

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

Memoria de Título

Desarrollo de Control Remoto con Botones Programables para la Tercera Edad

Memoria de Título para optar al título de
Ingeniería Civil Electrónica

Alumno

Tomás Alfredo Riveros Moreno

Profesor Guía

Agustín González Valenzuela

Comisión evaluadora

Alejandro Weinstein Oppenheimer, Correferente, UTFSM

Diciembre 2025, Valparaíso, Chile



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Desarrollo de Control Remoto con Botones Programables para la Tercera Edad

Nombre del candidato(a): Tomás Alfredo Riveros Moreno

Carrera / Grado: Ingeniería Civil Electrónica

Campus: Casa Central

Departamento: Electrónica

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Agustín J. González Valenzuela, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar una opción**):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 13 / 03 / 2026

Firma:

Agustín J. González

Estudiante o Candidato(a):

Fecha: 13 de marzo de 2026

Firma:

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

Dedicado a Sebastián Fernández, que me motivó a mucho y no estuvo aquí para ver el resultado

Agradecimientos

Gracias a mi madre, Marcela Moreno, por darme su apoyo siempre que me hizo falta e incentivar-me actuar en momentos de duda. A mis abuelos, Silvia Fernández y Enrique Moreno, por ser mis más grandes acompañantes de niño y quienes hoy me prohíben rendirme. A mi hermano, David Videla, por la felicidad que logra aportarme. A mi tío, Eduardo Moreno, por estar siempre disponible. A mi padre, Alfredo Riveros, por estar siempre presente sin importar la distancia. A mis abuelos, Nubia Tello y Ed-mundo Riveros, por siempre hacerme sentir bienvenido y parte de la familia. A mis tías Nubia Riveros y Andrea Riveros, por imposibilitar que se me olvidé que hay gente que me quiere. A mis primos Catalina Sierra y Cristóbal Riveros, por ser mi equipo de travesuras y con quienes he disfrutado toda la vida y así seguirá siendo.

A mis amigos, Pablo Fernández, Cristóbal López, Facundo Lorca, Nicolás Salgado, Ámbar Ávalos y Ne-kane Zubieta, por haber decidido que una amistad de colegio podía ir más allá de eso, y gracias por darme la confianza en mi mismo que nunca tuve. A mi amigo Daniel Cubillos, por ser un compañero tan confia-ble. A mis amigos, Raúl Cuello y José Tudela, por haber elegido a alguien en su momento muy callado, para conversar en una sala llena de gente. A mis labubus, Claudio Zanetta y Josefina Vera, por todo el tiempo disfrutado y mostrarme que hacer más amigos vale la pena.

A mi profesor, Agustín González, por la idea y la confianza en este proyecto.

Sinceramente, me encantaría escribir más, pero las palabras que tengo para cada uno son de las que se dicen en persona. Para todos y todas ustedes, mis más sinceros agradecimientos por todo, por cada risa que he dado y que daré, por cada éxito que he tenido y que tendré, por cada lágrima que me ayudaron a calmar, por cada consejo, por cada historia, por cada tontería. Por todo eso y mucho más les doy las gracias.

Desarrollo de Control Remoto con Botones Programables para la Tercera Edad

Tomás Alfredo Riveros Moreno

Memoria para optar al Título de Ingeniería Civil Electrónica

Profesor Guía: Agustín González Valenzuela

Diciembre 2025

La integración de tecnologías digitales en la vida cotidiana ha generado una brecha significativa para el segmento de la tercera edad. Los controles remotos modernos, caracterizados por una alta densidad de botones e interfaces complejas, imponen una carga cognitiva excesiva que limita la autonomía de los adultos mayores en actividades de ocio básicas.

Esta memoria presenta el diseño, implementación y validación del **Sistema de Asistencia IR Configurable (SAIC)**, una solución tecnológica híbrida que desacopla la complejidad de la configuración y variedad de operaciones de controles remotos de la simplicidad de las operaciones de uso diario. El sistema consta del dispositivo SAIC basado en el microcontrolador STM32L476RG, diseñado con una interfaz minimalista de botones grandes, y una aplicación móvil Android (“IR-Link Admin”) que permite a un administrador configurar de manera inalámbrica las funciones y macros del dispositivo vía Bluetooth.

La implementación técnica destaca por el desarrollo de un firmware *Bare-Metal* eficiente, utilizando una arquitectura no bloqueante basada en interrupciones para la síntesis precisa de protocolos infrarrojos (NEC y RC5) y un protocolo de comunicación serie robusto (MTR-Link). Asimismo, se incorpora un sistema de persistencia de datos en memoria Flash.

Los resultados de la verificación demuestran que el dispositivo SAIC garantiza un alcance superior a 5 metros y una autonomía estimada de 50 horas en uso continuo, validando su viabilidad como herramienta de asistencia tecnológica efectiva y replicable.

Palabras clave: Accesibilidad, Adulto Mayor, STM32, Infrarrojo, Android, Sistemas Embebidos.

Development of a Remote Control with Programmable Buttons for the Elderly

Tomás Alfredo Riveros Moreno

Final Project Report submitted for the degree of Electronic Civil Engineer

Advisor: Agustín González Valenzuela

December 2025

The integration of digital technologies into daily life has created a significant gap for the elderly population. Modern remote controls, characterized by high button density and complex interfaces, impose an excessive cognitive load that limits the autonomy of older adults in basic leisure activities.

This thesis presents the design, implementation, and validation of the **Configurable IR Assistance System (SAIC)**, a hybrid technological solution that decouples configuration complexity from operational simplicity. The system consists of a physical hardware device based on the STM32L476RG microcontroller, designed with a minimalist interface of large buttons, and an Android mobile application (“IR-Link Admin”) that allows an administrator to remotely configure device functions and macros via Bluetooth.

The technical implementation features the development of efficient *Bare-Metal* firmware, utilizing a non-blocking interrupt-based architecture for the precise synthesis of infrared protocols (NEC and RC5) and a robust serial communication protocol (MTR-Link). Furthermore, a data persistence system is incorporated using EEPROM emulation in Flash memory.

Verification results demonstrate that the device guarantees a range exceeding 5 meters and an estimated autonomy of 50 hours in continuous use, validating its viability as an effective and replicable assistive technology tool.

Keywords: Accessibility, Elderly, STM32, Infrared, Android, Embedded Systems.

Glosario y Siglas

- **ANR:** Application Not Responding (Error de aplicación no responde en Android).
- **BLE:** Bluetooth Low Energy (Tecnología inalámbrica de bajo consumo).
- **FSM:** Finite State Machine (Máquina de Estados Finitos).
- **GPIO:** General Purpose Input/Output (Entrada/Salida de Propósito General).
- **HAL:** Hardware Abstraction Layer (Capa de Abstracción de Hardware).
- **IR:** Infrarrojo.
- **JSON:** JavaScript Object Notation (Formato ligero de intercambio de datos).
- **MCU:** Microcontroller Unit (Unidad de Microcontrolador).
- **MTR:** Mobile To Remote (Protocolo de comunicación propuesto).
- **PWM:** Pulse Width Modulation (Modulación por Ancho de Pulso).
- **SAIC:** Sistema de Asistencia IR Configurable.
- **SPP:** Serial Port Profile (Perfil de Puerto Serie en Bluetooth).
- **SWD:** Serial Wire Debug (Protocolo de depuración para procesadores ARM).
- **UART:** Universal Asynchronous Receiver-Transmitter.
- **BJT:** Bipolar Junction Transistor (Transistor de Unión Bipolar).

Índice general

Agradecimientos	II
Resumen	III
Abstract	IV
Glosario y Siglas	V
1. Introducción	1
1.1. Contexto del Problema	1
1.1.1. El Contexto Demográfico y Digital en Chile	2
1.2. Soluciones Actuales y su Problemática	3
1.2.1. Controles Universales “Simplificados”	3
1.2.2. Aplicaciones Móviles y Hubs Inteligentes	3
1.3. Definición del Problema Específico	4
1.4. Nombre de la Solución Propuesta	4
1.5. Objetivos del Proyecto	4
1.5.1. Objetivo General	4
1.5.2. Objetivos Específicos	5
1.6. Estructura del Documento	5
2. Tecnologías de Control Remoto y Conectividad	6
2.1. Fundamentos de la Comunicación Infrarroja (IR)	6
2.1.1. Principios Físicos	6
2.1.2. Protocolos de Codificación Estándar	7
2.2. Fundamentos de Conectividad Bluetooth	10
2.2.1. Bluetooth Classic vs. Bluetooth Low Energy (BLE)	10
2.3. Accesibilidad en Interfaces Móviles	11
2.4. Estado del Arte en Soluciones de Control Universal	11
2.4.1. Controles Universales Programables	11
2.4.2. Aplicaciones Móviles y Hubs IR/Wi-Fi	11
2.5. Justificación de la Solución Propuesta	12

3. Análisis y Arquitectura de SAIC	13
3.1. Definición de Requerimientos de SAIC	13
3.1.1. Requerimientos Funcionales (RF)	13
3.2. Análisis de Casos de Uso	14
3.2.1. Caso de Uso 1: Configuración de Botón	15
3.2.2. Caso de Uso 2: Operación Diaria	15
3.3. Selección de Tecnología	15
3.4. Arquitectura General del Sistema	16
3.4.1. Capa de Aplicación (Android - Admin)	17
3.4.2. Capa de Control (Firmware STM32)	17
3.4.3. Capa Física (Hardware)	17
3.5. Diagramas de Secuencia del Sistema	17
4. Diseño Detallado de SAIC	19
4.1. Metodología y Evolución del Diseño	19
4.1.1. Iteraciones de Hardware	19
4.1.2. Decisiones de Arquitectura de Firmware	19
4.1.3. Justificación del Protocolo MTR-Link	20
4.2. Diseño del Hardware de SAIC: Control SAIC	20
4.2.1. Selección de Componentes	21
4.2.2. Diseño del Circuito (Esquemático)	21
4.3. Diseño del Firmware de SAIC: Control SAIC	23
4.3.1. Síntesis de Portadoras mediante PWM por Hardware	24
4.3.2. Arquitectura del Software de Firmware	24
4.3.3. Máquina de Estados de Envío (Macros)	25
4.3.4. Máquinas de Estados de Bajo Nivel (Drivers IR)	27
4.3.5. Gestión Dinámica de Protocolos y Reconfiguración de Registros	30
4.4. Diseño del Protocolo de Comunicación API MTR-Link	31
4.4.1. Comandos (App → STM32)	31
4.4.2. Respuestas (STM32 → App)	32
4.4.3. Manejo de Errores y Timeouts	32
4.5. Diseño del Software de la Aplicación IR-Link Admin	32
4.5.1. Arquitectura de Clases	32
4.5.2. Estructura de la Base de Datos (JSON)	32
5. Implementación de SAIC	33
5.1. Implementación del Firmware (STM32)	33
5.1.1. Lógica de Persistencia (Memoria Flash)	36
5.2. Implementación del Software (App IR-Link Admin)	37
5.2.1. Interfaz de Usuario (Grid Layout)	37

5.2.2.	Arquitectura de Datos (Base de Datos JSON)	38
5.2.3.	Gestión de Conectividad y Concurrency	38
5.2.4.	Flujo de Configuración (Lógica de Negocio)	39
5.3.	Desafíos de Implementación	40
5.3.1.	Precisión de Tiempos y Bloqueo de CPU	40
5.3.2.	Gestión de Memoria Flash (Read-Modify-Write)	40
6.	Verificación de SAIC	42
6.1.	Verificación del Protocolo de Comunicación MTR-Link	42
6.2.	Verificación de la Generación de Señal IR	42
6.2.1.	Verificación de la Capa Física (Portadora)	42
6.2.2.	Modelo de Referencia y Validación Temporal (Protocolo NEC)	44
6.3.	Verificación Dinámica de Macros	45
6.4.	Verificación Funcional Completa (Prueba de Usabilidad)	46
6.5.	Evaluación del Alcance y Potencia	47
6.6.	Análisis de Consumo Energético	48
7.	Conclusiones y Trabajo Futuro	50
7.1.	Conclusiones Generales	50
7.2.	Trabajo Futuro	51
	Bibliografía	53
A.	Código Fuente del Firmware (STM32)	55
B.	Código Fuente de la App (MainActivity.java)	72
C.	Código Fuente del Diálogo (ConfigureButtonDialogFragment.java)	97
D.	Base de Datos de Códigos (ir_codes.json)	101
E.	Esquema Eléctrico del Hardware	107

