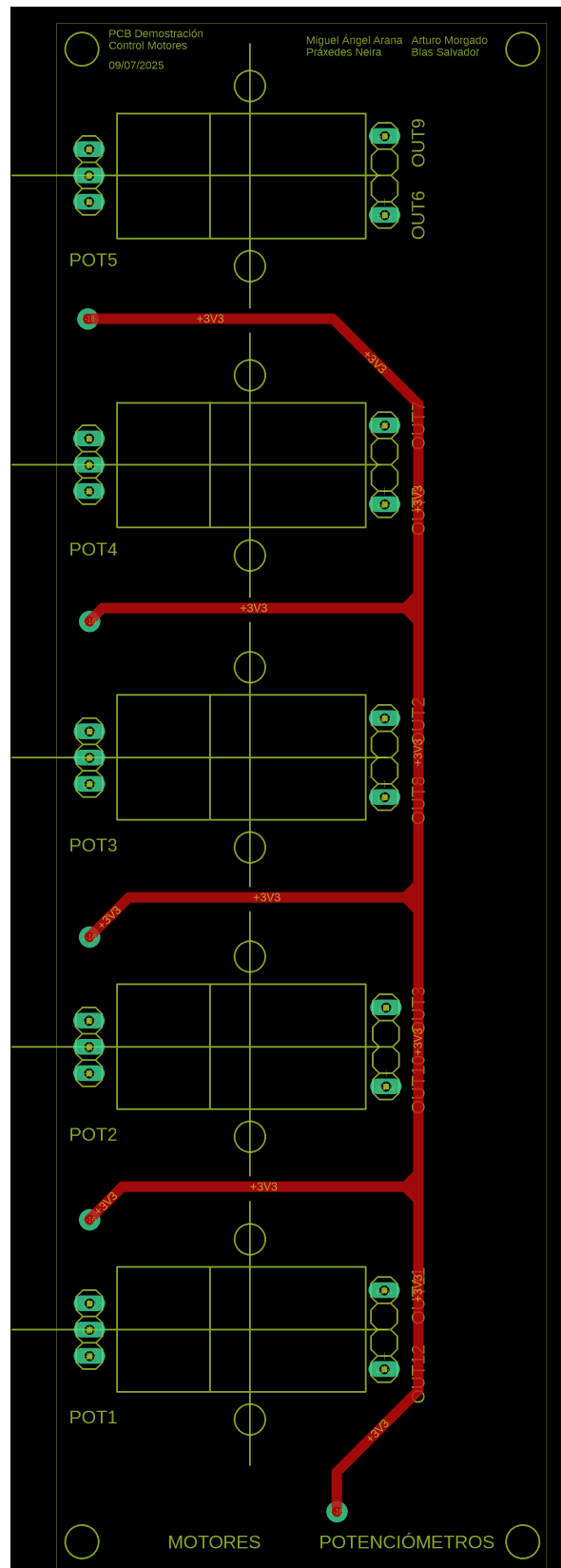


C.4. PCB motores

C.4.1. Esquemático PCB motores



C.4.2. Board Top PCB motores



C.4.3. Board Bottom PCB motores