Índice de figuras

1.1. Comparación conceptual entre la complejidad de un control estándar y la simplicidad objetivo de SAIC.	2
2.1. Ubicación de la banda IR utilizada para control remoto (940 nm) dentro del espectro electromagnético.	7
2.2. Diagrama de tiempos del protocolo NEC.	8
2.3. Diagrama de tiempos del protocolo RC5.	9
3.1. Arquitectura general del sistema SAIC, detallando los módulos de software y hardware. .	16
3.2. Diagrama de secuencia detallado para el almacenamiento de una nueva configuración, mostrando el flujo de confirmación (ACK).	18
4.1. Diagrama esquemático de la Etapa de Potencia.	20
4.2. Diagrama esquemático del hardware de control SAIC, mostrando las conexiones entre MCU, Bluetooth, Teclado y Etapa de Potencia.	22
4.3. Diagrama de bloques de la arquitectura del firmware.	25
4.4. Diagrama de estados de Macros.	26
4.5. Máquina de Estados del driver NEC (Bajo Nivel).	28
4.6. Máquina de fases del driver RC5 (Bajo Nivel). Se observa la estructura cíclica y simétrica de la transmisión Manchester.	30
5.1. Interfaz principal de IR-Link Admin.	37
5.2. Diálogo modal para la asignación de macros.	39
6.1. Oscilograma de la señal portadora. Se observa una frecuencia de 37,31 kHz y la conmutación de la carga IR.	43
6.2. Medición del voltaje de saturación del transistor 2N2222. El cursor B (CurB) registra un valor de 560 mV en el nivel bajo de la señal de colector.	44
6.3. Medición del <i>Header Mark</i> (9.00 ms).	45
6.4. Medición del <i>Header Space</i> (4.40 ms).	45
6.5. Detalle de un pulso de bit individual (<i>Bit Mark</i>). La duración medida con los cursores es de 560 μ s, lo que representa una desviación mínima del 0,4% respecto al estándar de 562,5 μ s.	45

6.6. Reconstrucción vectorial de una macro de 4 pasos a partir de telemetría. El análisis confirma un intervalo inter-trama estable de ≈ 167 ms.	46
E.1. Esquema Eléctrico del Hardware	107

Índice de tablas

2.1. Comparación cualitativa entre soluciones de control existentes y el sistema SAIC.	11
3.1. Matriz de decisión para la selección del microcontrolador.	16
6.1. Comparación entre el estándar NEC y la medición experimental en osciloscopio.	44
6.2. Resultados empíricos de las pruebas de alcance y directividad.	48
6.3. Desglose del consumo de corriente medido en el sistema SAIC.	48

Capítulo 1

Introducción

La rápida evolución de la tecnología de consumo ha transformado radicalmente la interacción diaria con dispositivos electrónicos, integrándolos en casi todos los aspectos de la vida moderna. Sin embargo, este avance vertiginoso ha generado una “brecha digital” que afecta desproporcionadamente a la población de adultos mayores [1]. Mientras que los dispositivos actuales ofrecen una funcionalidad cada vez mayor, sus interfaces de usuario a menudo priorizan la densidad de funciones y la estética minimalista sobre la accesibilidad y la usabilidad, creando barreras significativas para aquellos con capacidades sensoriales, motoras o cognitivas disminuidas [2].

Este capítulo introduce el proyecto “**Sistema de Asistencia IR Configurable (SAIC)**”, diseñado para mitigar estas barreras en el contexto específico del control de dispositivos audiovisuales, una actividad de ocio fundamental para la calidad de vida y la autonomía de la tercera edad.

1.1. Contexto del Problema

En la actualidad, el control remoto infrarrojo (IR) sigue siendo la interfaz predominante para la interacción con televisores, decodificadores y equipos de audio. No obstante, los controles remotos estándar suministrados con estos equipos han evolucionado hacia dispositivos de alta complejidad, alejándose de los principios de diseño universal.

Según estudios de usabilidad en gerontotecnología, los principales problemas que enfrentan los adultos mayores con estos dispositivos son:

1. **Densidad de Botones y Sobrecarga Cognitiva:** Los controles modernos frecuentemente superan los 40 o 50 botones, la mayoría de los cuales nunca son utilizados por el usuario promedio. Esta saturación visual dificulta la localización de funciones básicas como “Encender” o “Cambiar Canal”, incrementando la carga cognitiva necesaria para operar el dispositivo [3].
2. **Ergonomía y Motricidad Fina:** La tendencia hacia la miniaturización ha resultado en botones pequeños y con espaciado reducido. Esto representa un desafío crítico para usuarios con patologías como artritis o disminución de la motricidad fina, derivando en pulsaciones accidentales y errores de operación.

3. **Déficits Sensoriales (Visión y Tacto):** Las etiquetas de texto suelen ser pequeñas y de bajo contraste (ej. gris sobre negro), haciéndolas ilegibles para personas con presbicia o cataratas. Además, la falta de diferenciación táctil entre botones impide el uso intuitivo sin contacto visual constante.

La Figura 1.1 ilustra conceptualmente la diferencia entre la alta densidad de botones de un control remoto estándar, que genera una alta carga cognitiva y fatiga decisional, y la interfaz simplificada del sistema SAIC, diseñada para mitigar este problema.

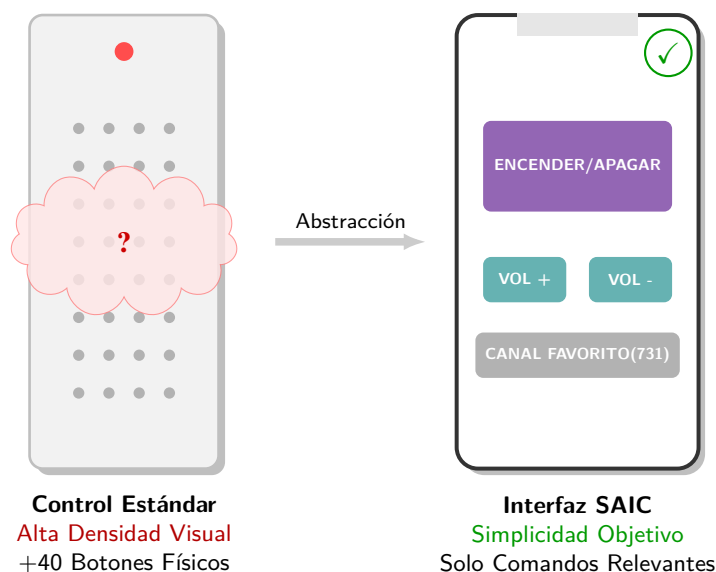


Figura 1.1: Comparación conceptual entre la complejidad de un control estándar y la simplicidad objetivo de SAIC.

Para un usuario promedio, esta densidad de controles puede generar una fricción innecesaria; pero para un adulto mayor, constituye una barrera de entrada significativa que limita su autonomía, generando dependencia de terceros para tareas triviales como ajustar el volumen, lo que impacta negativamente en su autoestima.

La problemática se agrava por la falta de estandarización en las interfaces y la introducción de botones multifunción que requieren secuencias de pulsaciones precisas o tiempos de respuesta rápidos. Esta complejidad contrasta con las necesidades de una población que requiere interfaces intuitivas, robustas y con retroalimentación clara.

1.1.1. El Contexto Demográfico y Digital en Chile

La relevancia de este problema se magnifica al analizar la realidad demográfica de Chile. Según proyecciones del Instituto Nacional de Estadísticas (INE) [4], el país atraviesa un proceso de envejecimiento acelerado. En 2024, aproximadamente el 19 % de la población tiene 60 años o más, y se proyecta que esta cifra alcance el 31 % para el año 2050. Este cambio plantea desafíos urgentes en términos de integración social y tecnológica.

A pesar de la alta penetración de Internet en el país, persiste una brecha digital significativa en este

segmento etario. Un informe reciente del Observatorio del Envejecimiento UC-Confuturo (2025) [5] revela que, si bien el 89 % de las personas mayores posee un teléfono celular, solo el 51 % cuenta con un smartphone capaz de ejecutar aplicaciones modernas. Más crítico aún es el uso efectivo de estas tecnologías: solo el 42 % de los adultos mayores declara utilizar internet de manera regular, limitándose principalmente a la búsqueda de información básica y comunicación, mientras que usos más complejos alcanzan apenas un 29 %.

Estudios adicionales, como la “Radiografía Digital de Personas Mayores” [6], indican que la principal barrera no es solo el acceso, sino la usabilidad y la falta de alfabetización digital. Iniciativas gubernamentales como el “Plan Brecha Digital Cero” del SENAMA [7] buscan mitigar esto, pero la curva de aprendizaje para dispositivos táctiles sigue siendo empinada. En este contexto, soluciones que aprovechen interfaces físicas familiares (como un control remoto tradicional) pero potenciadas por tecnología moderna, ofrecen un puente pragmático para la inclusión tecnológica.

1.2. Soluciones Actuales y su Problemática

El mercado ofrece diversas soluciones orientadas a simplificar el control de dispositivos, pero estas presentan limitaciones significativas (técnicas y de usabilidad) cuando se evalúan desde la perspectiva de la accesibilidad para la tercera edad. A continuación se analizan las dos categorías principales:

1.2.1. Controles Universales “Simplificados”

Son dispositivos físicos diseñados específicamente para adultos mayores, caracterizados por tener botones grandes y un número reducido de funciones. Sin embargo, presentan limitaciones importantes:

- **Configuración Compleja:** Su programación suele depender de métodos poco intuitivos como la búsqueda automática de códigos (ensayo y error) o la introducción manual de secuencias numéricas (“Magic Codes”), procesos confusos para el usuario objetivo.
- **Rigidez Funcional:** Una vez programados, se limitan a replicar funciones individuales básicas. No permiten crear “Macros” o secuencias personalizadas. Por ejemplo, si un usuario desea ver su canal favorito (ej. “731”), debe presionar tres botones en secuencia rápida, una tarea motora compleja que estos controles no resuelven automáticamente.

1.2.2. Aplicaciones Móviles y Hubs Inteligentes

Las soluciones basadas en software, como aplicaciones en smartphones (ej. Xiaomi Mi Remote [8]) o hubs IR conectados a Wi-Fi (ej. BroadLink [9]), ofrecen gran flexibilidad, pero introducen nuevas barreras:

- **La “Barrera del Smartphone”:** Para realizar una acción simple, el usuario debe localizar el teléfono, desbloquearlo, navegar por menús y abrir una aplicación. Este proceso es cognitivamente mucho más demandante que presionar un botón físico dedicado.

- **Ausencia de Feedback Háptico:** Las pantallas táctiles carecen de retroalimentación física, obligando al usuario a mirar la pantalla para confirmar cada pulsación, lo que dificulta el uso por “memoria muscular” [10].
- **Dependencia de Conectividad:** Muchas de estas soluciones requieren conexión a Internet o Wi-Fi estable, añadiendo puntos de falla innecesarios para una tarea local.

1.3. Definición del Problema Específico

Existe una carencia de soluciones que combinen la simplicidad de una interfaz física dedicada con la potencia y flexibilidad de la configuración por software moderna. El problema específico que aborda este proyecto es la necesidad de un dispositivo de “accesibilidad tecnológica” que **disocie la complejidad de la configuración de la simplicidad del uso diario**.

Se requiere un sistema que ofrezca al usuario final una interfaz física minimalista y robusta (Hardware), mientras permite a un administrador (familiar o cuidador) configurar comportamientos complejos, como macros de múltiples pasos, a través de una interfaz rica y amigable (Software). Esta separación de roles (Administrador vs. Usuario Final) es clave para el éxito de la tecnología de asistencia.

1.4. Nombre de la Solución Propuesta

Para responder a esta necesidad, se propone el diseño e implementación del “**Sistema de Asistencia IR Configurable**” (SAIC). Este sistema es una solución híbrida compuesta por:

1. **Control SAIC (Hardware):** Un dispositivo físico simplificado, basado en el microcontrolador STM32, diseñado para el usuario final. Prioriza la ergonomía, la robustez y la respuesta inmediata.
2. **App IR-Link Admin (Software):** Una aplicación móvil Android destinada al administrador, encargada de la configuración avanzada y gestión del hardware mediante conectividad Bluetooth.

1.5. Objetivos del Proyecto

El propósito fundamental de este trabajo es desarrollar una herramienta tecnológica que promueva la independencia de los adultos mayores en su interacción con dispositivos de entretenimiento doméstico.

1.5.1. Objetivo General

Diseñar, implementar y validar un prototipo funcional del “Sistema de Asistencia IR Configurable” (SAIC), enfocado en adultos mayores, que consiste en un hardware físico simplificado y un software de configuración móvil (Android/Bluetooth) para un administrador.

1.5.2. Objetivos Específicos

Para alcanzar el objetivo general, se han definido los siguientes objetivos específicos:

1. Diseñar e implementar el firmware del Control SAIC sobre un microcontrolador STM32L476RG, capaz de generar señales IR precisas para múltiples protocolos estándar (NEC, RC5).
2. Desarrollar un sistema de almacenamiento persistente en la memoria Flash del microcontrolador para asegurar que la configuración se mantenga ante cortes de energía, garantizando la robustez del dispositivo.
3. Diseñar un protocolo de comunicación serie, denominado “MTR-Link”, sobre Bluetooth para la transferencia segura y eficiente de datos de configuración entre la aplicación móvil y el hardware.
4. Desarrollar la aplicación “IR-Link Admin” para Android, que permita gestionar una base de datos de códigos IR y configurar los botones físicos del hardware, incluyendo la creación de macros de múltiples pasos.
5. Evaluar el consumo energético del prototipo en sus distintos estados operativos para estimar la autonomía de la batería y validar su viabilidad como dispositivo portátil.

1.6. Estructura del Documento

La presente memoria se organiza en siete capítulos que detallan el proceso de desarrollo del sistema SAIC:

- **Capítulo 2: Tecnologías de Control Remoto y Conectividad.** Revisa los fundamentos teóricos de la comunicación infrarroja, los protocolos estándar y el estado del arte en soluciones de control.
- **Capítulo 3: Análisis y Arquitectura de SAIC.** Define los requerimientos funcionales y no funcionales, y presenta la arquitectura general del sistema híbrido.
- **Capítulo 4: Diseño Detallado de SAIC.** Profundiza en el diseño del hardware, la lógica del firmware y la arquitectura de software de la aplicación móvil.
- **Capítulo 5: Implementación de SAIC.** Describe el proceso de construcción del prototipo, detallando la configuración de periféricos, algoritmos clave y desarrollo de la interfaz de usuario.
- **Capítulo 6: Verificación de SAIC.** Presenta los resultados de las pruebas realizadas para verificar el cumplimiento de los requerimientos Y evaluar su desempeño.
- **Capítulo 7: Conclusiones y Trabajo Futuro.** Resume los logros alcanzados, las limitaciones detectadas y propone líneas de trabajo para futuras iteraciones del proyecto.

Capítulo 2

Tecnologías de Control Remoto y Conectividad

En este capítulo se establece el marco teórico y técnico necesario para el desarrollo del proyecto. Se revisan los fundamentos físicos de la comunicación infrarroja, los protocolos de codificación estándar utilizados en la industria de electrónica de consumo y se analiza el estado del arte de las soluciones de control universal existentes, con el fin de identificar las limitaciones presentes para el público objetivo de este trabajo.

2.1. Fundamentos de la Comunicación Infrarroja (IR)

La comunicación por infrarrojos (IR) es una tecnología de transmisión inalámbrica de línea de vista (LOS), de bajo costo y corto alcance, que utiliza pulsos de luz no visible para el transporte de información. Constituye el estándar predominante para el control remoto de dispositivos audiovisuales debido a su inmunidad frente a interferencias de radiofrecuencia (RF) y su contención espacial [11].

2.1.1. Principios Físicos

El espectro infrarrojo comprende longitudes de onda superiores a la luz visible (aproximadamente 700 nm) e inferiores a las microondas. Para aplicaciones de control remoto, se emplea típicamente la banda del infrarrojo cercano (Near Infrared - NIR), con una longitud de onda pico de **940 nm**. Esta longitud de onda es generada de manera eficiente mediante diodos emisores de luz (LEDs) basados en Arseniuro de Galio (GaAs), tales como el modelo **TSAL6200**, seleccionado para este proyecto por su alta intensidad radiante [12]. La Figura 2.1 ilustra la posición de esta banda dentro del espectro electromagnético, esta longitud de onda pertenece al Infrarrojo Cercano (NIR), invisible al ojo humano pero detectable por fotodiodos de silicio estándar.

Para distinguir la señal de control de otras fuentes de radiación infrarroja ambiental (como la luz solar o la iluminación artificial), la señal no se transmite en banda base, sino que se modula sobre una frecuencia portadora. Las frecuencias estándar oscilan entre 30 kHz y 60 kHz, siendo **38 kHz** y **36 kHz** las más

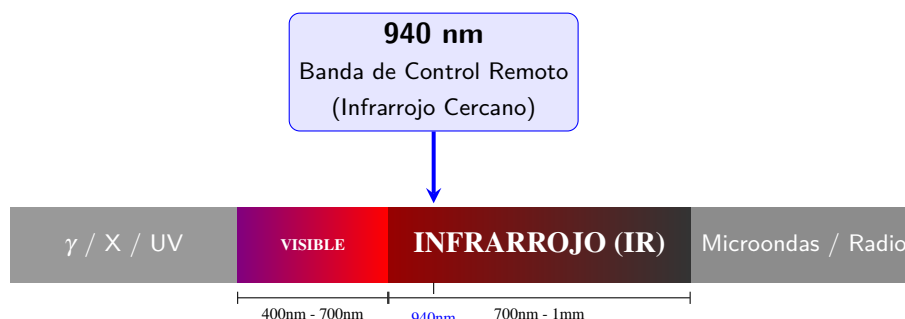


Figura 2.1: Ubicación de la banda IR utilizada para control remoto (940 nm) dentro del espectro electro-magnético.

comunes. Un aspecto crítico del diseño es el **ciclo de trabajo** (duty cycle) de la portadora, típicamente del 25 % al 33 %. Este bajo ciclo de trabajo permite impulsar el LED con corrientes de pico altas (superiores a 100 mA) sin exceder su disipación de potencia media, lo que maximiza el alcance efectivo.

El receptor (ej. módulo TSOP) incorpora un filtro paso banda óptico y electrónico sintonizado a esta frecuencia, lo cual permite rechazar el ruido ambiental y entregar un tren de pulsos digitales limpios (demodulados) al microcontrolador.

2.1.2. Protocolos de Codificación Estándar

La información digital se codifica mediante la modulación de la presencia (Marca) o ausencia (Espacio) de la portadora. Existen múltiples protocolos, siendo NEC [13] y RC5 [14] los más relevantes para este estudio. La implementación de estos protocolos requiere una temporización precisa del orden de los microsegundos, aspecto que fundamenta la selección de temporizadores de hardware (Hardware Timers) en la etapa de diseño posterior.

Protocolo NEC

El protocolo NEC utiliza una codificación por distancia de pulsos (*Pulse Distance Encoding*) para transmitir los datos, los cuales viajan modulados sobre una señal portadora infrarroja. Para comprender su estructura, es fundamental distinguir entre la capa física (modulación) y la capa de enlace (codificación lógica)

- **Modulación de la Portadora:** La señal infrarroja se emite a una frecuencia base típica de 38 kHz. Para maximizar la eficiencia energética y la vida útil del diodo emisor, esta señal portadora suele configurarse con un ciclo de trabajo (*Duty Cycle*) de 1/3 (o 1/4). Esto significa que, durante el tiempo que dura un pulso de transmisión, el LED infrarrojo conmuta rápidamente, permaneciendo encendido solo un 33 % del tiempo de cada ciclo de la portadora.
- **Codificación Lógica (Bits):** Como se ilustra en la Figura 2.2, los bits de información ('0' y '1') se diferencian estrictamente por la duración del espacio de silencio que le sigue a un pulso activo.

Todos los pulsos activos (ráfagas de portadora a 38 kHz) tienen una duración temporal fija de $562.5 \mu s$.

- **Bit Lógico '0'**: Se compone de un pulso activo de $562.5 \mu s$ seguido de un espacio inactivo de $562.5 \mu s$. El tiempo total de transmisión es de 1.125 ms , presentando una relación de tiempo activo-inactivo de 1:1 (mitad y mitad).
- **Bit Lógico '1'**: Se compone de un pulso activo de $562.5 \mu s$ seguido de un espacio inactivo prolongado de 1.6875 ms . El tiempo total de transmisión es de 2.25 ms , donde el tiempo activo representa solo un cuarto ($1/4$) de la duración total del bit.

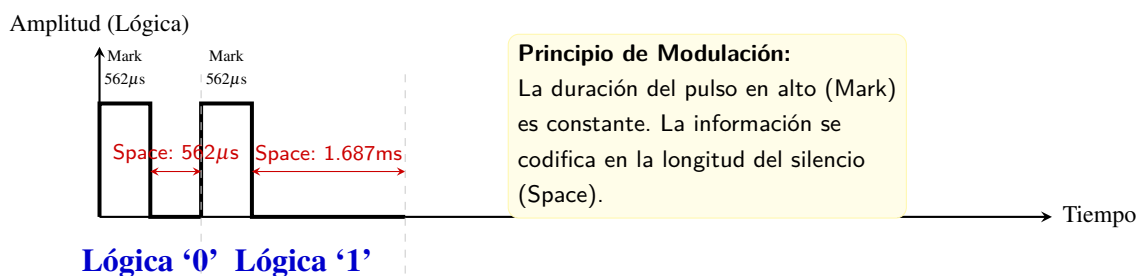


Figura 2.2: Diagrama de tiempos del protocolo NEC.

Se observa la modulación por distancia de pulsos, donde un espacio corto representa un '0' lógico y un espacio largo representa un '1' lógico.

- **Estructura de la Trama (Mensaje Completo):** Un mensaje estándar en el protocolo NEC transmite una palabra de 32 bits, precedida por un pulso de sincronización, estructurada en el siguiente orden:
 1. **Encabezado (Header/Leader Code):** Un pulso de sincronización inicial de 9 ms seguido de un espacio de $4,5 \text{ ms}$. Sirve para despertar al receptor y ajustar su Control Automático de Ganancia (AGC).
 2. **Dirección (Address):** 8 bits que identifican al fabricante o tipo de dispositivo objetivo (ej. Televisor).
 3. **Dirección Invertida ($\sim$ Address):** Los mismos 8 bits invertidos lógicamente, utilizados para detección de errores.
 4. **Comando (Command):** 8 bits que contienen la instrucción específica a ejecutar (ej. Subir volumen).
 5. **Comando Invertido ($\sim$ Command):** Los 8 bits del comando invertidos lógicamente para validación de integridad.
 6. **Bit de Parada (Stop Bit):** Un pulso activo final de $562,5 \mu s$ para marcar explícitamente el final del último bit de espacio.

Protocolo RC5

Al igual que en el caso anterior, para comprender la estructura del protocolo RC5 de Philips es imperativo separar el análisis de la capa física (modulación analógica) de la capa de enlace (codificación lógica).

- **Modulación de la Portadora:** La señal infrarroja se emite a una frecuencia base típica de 36 kHz (aunque algunos receptores toleran 38 kHz). Similar al estándar NEC, la portadora emplea un ciclo de trabajo (*Duty Cycle*) reducido, típicamente de 1/4 o 1/3, lo que significa que dentro de cada ciclo de la portadora, el emisor IR está activo solo entre un 25 % y un 33 % del tiempo, reduciendo drásticamente el consumo de corriente promedio.
- **Codificación Lógica (Bits):** A diferencia de la modulación por distancia de pulsos, el protocolo RC5 utiliza **Codificación Manchester** (bifase). En este esquema, todos los bits lógicos tienen exactamente la misma duración total (1,778 ms) y el valor del bit está determinado por una transición de estado en la mitad exacta del periodo ($889\mu s$), como se ilustra en la Figura 2.3.
 - **Bit Lógico '0':** Se compone de una ráfaga activa de portadora durante la primera mitad del periodo ($889\mu s$), seguida de un espacio inactivo durante la segunda mitad ($889\mu s$). Es decir, presenta una transición de alto a bajo en el medio del bit.
 - **Bit Lógico '1':** Se compone de un espacio inactivo durante la primera mitad del periodo ($889\mu s$), seguido de una ráfaga activa de portadora en la segunda mitad ($889\mu s$). Es decir, presenta una transición de bajo a alto en el medio del bit.

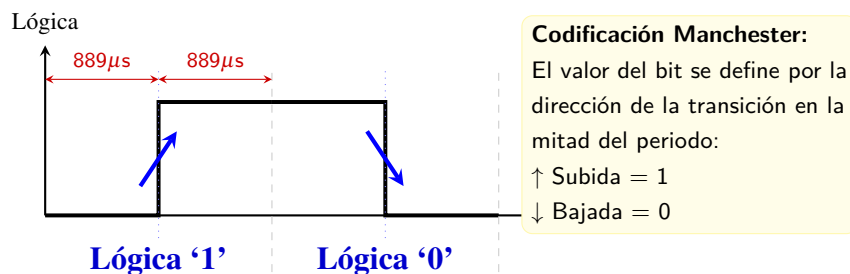


Figura 2.3: Diagrama de tiempos del protocolo RC5.

Se observa la codificación Manchester bifase, donde la transición ocurre siempre en el centro del intervalo de bit, sirviendo simultáneamente para datos y sincronización (reloj).

- **Estructura de la Trama (Mensaje Completo):** La trama estándar del protocolo RC5 es más concisa que la del NEC, no emplea redundancia por inversión lógica y transmite un total de 14 bits estructurados de la siguiente forma:
 1. **Bits de Inicio (Start Bits):** Dos bits lógicos '1' al inicio de la trama que permiten al receptor sincronizarse temporalmente con el emisor y ajustar el AGC.

2. **Bit de Alternancia (*Toggle Bit*):** Un bit que invierte su estado lógico (de 0 a 1, o viceversa) cada vez que el usuario suelta un botón y presiona otro (o el mismo). Es el mecanismo que permite al receptor distinguir entre una presión mantenida continuamente y múltiples pulsaciones discretas rápidas.
3. **Dirección de Sistema (*System Address*):** 5 bits que definen la categoría del dispositivo a controlar (ej. el código 00000 está reservado para Televisores).
4. **Comando (*Command*):** 6 bits de datos que codifican la instrucción a realizar, permitiendo un máximo de 64 acciones distintas por cada dispositivo.

2.2. Fundamentos de Conectividad Bluetooth

En el contexto del diseño del Sistema de Administración Infrarrojo Centralizado (SAIC), surge la necesidad de dotar al dispositivo de una interfaz de configuración dinámica. Dado que el *hardware* del control remoto prescinde de una pantalla o un teclado alfanumérico complejo para mantener un diseño minimalista, la parametrización de las secuencias (macros) debe delegarse a una aplicación móvil externa. Este requerimiento arquitectónico hace indispensable la integración de un canal de comunicación inalámbrica de corto alcance entre el microcontrolador local y el teléfono inteligente (*smartphone*) del usuario. Es bajo esta premisa que se analiza la tecnología Bluetooth, al ser el estándar predominante para redes de área personal (PAN) en la electrónica de consumo.

Bluetooth es un estándar WPAN (Wireless Personal Area Network) que opera en la banda ISM de 2.4 GHz. La elección del perfil de comunicación constituye una decisión crítica de diseño.

2.2.1. Bluetooth Classic vs. Bluetooth Low Energy (BLE)

Existen dos variantes dominantes. BLE (*Bluetooth Low Energy*) es el estándar moderno para IoT, optimizado para transmisiones esporádicas y ultra bajo consumo. Sin embargo, su implementación requiere el manejo de una pila de protocolos asimétrica y compleja (GATT/GAP). Por otro lado, *Bluetooth Classic* (BR/EDR) soporta el perfil SPP (*Serial Port Profile*), que emula una conexión serie UART tradicional.

Para el desarrollo del prototipo SAIC, se optó por el uso de SPP (mediante el módulo HC-05). Si bien BLE ofrece indudables ventajas de eficiencia energética, un factor crítico en dispositivos portátiles, la elección de SPP obedece a criterios de viabilidad y enfoque de desarrollo para esta prueba de concepto. SPP proporciona un enlace transparente (*UART bridge*) que abstrae la complejidad de la comunicación inalámbrica, agilizando el desarrollo del *firmware* y la depuración. Esta decisión de diseño permitió concentrar el esfuerzo de ingeniería en los objetivos primarios de esta memoria: la lógica de control, la persistencia de datos en memoria *Flash* y la síntesis precisa de las tramas IR. El alto impacto energético derivado de mantener el módulo transceptor activo de forma continua es una limitación reconocida del diseño actual, cuya mitigación, mediante la separación de modos de operación y la eventual migración a tecnología BLE, se aborda en las proyecciones de trabajo futuro.

2.3. Accesibilidad en Interfaces Móviles

Aunque la aplicación API MTR-Link está destinada al rol de “administrador”, el diseño de su interfaz sigue principios de usabilidad para minimizar la fricción en la configuración. Según las guías de Material Design, una interfaz efectiva debe minimizar la carga de memoria y prevenir errores. En el sistema SAIC, esto se implementa evitando que el usuario deba manipular códigos hexadecimales; en su lugar, se utilizan listas desplegables con lenguaje natural (ej. “Volumen+”), abstrayendo así la complejidad técnica subyacente.

2.4. Estado del Arte en Soluciones de Control Universal

A continuación se analizan las alternativas de mercado disponibles para la unificación del control remoto.

2.4.1. Controles Universales Programables

Esta categoría incluye desde controles de reemplazo económicos hasta la línea Logitech Harmony [15]. Los modelos básicos dependen de códigos preestablecidos y suelen perder la configuración al agotarse las baterías (memoria volátil). Además, carecen de la capacidad para crear macros complejas de forma intuitiva. Los modelos avanzados que sí permiten macros son costosos y mantienen una alta densidad de botones, por lo que no resuelven el problema de accesibilidad física planteado.

2.4.2. Aplicaciones Móviles y Hubs IR/Wi-Fi

Dispositivos como BroadLink [9] permiten controlar equipos desde el smartphone. Si bien ofrecen alta flexibilidad, introducen la denominada “barrera del smartphone”: desbloquear, navegar y observar la pantalla para ejecutar una acción simple. Esto aumenta la carga cognitiva, lo que los hace inadecuados como interfaz primaria para el adulto mayor. La Tabla 2.1 presenta un análisis comparativo que resume las ventajas de la propuesta SAIC frente a estas alternativas.

Tabla 2.1: Comparación cualitativa entre soluciones de control existentes y el sistema SAIC.

Característica	Control Original	Universal (Básico)	App Móvil	SAIC (Propuesto)
Interfaz Física	Botones Pequeños	Botones Medios	Pantalla Táctil	Botones Grandes
Configurabilidad	Nula	Códigos Manuales	Alta	Alta (App)
Macros	No	No	Sí (Multi-Paso)	Sí
Feedback Mecánico	Sí	Sí	No	Sí
Dependencia Smartphone	No	No	Total	Solo Configuración

2.5. Justificación de la Solución Propuesta

El análisis previo revela una oportunidad no cubierta: la inexistencia de una solución que combine la **simplicidad extrema** de un control físico con la **potencia de configuración** del software moderno.

El diseño del sistema SAIC se fundamenta en la disociación de estas interfaces:

1. **Interfaz de Usuario (Hardware):** Un control robusto, sin pantallas y con un número reducido de botones, que elimina la complejidad operativa. Se utilizan componentes discretos para asegurar un alcance comercial ($> 5\text{m}$).
2. **Interfaz de Administración (Software):** Una App Android que permite configurar macros complejas (ej. “Canal 731”) y cargarlas al hardware vía Bluetooth. Esto traslada la complejidad del “cómo hacerlo” al software, dejando al usuario final únicamente la decisión de “qué hacer”.

Capítulo 3

Análisis y Arquitectura de SAIC

En este capítulo se definen formalmente los requerimientos del sistema SAIC, derivados del análisis de la problemática y el estado del arte presentados en los capítulos precedentes. A partir de estos requerimientos, se diseña y presenta la arquitectura de alto nivel de la solución, identificando los módulos funcionales, las interfaces de comunicación y la selección tecnológica debidamente justificada.

3.1. Definición de Requerimientos de SAIC

Para garantizar el éxito del proyecto, se han establecido requerimientos funcionales y no funcionales que guiarán el proceso de diseño e implementación.

3.1.1. Requerimientos Funcionales (RF)

Requisitos Funcionales (RF)

Los requisitos funcionales definen las acciones específicas y el comportamiento lógico que el sistema debe ejecutar.

- **RF1 - Emisión Multi-Protocolo:** El sistema debe ser capaz de sintetizar y emitir señales infrarrojas moduladas compatibles, como mínimo, con los estándares industriales NEC y RC5, operando a frecuencias portadoras de 38 kHz y 36 kHz respectivamente.
- **RF2 - Procesamiento de Configuración Remota:** El sistema debe ser capaz de recibir, decodificar y procesar tramas de configuración de manera inalámbrica desde una aplicación móvil externa, permitiendo la actualización de sus parámetros en tiempo de ejecución.
- **RF3 - Almacenamiento y Recuperación de Estado:** El *firmware* del microcontrolador debe escribir (guardar) la estructura de datos que compone las macros en un sector dedicado de la memoria no volátil (*Flash*). Asimismo, durante cada ciclo de encendido (*Boot*), el sistema debe leer (cargar) automáticamente esta información en la memoria RAM para garantizar que la configuración operativa no se pierda ante cortes de energía.

- **RF4 - Integridad de Datos:** El protocolo de comunicación debe implementar mecanismos de validación (*Acknowledge*) para asegurar que la configuración recibida por el *hardware* es estructuralmente correcta antes de autorizar su escritura en memoria.
- **RF5 - Funcionalidad de Macros:** El *firmware* debe soportar la definición y ejecución de “macros”, conceptualizadas como secuencias lógicas de hasta 4 comandos IR consecutivos emitidos de forma secuencial tras un único estímulo del usuario.
- **RF6 - Base de Datos de Códigos (App):** La aplicación móvil debe proveer una base de datos precargada y seleccionable de comandos IR para los fabricantes de TV más comunes (ej. Samsung, LG), abstrayendo al usuario de la complejidad de los tiempos lógicos subyacentes.
- **RF7 - Indicación de Estado Operativo:** El sistema debe notificar al usuario sobre cambios críticos en su estado interno (ej. transmisión IR en curso, éxito/fallo en la recepción de configuración) mediante un canal de comunicación visual.

Requisitos No Funcionales (RNF)

Los requerimientos no funcionales definen los criterios de calidad, rendimiento y las restricciones de diseño del sistema:

- **RNF1 - Alcance Efectivo:** El subsistema emisor IR debe garantizar una transmisión de datos exitosa a una distancia mínima de 5.0 metros en línea de vista directa (LoS) con el receptor.
- **RNF2 - Usabilidad de Administración:** La interfaz de la aplicación móvil debe permitir al administrador reconfigurar la función de un botón físico en un máximo de 3 pasos lógicos una vez establecida la conexión.
- **RNF3 - Latencia de Respuesta:** El tiempo transcurrido entre la solicitud física de ejecución de una macro y el inicio real de la emisión de la ráfaga IR no debe exceder los 200 ms, asegurando una percepción de respuesta en tiempo real.
- **RNF4 - Eficiencia Energética:** Para el prototipo de validación, se exige una autonomía operativa mínima de 24 horas en régimen continuo, sentando las bases analíticas para un diseño escalable de bajo consumo en iteraciones futuras.
- **RNF5 - Restricciones de Interfaz Física:** Para mantener un diseño minimalista y de bajo costo, la interacción física local del usuario con el hardware debe estar estrictamente limitada a pulsadores mecánicos y un único indicador luminoso (LED). Cualquier parametrización compleja debe delegarse invariablemente a la aplicación móvil.

3.2. Análisis de Casos de Uso

Se han identificado dos flujos de interacción principales que dictan el comportamiento del sistema: el flujo de administración y el flujo de operación.

3.2.1. Caso de Uso 1: Configuración de Botón

Actor: Administrador (Familiar/Cuidador).

Descripción: El administrador utiliza la App IR-Link para asignar una función específica a un botón físico del SAIC.

1. El administrador inicia la App y establece conexión Bluetooth con el SAIC.
2. Selecciona la marca del televisor (ej. Samsung) desde la lista predefinida.
3. Selecciona la representación gráfica del botón a configurar en la interfaz móvil.
4. Selecciona el comando (ej. Power) desde la lista predefinida.
5. La App envía la trama de configuración (ej. SET:BTN1:NEC:E0E040BF) al hardware.
6. El hardware recibe, valida la trama y almacena temporalmente la configuración en RAM, respondiendo con una confirmación (SET_OK).
7. Tras confirmar los cambios, el hardware persiste los datos en la memoria Flash.

3.2.2. Caso de Uso 2: Operación Diaria

Actor: Usuario Final (Adulto Mayor).

Descripción: El usuario utiliza el dispositivo para controlar su televisor.

1. El usuario presiona un botón físico en el teclado matricial.
2. El firmware consulta la memoria Flash para recuperar el protocolo y el código (o secuencia de códigos) asociados.
3. Se sintetiza y emite la señal IR modulada a través del LED de potencia.
4. Finalizada la transmisión, el sistema retorna automáticamente al estado de reposo.

3.3. Selección de Tecnología

La selección del microcontrolador (MCU) es una decisión de diseño crítica. Se evaluaron tres arquitecturas considerando el equilibrio entre determinismo en tiempo real, eficiencia energética y capacidad de periféricos, cuyos resultados se consolidan en la Tabla 3.1.

Justificación de la Selección: Se descartó la plataforma **Arduino Uno** debido a las limitaciones de sus temporizadores de 8 bits, insuficientes para generar con precisión las bases de tiempo largas requeridas por protocolos lentos sin recurrir a Preescaladores complejos que degradan la resolución.

Respecto al **ESP32**, aunque ofrece conectividad nativa, se descartó por dos factores:

Tabla 3.1: Matriz de decisión para la selección del microcontrolador.

Criterio	Arduino Uno (ATmega328P)	ESP32 (Espressif)	STM32L476RG (ST Microelectronics)
Arquitectura	8-bit AVR	32-bit Xtensa (Dual)	32-bit ARM Cortex-M4
Frecuencia	16 MHz	240 MHz	80 MHz
Consumo MCU (Activo)	~15 mA	~60-80 mA	< 10 mA
Determinismo	Alto (Bare metal)	Medio (RTOS overhead)	Alto (Interrupciones)
Periféricos Timers	Básicos (8/16-bit)	PWM Software/Hardware	Avanzados (32-bit)

1. **Eficiencia Energética:** Su consumo base en modo activo (incluso con radio apagada) es significativamente superior al de la familia STM32L4, lo cual penaliza la autonomía en un dispositivo alimentado por batería.
2. **Determinismo Real-Time:** La arquitectura del ESP32 opera típicamente sobre un sistema operativo de tiempo real (FreeRTOS). Para la generación de señales IR, donde la estabilidad del *jitter* (variación de tiempo) debe ser menor a 10 μ s, el control directo de hardware ("Bare Metal") que ofrece el STM32 con su controlador de interrupciones anidado (NVIC) garantiza una síntesis de señal más limpia y predecible.

Por lo tanto, se seleccionó el **STM32L476RG**. Su núcleo Cortex-M4 con unidad de punto flotante (FPU) y sus temporizadores de 32 bits permiten una implementación robusta de la máquina de estados del firmware, manteniendo un consumo de corriente mínimo ideal para la aplicación.

3.4. Arquitectura General del Sistema

La solución SAIC propone una arquitectura híbrida Hardware-Software, donde la complejidad de la configuración se delega a una aplicación móvil, manteniendo el hardware de operación (usuario final) lo más simple y robusto posible. La Figura 3.1 esquematiza la interacción entre los módulos de hardware y software.

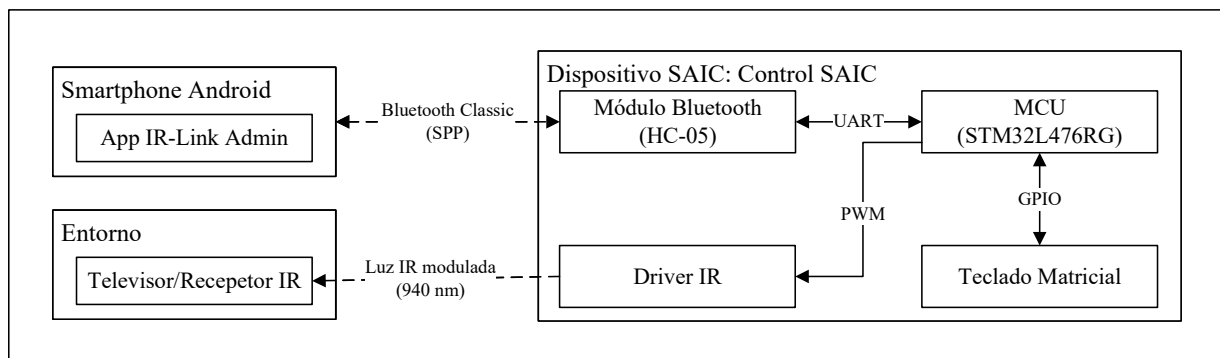


Figura 3.1: Arquitectura general del sistema SAIC, detallando los módulos de software y hardware.

El diseño se estructura en tres capas lógicas:

3.4.1. Capa de Aplicación (Android - Admin)

Constituye la interfaz de configuración (Front-End Administrativo). Gestiona la lógica de negocio relacionada con la base de datos de códigos IR (JSON) y la Interfaz de Usuario (UI). Es responsable de la serialización de comandos y la gestión del socket Bluetooth (RFCOMM) para la transmisión de datos.

3.4.2. Capa de Control (Firmware STM32)

Es el núcleo operativo del sistema. Sus submódulos principales son:

- **Gestor de Comunicaciones:** Implementa una máquina de estados para el parseo y validación de tramas UART provenientes del módulo Bluetooth.
- **Controlador de Memoria:** Abstrae las operaciones de bajo nivel (borrado de página y programación) en la memoria Flash interna para asegurar la persistencia.
- **Sintetizador IR:** Módulo crítico que utiliza los temporizadores de hardware (TIM) y gestión por interrupciones para generar la portadora (38/36 kHz) y modular los anchos de pulso con precisión de microsegundos, asegurando la compatibilidad con receptores comerciales.

3.4.3. Capa Física (Hardware)

Comprende los componentes electrónicos tangibles:

- **Driver de Potencia IR:** Etapa de amplificación basada en transistores BJT (Bipolar Junction Transistor) diseñada para excitar el LED emisor (TSAL6200) con corrientes de pico altas (> 100 mA) necesarias para cumplir el requerimiento de alcance (RNF1).
- **Matriz de Botones:** Interfaz de entrada física de alta durabilidad.
- **Módulo Bluetooth (HC-05):** Interfaz de comunicación serie inalámbrica SPP (Serial Port Profile).

3.5. Diagramas de Secuencia del Sistema

Para validar la lógica de interacción y el cumplimiento del RF5 (Integridad), se presenta el diagrama de secuencia para el proceso de configuración en la Figura 3.2.

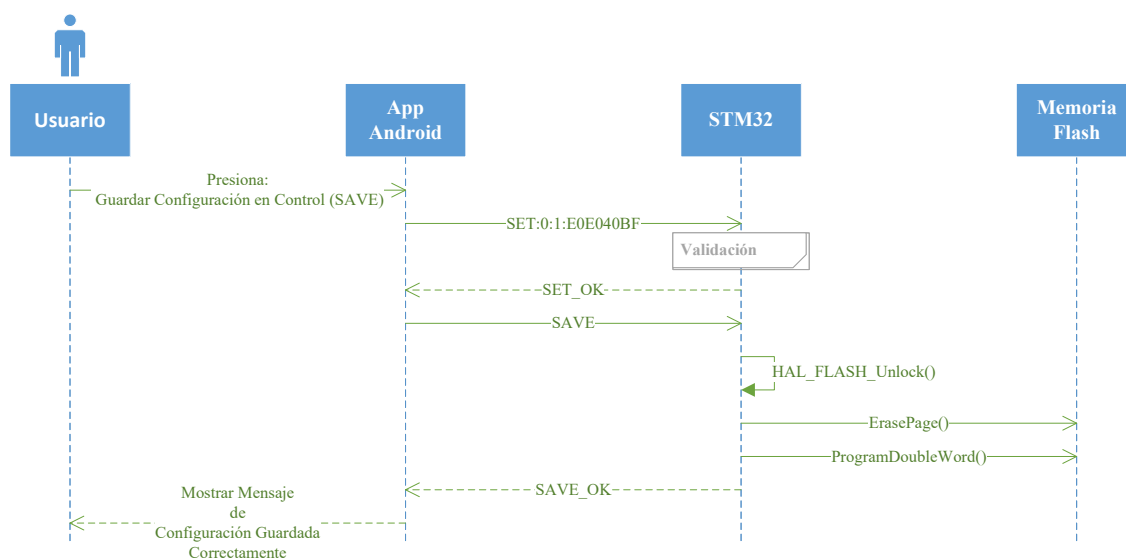


Figura 3.2: Diagrama de secuencia detallado para el almacenamiento de una nueva configuración, mostrando el flujo de confirmación (ACK).

Este flujo garantiza la integridad de los datos, ya que el hardware valida la sintaxis de la trama antes de proceder con el almacenamiento persistente, evitando la corrupción de la memoria Flash con datos inválidos.

Capítulo 4

Diseño Detallado de SAIC

Este capítulo profundiza en el diseño técnico de la solución SAIC, describiendo la arquitectura interna de cada subsistema. Se detalla la selección de componentes de hardware, justificando cada elección mediante cálculos y criterios de ingeniería; la estructura modular del firmware, incluyendo diagramas de flujo y máquinas de estados; el protocolo de comunicación diseñado y la arquitectura de software de la aplicación móvil.

4.1. Metodología y Evolución del Diseño

El diseño del sistema SAIC no fue un proceso lineal, sino iterativo. Se evaluaron múltiples alternativas para cada subsistema, refinando la solución en base a pruebas empíricas y restricciones de hardware.

4.1.1. Iteraciones de Hardware

Inicialmente, se intentó controlar el LED IR directamente desde un pin GPIO del microcontrolador (**Iteración 1**). Sin embargo, esta aproximación simple resultó ineficaz, ya que las pruebas mostraron un alcance insuficiente (menor a 1 metro) debido a la limitación de corriente de 20mA del puerto. Posteriormente, en la **Iteración 2**, se incorporó un transistor 2N2222 en configuración de emisor común, lo que permitió aumentar la corriente de excitación y lograr el alcance objetivo de 5 metros, el esquemático de esta configuración se aprecia en la Figura 4.1. Finalmente, se evaluó una **Iteración 3** utilizando un par Darlington para obtener mayor ganancia, pero fue descartada debido a su mayor caída de tensión de saturación ($V_{CE(sat)}$), la cual reducía el voltaje efectivo sobre el LED en el sistema de baja tensión (3.3V).

4.1.2. Decisiones de Arquitectura de Firmware

Para el desarrollo del firmware, se debió elegir entre el uso de la capa de abstracción HAL (Hardware Abstraction Layer) o LL (Low-Layer) de ST. Se optó por **HAL** a pesar de su mayor consumo de memoria, debido a que acelera significativamente el desarrollo y facilita la portabilidad entre diferentes MCUs de la familia STM32. En cuanto al modelo de ejecución, se descartó un enfoque secuencial (polling) en favor de una arquitectura **basada en interrupciones y eventos**. Esta decisión es crucial para garantizar que la

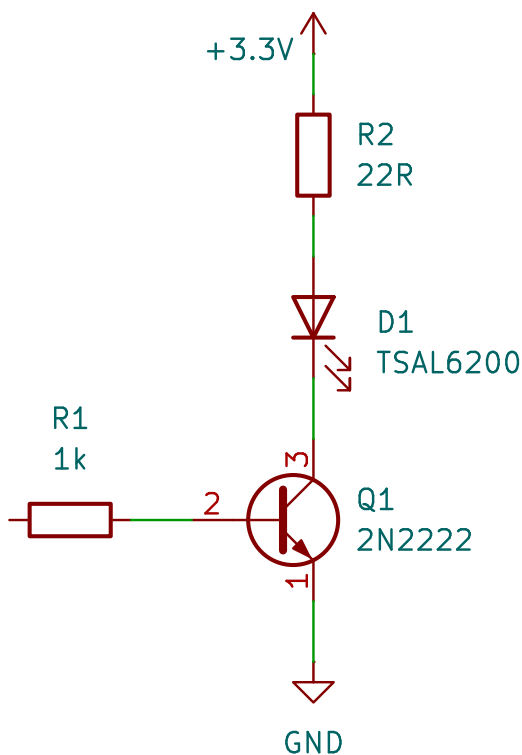


Figura 4.1: Diagrama esquemático de la Etapa de Potencia.

generación de la señal IR, que requiere precisión de microsegundos, no sea interrumpida por tareas de menor prioridad como la gestión de la interfaz Bluetooth.

4.1.3. Justificación del Protocolo MTR-Link

Se diseñó un protocolo basado en texto (ASCII) en lugar de un formato binario compacto. La principal **ventaja** de este enfoque es que permite la depuración directa utilizando cualquier terminal serial estándar sin necesidad de herramientas de decodificación personalizadas. Aunque esto conlleva la **desventaja** de un mayor uso de ancho de banda, el impacto es despreciable dado que los comandos de configuración son esporádicos y cortos (menos de 100 bytes), justificándose plenamente por la ganancia en mantenibilidad y facilidad de “debugging”.

4.2. Diseño del Hardware de SAIC: Control SAIC

El hardware del Control SAIC ha sido diseñado priorizando la robustez, la simplicidad de integración y la eficiencia energética.

4.2.1. Selección de Componentes

- **Microcontrolador (MCU):** Se seleccionó el **STM32L476RG** (familia ARM Cortex-M4). Esta familia está optimizada para ultra-bajo consumo, ofreciendo modos como “Stop 2” con consumo de $1.6 \mu\text{A}$, ideal para dispositivos a batería. Además, sus temporizadores avanzados (Timers) permiten generar señales PWM precisas sin carga excesiva de CPU.
- **Módulo Bluetooth:** Se utiliza el módulo **HC-05** configurado en modo esclavo. Aunque es un módulo basado en Bluetooth 2.0 (Classic), su interfaz UART transparente simplifica enormemente el desarrollo del firmware al abstraer la complejidad de la pila RF, permitiendo centrar el esfuerzo en la lógica de control.
- **Emisor Infrarrojo:** Se eligió el LED IR **Vishay TSAL6200**. Posee una longitud de onda pico de **940 nm**, que coincide con la sensibilidad espectral máxima de los receptores comerciales (evitando los 850 nm visibles). Su alta intensidad radiante (40 mW/sr a 100 mA) es clave para garantizar el alcance.
- **Driver de Potencia:** Transistor BJT **2N2222**. Dado que los pines GPIO del microcontrolador STM32L476RG presentan una limitación física de corriente de **20 mA** (corriente máxima de *source/sink*), resulta imposible excitar el LED IR directamente para alcanzar distancias superiores a un metro. Se implementó una etapa de potencia utilizando el 2N2222 ($I_{C,max} = 800 \text{ mA}$) en configuración de **interruptor de lado bajo** (*Low-side switch*). Esta topología permite que el GPIO controle únicamente la corriente de base (en el orden de los microamperios), delegando la carga de potencia del LED al colector del transistor, garantizando así la integridad del microcontrolador y la potencia de emisión necesaria.

4.2.2. Diseño del Circuito (Esquemático)

El diseño eléctrico interconecta los módulos, como se detalla en el diagrama esquemático de la Figura [4.2](#)

Cálculo de la Etapa de Potencia

Para garantizar la saturación del transistor y la corriente operativa del LED, se realizaron los siguientes cálculos de diseño:

1. **Corriente de Colector Objetivo (I_C):** Se busca una corriente de pico $I_C \approx 75 - 80 \text{ mA}$ para maximizar la emisión del TSAL6200 sin exceder sus límites térmicos en modo pulsado.
2. **Resistencia de Base (R_B):** Considerando una ganancia mínima (β) de 30 para el 2N2222, la corriente de base necesaria es $I_B \approx 2,5 \text{ mA}$.

$$I_B = \frac{V_{GPIO} - V_{BE}}{R_B}$$

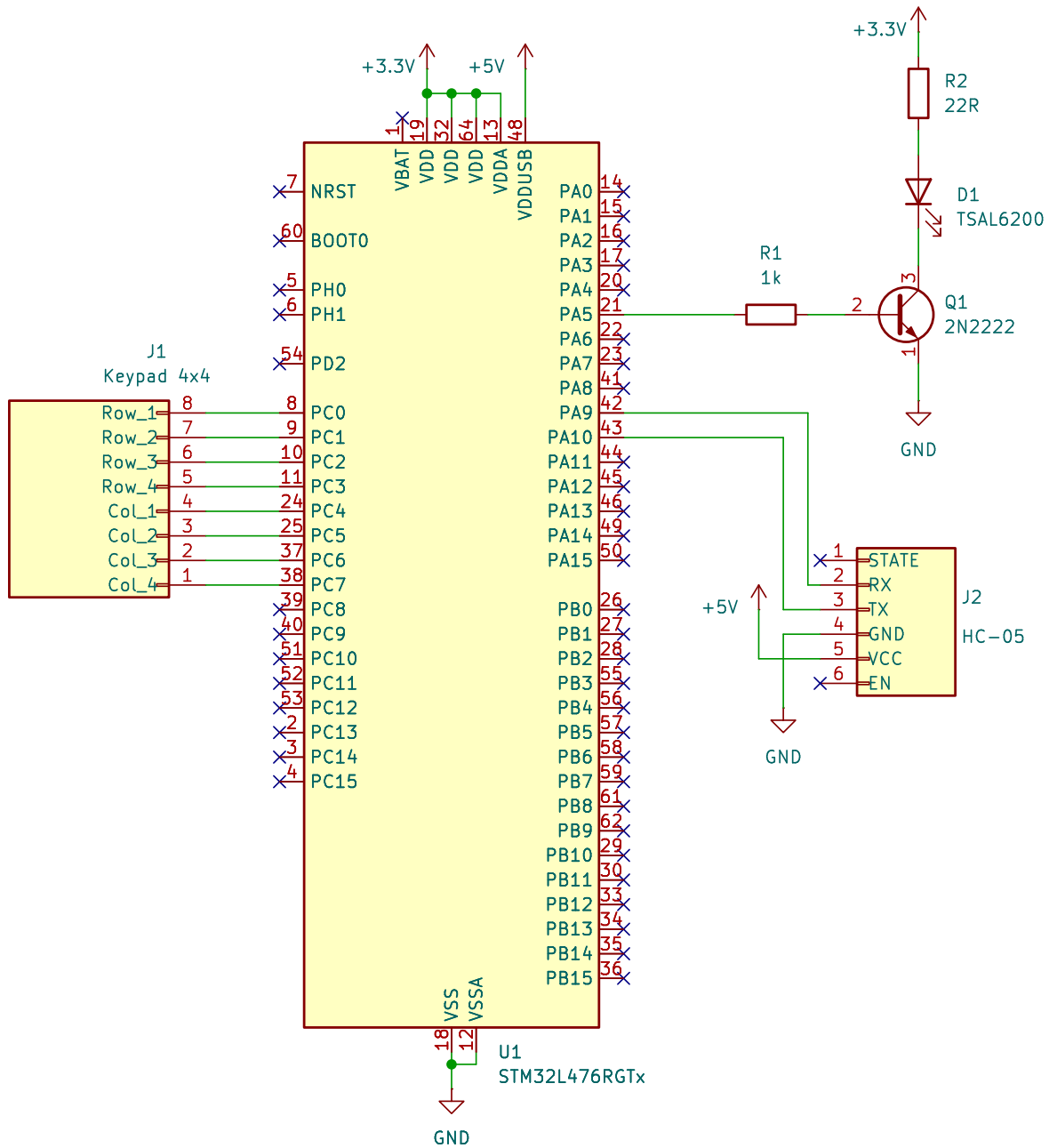


Figura 4.2: Diagrama esquemático del hardware de control SAIC, mostrando las conexiones entre MCU, Bluetooth, Teclado y Etapa de Potencia.

Se seleccionó un valor comercial de **1 kΩ**.

$$I_{B,real} = \frac{3,3V - 0,7V}{1000\Omega} = 2,6 \text{ mA}$$

Esta corriente es suficiente para saturar el transistor dado que la ganancia típica del 2N2222 suele ser superior a 30, asegurando la conmutación correcta.

3. Resistencia Limitadora de LED (R_L):

$$R_L = \frac{V_{CC} - V_{CE(sat)} - V_{F(LED)}}{I_C} \approx \frac{3,3V - 0,3V - 1,35V}{0,075A} \approx 22\Omega$$

Se seleccionó una resistencia comercial de **22 Ω** . Con este valor, la corriente real estimada se mantiene en el rango seguro de **75 mA**, lo cual ofrece un margen de seguridad adecuado y un alcance verificado superior a 5 metros.

Previsión para Alimentación Autónoma y Gestión de Energía

Para la fase de prototipado funcional descrita en esta memoria, el sistema se alimenta mediante el puerto USB (5V), utilizando el regulador lineal de la placa Nucleo para obtener los 3.3V constantes. Sin embargo, el diseño electrónico del Control SAIC se ha proyectado para permitir una transición sencilla hacia una fuente de alimentación autónoma basada en dos baterías AA.

Desde una perspectiva de ingeniería, el uso de baterías químicas presenta el desafío de la descarga no lineal; una celda AA nominal de 1.5V puede decaer hasta los 1.0V antes de agotarse. Por lo tanto, un arreglo en serie de dos celdas (2.0V - 3.0V) resultaría insuficiente para mantener la estabilidad del microcontrolador y el módulo Bluetooth a 3.3V.

Para solventar esta limitación en futuras iteraciones del *hardware*, se ha previsto la incorporación de un **Convertor DC-DC Step-Up (Boost)**. La inclusión de este componente permitiría:

- **Sustentabilidad de la Tensión:** Elevar y estabilizar el voltaje variable de las baterías a un nivel constante de 3.3V durante todo su ciclo de vida.
- **Invariabilidad del Alcance:** Garantizar que la potencia de emisión del LED IR y la precisión de la portadora no se degraden a medida que la batería se agota, manteniendo el rendimiento de 5 metros de alcance de forma consistente.

El esquema actual de la etapa de potencia y el teclado es totalmente compatible con esta arquitectura, requiriendo únicamente la adición del módulo convertor entre la fuente de energía y el bus de alimentación principal.

4.3. Diseño del Firmware de SAIC: Control SAIC

El firmware del sistema SAIC se ha concebido bajo un paradigma de control de tiempo real, donde la precisión en la generación de pulsos infrarrojos es la prioridad absoluta. A diferencia de arquitecturas secuenciales, este diseño delega las tareas críticas de temporización a los periféricos internos del microcontrolador, permitiendo que la lógica de control principal (gestión de macros y comunicación Bluetooth) opere de forma asíncrona y sin interferir con la integridad de las tramas IR. La arquitectura se estructura en una jerarquía de capas que abstrae el hardware mediante la librería HAL, facilitando la mantenibilidad y permitiendo una gestión eficiente de los recursos del sistema.

4.3.1. Síntesis de Portadoras mediante PWM por Hardware

Antes de abordar la lógica modular del software, es imperativo definir cómo se genera la capa física de la señal. Para garantizar una precisión de microsegundos sin saturar la CPU con bucles de retardo, las portadoras de 36 kHz y 38 kHz se generan mediante el periférico **Timer 2** configurado en modo *Pulse Width Modulation* (PWM).

La señal producida es una **onda cuadrada pura** cuya estabilidad depende directamente del reloj del sistema (f_{CLK}), el cual se fijó en 80 MHz. El *firmware* configura dinámicamente el registro de auto-recarga (ARR) del temporizador para establecer la frecuencia de oscilación según el protocolo seleccionado:

$$ARR = \left(\frac{f_{CLK}}{f_{PWM} \times (PSC + 1)} \right) - 1 \quad (4.1)$$

Bajo esta arquitectura, el sistema conmuta entre los dos estándares implementados, el estándar NEC y el estándar RC5.

- **Estándar NEC (38 kHz):** Con un PSC de 0, el valor de ARR es ≈ 2104 . Se utiliza un *Duty Cycle* de 1/3 (CCR ≈ 701), lo que genera un pulso activo de $8,7\mu s$.
- **Estándar RC5 (36 kHz):** El sistema actualiza el registro ARR a ≈ 2221 . En este caso, para mantener la compatibilidad con el estándar de Philips, se ajusta el *Duty Cycle* al 50% (CCR ≈ 1110), resultando en un pulso activo de $13,8\mu s$.

Esta capacidad de reconfiguración en tiempo de ejecución permite que una sola etapa de salida infra-roja sea compatible con una amplia gama de dispositivos, optimizando el uso de recursos del microcontrolador.

El firmware se ha diseñado con una arquitectura modular basada en eventos para garantizar la capacidad de respuesta en tiempo real (Real-Time), cuya estructura modular se ilustra en la Figura 4.3.

4.3.2. Arquitectura del Software de Firmware

El código se organiza en módulos funcionales independientes:

- **Generador IR:** Gestiona la generación física de la señal. Controla los Timers para sintetizar la portadora y la temporización precisa de pulsos/espacios.
- **Teclado:** Realiza el barrido de la matriz de botones e implementa algoritmos de eliminación de rebotes por software para evitar falsos positivos.
- **Almacenamiento:** Abstrae las operaciones de bajo nivel (HAL_FLASH) para la lectura y escritura persistente en la memoria Flash.
- **UART Parser:** Interpreta los comandos ASCII recibidos por UART y despacha las acciones de configuración.
- **Lógica Principal:** Actúa como orquestador, ejecutando el bucle principal y la máquina de estados de macros.

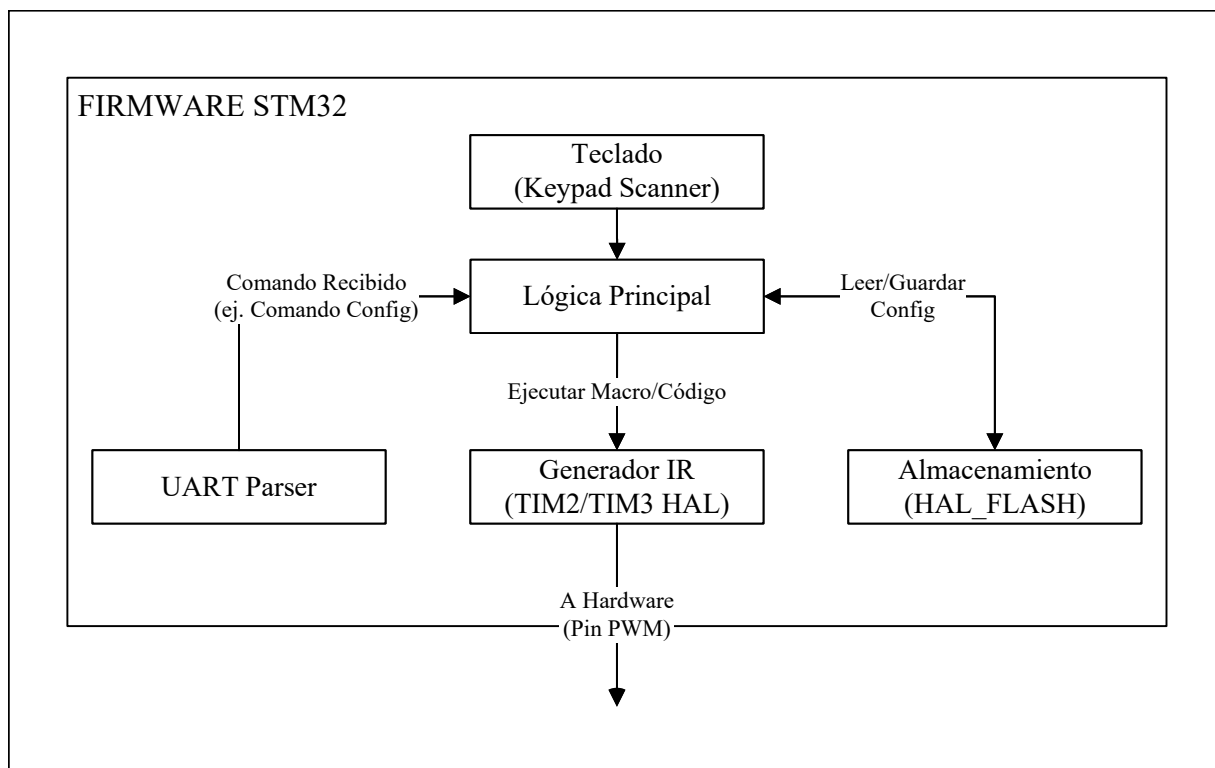


Figura 4.3: Diagrama de bloques de la arquitectura del firmware.

4.3.3. Máquina de Estados de Envío (Macros)

Para la gestión de macros (secuencias de comandos), se implementó una máquina de estados finitos (FSM) de alto nivel ejecutada en el bucle principal (`main loop`).

Para garantizar la robustez del sistema y evitar condiciones de carrera, se optó por un diseño síncrono desacoplado mediante banderas. A diferencia de una implementación donde las interrupciones fuerzan cambios de estado, en este diseño:

1. Los eventos externos (interrupción por pulsación de teclado o comando UART) levantan una bandera: `macro_event_solicited = true` e inicializan los punteros de la macro.
2. La máquina de estados, en su ciclo ocioso (`MACRO_IDLE`), evalúa esta bandera y decide transicionar de forma determinista. Durante esta transición, la FSM ejecuta el primer comando IR del macro (`macro_codes_ptr[0]`), consumiendo el evento y pasando al estado de espera (`MACRO_WAIT_IR_FINISH`).

La Figura 4.4 ilustra el diagrama de estados implementado, Nótese el diseño no bloqueante, la transición ocurre por evaluación de bandera y los retardos se gestionan comparando el tiempo del sistema (`HAL_GetTick`), permitiendo que el microcontrolador atienda otras tareas durante la espera.

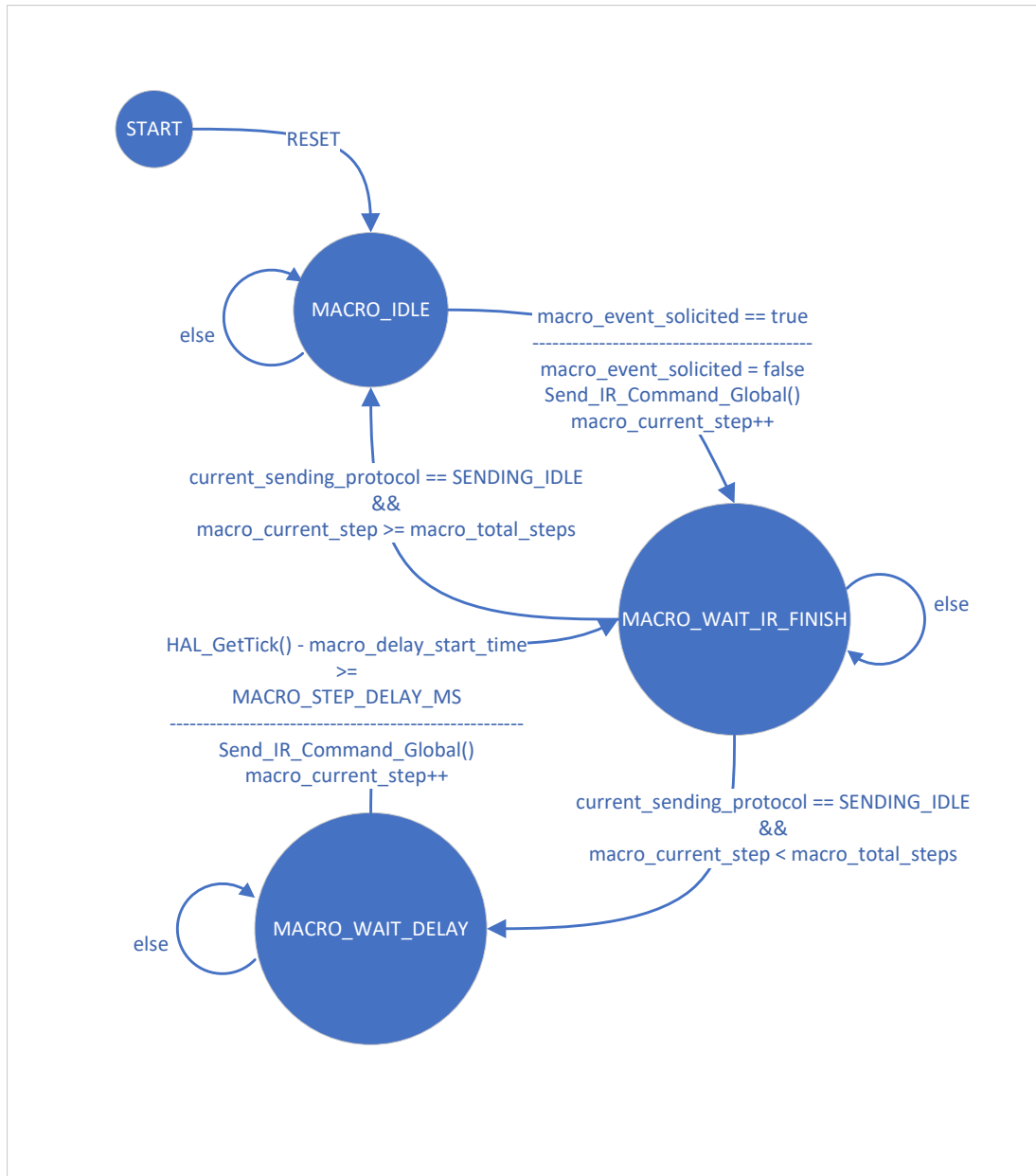


Figura 4.4: Diagrama de estados de Macros.

Los estados definidos son MACRO_IDLE, MACRO_WAIT_IR_FINISH y MACRO_WAIT_DELAY.

- **MACRO_IDLE:** Estado de reposo. Espera activamente la bandera de solicitud. Al detectarla, inicia el contador de pasos.
- **MACRO_WAIT_IR_FINISH:** Sincronización con el hardware. La máquina de alto nivel se pausa mientras el driver de bajo nivel (IR) está ocupado transmitiendo (`protocol != IDLE`).
- **MACRO_WAIT_DELAY:** Implementa el retardo entre comandos (por defecto 150ms) necesario para que el receptor (TV) procese la señal.

4.3.4. Máquinas de Estados de Bajo Nivel (Drivers IR)

Para cumplir con los estrictos requisitos temporales de los protocolos infrarrojos, donde la precisión requerida es del orden de microsegundos, se diseñó una arquitectura asíncrona basada en interrupciones de hardware (TIM3).

El driver implementa dos estrategias de control distintas, seleccionadas dinámicamente según la naturaleza del protocolo a transmitir, usando una máquina de estados para el protocolo NEC y una máquina de fases para el protocolo RC5.

Driver NEC (Máquina de Estados Secuencial)

El protocolo NEC utiliza una codificación por distancia de pulsos. Para generarlo, se utiliza una máquina de estados explícita controlada por la variable `ir_state`. Cada transición es disparada por la interrupción del Timer al finalizar el periodo anterior, configurando inmediatamente el tiempo para el siguiente estado.

La Figura 4.5 detalla la lógica interna del driver NEC. El flujo es impulsado exclusivamente por interrupciones del Timer (TIMER_ISR). Nótese la bifurcación lógica en `IR_SENDING_BIT_ON` que determina la duración del silencio para codificar un '0' o un '1'.

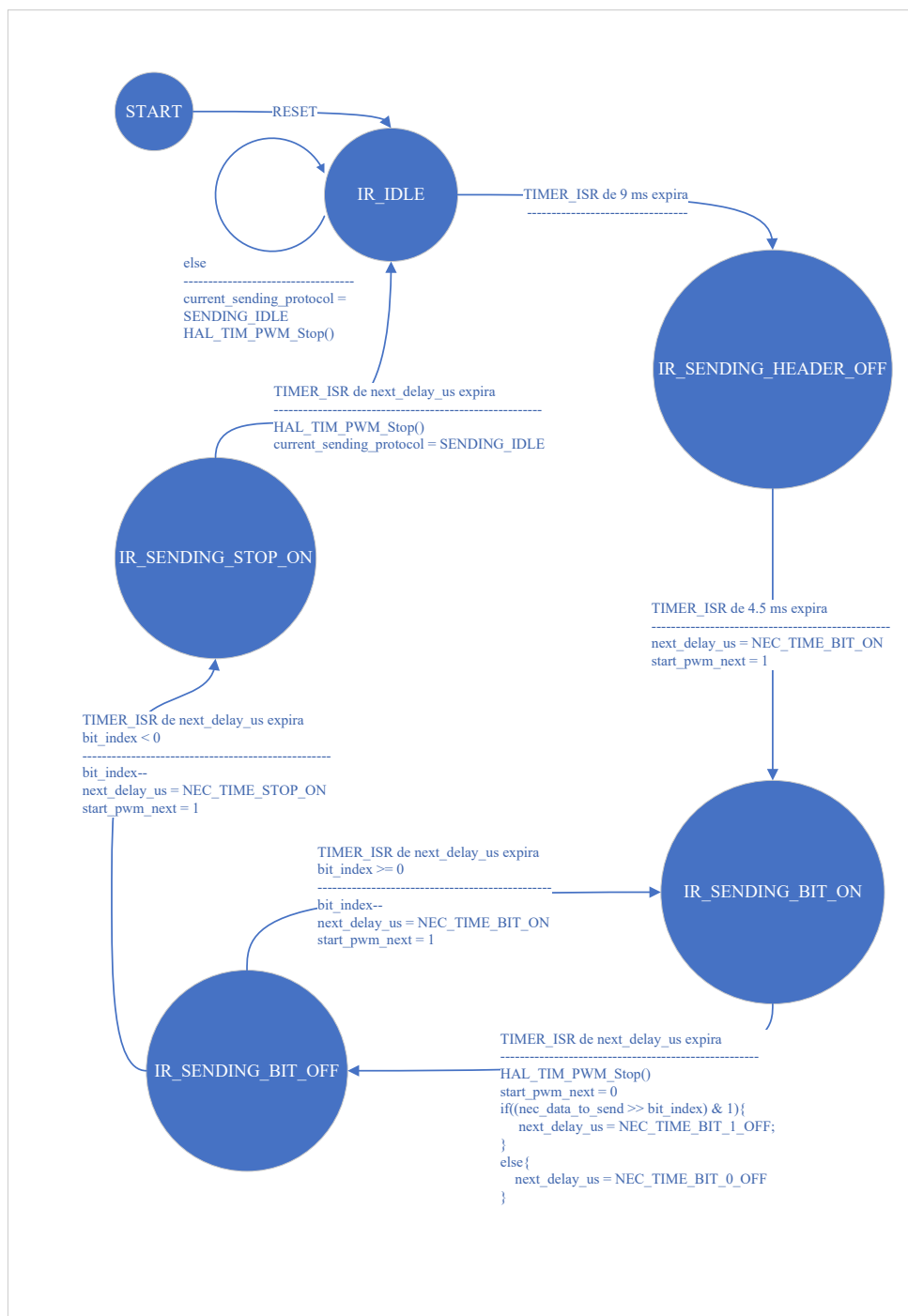


Figura 4.5: Máquina de Estados del driver NEC (Bajo Nivel).

El ciclo de transmisión consta de tres fases críticas gestionadas por esta FSM, Preámbulo, Datos y Finalización.

1. **Preámbulo (Header):** La ráfaga inicial de 9ms (Header ON) es iniciada directamente por la función de envío (`Send_NEC_Code`). La FSM toma control cuando la interrupción del Timer finaliza este pulso, llevando el estado de `IR_IDLE` a `IR_SENDING_HEADER_OFF` para generar el silen-

cio de 4.5ms.

2. **Serialización de Datos:** Un bucle que itera sobre los 32 bits de datos. La lógica decide la duración del estado OFF ($562\mu s$ para bit 0, 1.68ms para bit 1) en tiempo real.
3. **Finalización:** Generación del bit de parada y retorno a IDLE.

Driver RC5 (Máquina de Fases Manchester)

A diferencia del protocolo NEC, el estándar RC5 utiliza codificación Manchester (bifásica), lo que permite que la estrategia de control sea reemplazada por una máquina de fases (`rc5_phase`) de tiempo constante, ilustrada en la Figura 4.6.

- **Temporización Fija:** El Timer se configura con un periodo estricto de $889\mu s$, correspondiente exactamente a la mitad de la duración de un bit. Esto elimina la necesidad de recalcular el registro ARR en cada interrupción, reduciendo la fluctuación temporal (*jitter*).
- **Lógica de Conmutación:** En cada interrupción, el sistema evalúa el bit actual y el estado de la fase:
 - En la **Fase 0** (inicio del bit), el LED se enciende para un bit '1' y se apaga para un bit '0'.
 - En la **Fase 1** (mitad del bit), el sistema invierte obligatoriamente el estado del LED.

Esta transición garantizada en el centro de cada bit es la que permite al receptor sincronizar su reloj interno con la transmisión del SAIC.

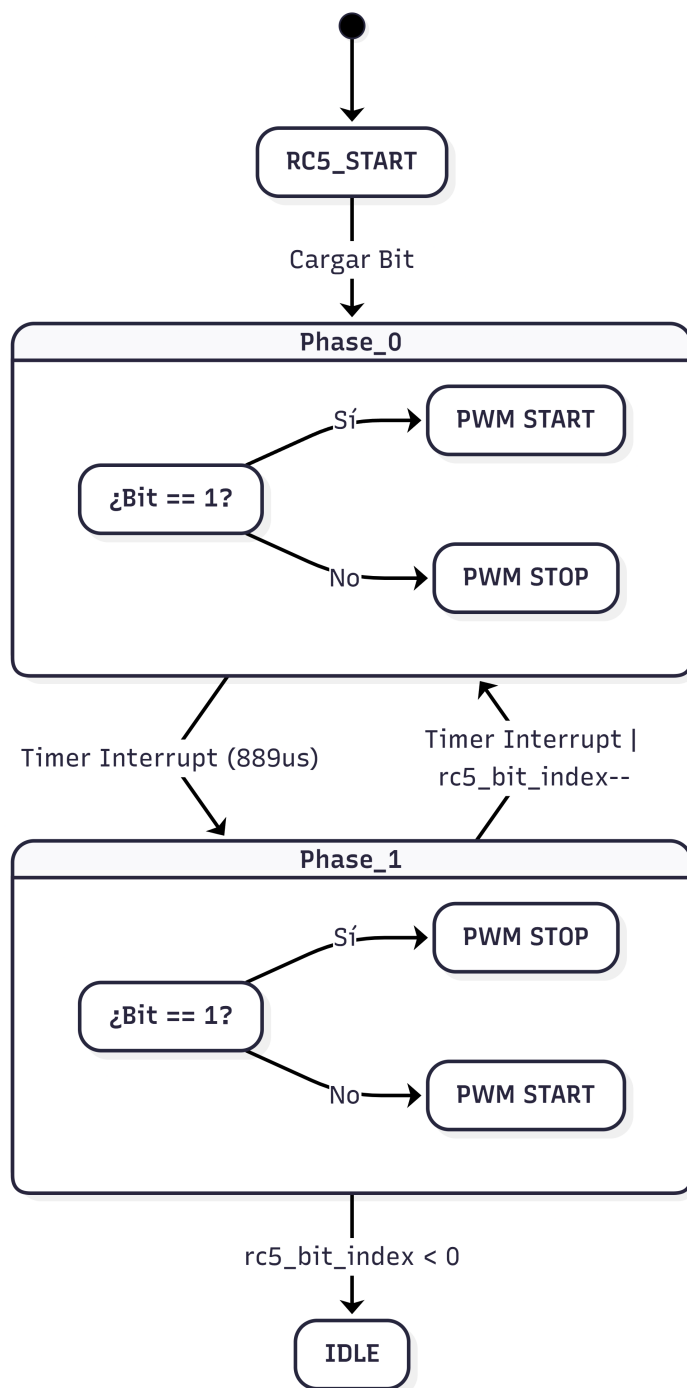


Figura 4.6: Máquina de fases del driver RC5 (Bajo Nivel). Se observa la estructura cíclica y simétrica de la transmisión Manchester.

4.3.5. Gestión Dinámica de Protocolos y Reconfiguración de Registros

La versatilidad del sistema SAIC para operar con múltiples estándares radica en la capacidad del *firmware* para reconfigurar sus periféricos de *hardware* en tiempo de ejecución. Como se fundamentó

analíticamente en la Sección 4.3.1, las variaciones entre los protocolos NEC y RC5 no se limitan a la lógica de bits, sino que afectan directamente a la capa física de la señal portadora.

Para gestionar este cambio sin requerir un reinicio del sistema, se implementó una función de abstracción que actualiza los registros críticos del temporizador TIM2 cada vez que el usuario selecciona un nuevo fabricante en la aplicación móvil. El proceso de conmutación sigue la siguiente lógica:

1. **Carga de Parámetros:** Al recibir el comando SET_PROTO vía UART, el *parser* identifica el identificador del protocolo y recupera de una tabla de búsqueda (*Look-up Table*) los valores de ARR y CCR calculados previamente.
2. **Acceso Directo a Registros:** Para garantizar una transición atómica, el *firmware* escribe directamente en los registros de control del temporizador:
 - TIM2→ARR: Actualiza el periodo total para conmutar entre 36 kHz y 38 kHz.
 - TIM2→CCR1: Ajusta el ciclo de trabajo (1/3 para NEC o 1/2 para RC5), modificando la potencia promedio de emisión y la compatibilidad del receptor.
3. **Sincronización de Fase:** El cambio se realiza asegurando que el temporizador complete su ciclo actual antes de aplicar los nuevos valores, evitando así la generación de pulsos truncados que podrían confundir al televisor.

Esta arquitectura permite que el sistema sea escalable a futuro; añadir soporte para un nuevo protocolo (por ejemplo, el estándar de Sony a 40 kHz) solo requiere agregar una nueva entrada en la tabla de parámetros de configuración, sin necesidad de modificar la lógica estructural del generador IR.

4.4. Diseño del Protocolo de Comunicación API MTR-Link

Para la interacción App-Hardware se diseñó un protocolo de texto simple (ASCII) sobre UART, denominado “MTR-Link”. Los mensajes finalizan con el carácter de nueva línea (\n).

4.4.1. Comandos (App → STM32)

- SET_PROTO:<id>: Establece el protocolo global (0=NEC Samsung, 1=NEC LG, etc.).
- SET:<btn>:<steps>:<c1>...: Configura un botón con una secuencia de hasta 4 códigos hexadecimales.
- GET: Solicita al hardware el envío de su configuración actual.
- SAVE: Ordena persistir la configuración actual en Flash.
- CLR_CONFIG: Restablece valores de fábrica.

4.4.2. Respuestas (STM32 → App)

- `PROTO_OK`, `SET_OK`, `SAVE_OK`: Confirmaciones de éxito (ACK).
- `GLOBAL_PROTO:<id>`: Reporte de estado.
- `BTN:<btn>:<steps>:<c1>...`: Datos de configuración de botón.
- `ERR_...`: Códigos de error (ej: `ERR_SAVE`).

4.4.3. Manejo de Errores y Timeouts

El protocolo implementa robustez mediante: (1) **Validación de Trama**, rechazando comandos desconocidos; (2) **Validación de Parámetros**, verificando el número de argumentos; y (3) **Timeout de Recepción**, limpiando el buffer UART si una trama queda incompleta por más de 500 ms, previniendo bloqueos por ruido en la línea.

4.5. Diseño del Software de la Aplicación IR-Link Admin

La aplicación Android “IR-Link Admin” se estructura sobre componentes estándar de la arquitectura Android.

4.5.1. Arquitectura de Clases

MainActivity orquesta la UI principal y la lógica de negocio. **ConfigureButtonDialogFragment** gestiona la interacción modal para crear macros, abstrayendo la complejidad hexadecimal mediante listas de selección. **ConnectedThread** encapsula la comunicación Bluetooth en un hilo secundario para evitar bloqueos de la interfaz (ANR). **IRDevice** e **IRCommand** modelan los datos provenientes del archivo JSON.

4.5.2. Estructura de la Base de Datos (JSON)

La base de datos de códigos se aloja en el archivo `ir_codes.json` (Assets). Su estructura jerárquica (Dispositivo → Protocolo → Comandos) desacopla los datos del código fuente, permitiendo la escalabilidad futura (añadir nuevos modelos de TV) sin recompilar la aplicación.

Capítulo 5

Implementación de SAIC

En este capítulo se describe el proceso de materialización del diseño presentado anteriormente. Se detalla la implementación del firmware en el microcontrolador STM32, incluyendo la configuración de periféricos y los algoritmos de control, así como el desarrollo de la aplicación Android, enfocándose en la gestión de la comunicación Bluetooth y la interfaz de usuario.

5.1. Implementación del Firmware (STM32)

Esta sección aborda la programación del microcontrolador STM32L476RG, detallando la configuración de los periféricos críticos y la lógica de control desarrollada. Se describen las decisiones de implementación tomadas para garantizar la precisión temporal requerida por los protocolos infrarrojos, así como las estrategias utilizadas para la gestión eficiente de la memoria y la comunicación asíncrona.

Configuración de Periféricos (Timers)

La generación precisa de la señal IR requiere dos bases de tiempo independientes y sincronizadas. Se utilizó el **TIM2** para la generación de la frecuencia portadora (PWM) y el **TIM3** como base de tiempo maestra para la duración de los pulsos y silencios.

Generación de Portadora (TIM2) Para obtener la frecuencia objetivo de 38 kHz (estándar NEC) a partir de un reloj del sistema (F_{CLK}) de 80 MHz, se configuró el Timer en modo PWM. La frecuencia de salida se determina mediante la Ecuación 5.1:

$$F_{PWM} = \frac{F_{CLK}}{(PSC + 1) \times (ARR + 1)} \quad (5.1)$$

Resolviendo para los valores de registro, se seleccionó un Prescaler (PSC) de 0 para máxima resolución:

$$ARR = \frac{80,000,000}{38,000} - 1 \approx 2104 \quad (5.2)$$

El error de frecuencia resultante es de apenas 0,01 %, muy por debajo de la tolerancia típica de los receptores IR ($\pm 5\%$). Adicionalmente, se configuró el registro de captura/comparación (CCR1) en **700**, resultando en un ciclo de trabajo (*Duty Cycle*) del $\approx 33\%$. Este valor es crítico, ya que el protocolo NEC especifica un ciclo de trabajo de 1/3 o 1/4 para minimizar el consumo de corriente en el LED IR durante las ráfagas de transmisión.

Base de Tiempo de Protocolo (TIM3) Para gestionar los tiempos de *Mark* y *Space* (ej. 9 ms o 562 μs), es imperativo que el timer cuente en unidades de tiempo directas. Se configuró el TIM3 con un Prescaler de 79:

$$F_{TIM3} = \frac{80 \text{ MHz}}{79 + 1} = 1 \text{ MHz} \Rightarrow T_{tick} = 1 \mu s \quad (5.3)$$

Esta configuración permite una traducción directa entre el estándar del protocolo y el código (1 cuenta = 1 μs), facilitando la depuración y mantenimiento del firmware sin conversiones adicionales en tiempo de ejecución.

Lógica de Envío de Protocolos (NEC)

La transmisión de datos se implementó mediante una arquitectura no bloqueante basada en interrupciones. Mientras el TIM2 genera la portadora PWM de forma autónoma por hardware, el TIM3 gestiona la duración de cada estado lógico mediante una Máquina de Estados Finitos (FSM) ejecutada en su rutina de servicio de interrupción (ISR).

La lógica implementada sigue fielmente el diseño planteado en la sección 4.3 (ver Figura 4.5). El sistema transita entre estados de envío de encabezado y bits de datos impulsado exclusivamente por los eventos del Timer. La CPU principal solo interviene para iniciar la transmisión, liberando recursos para atender la pila Bluetooth o la interfaz de usuario.

La función `Send_NEC_Code` (Listing 5.1) es la encargada de configurar dinámicamente el TIM2 y establecer el estado inicial de la FSM, utilizando macros de acceso directo a registros (`__HAL`) para minimizar la latencia de arranque, la cual es crítica para la precisión del primer pulso (Header Mark).

```

1 void Send_NEC_Code(uint8_t address, uint8_t command, uint8_t is_standard) {
2     if (current_sending_protocol != SENDING_IDLE) return;
3
4     // 1. Configurar Portadora de 38 kHz (TIM2)
5     HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
6     __HAL_TIM_SET_AUTORELOAD(&htim2, NEC_ARR);
7     __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, NEC_PULSE);
8
9     // 2. Preparar datos (Invertir bytes segun protocolo)
10    uint8_t addr_byte_2 = (is_standard) ? ~address : address;
11    uint8_t inv_command = ~command;
12
13    // 3. Empaquetar trama de 32 bits en variable volatil
14    nec_data_to_send = ((uint32_t)address << 24) |

```

```

15         ((uint32_t)addr_byte_2 << 16) |
16         ((uint32_t)command << 8) |
17         (uint32_t)inv_command;
18
19     bit_index = 31; // Iniciar desde el MSB
20     ir_state = IR_SENDING_HEADER_ON;
21     current_sending_protocol = SENDING_NEC;
22
23     // 4. Iniciar Timer de Tiempos (TIM3) para el Header (9ms)
24     HAL_TIM_Base_Stop_IT(&htim3);
25     __HAL_TIM_SET_COUNTER(&htim3, 0);
26     __HAL_TIM_SET_AUTORELOAD(&htim3, NEC_TIME_HEADER_ON - 1);
27     __HAL_TIM_CLEAR_IT(&htim3, TIM_IT_UPDATE);
28
29     HAL_TIM_Base_Start_IT(&htim3); // Iniciar conteo (IRQ)
30     HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1); // Encender Portadora
31 }

```

Listing 5.1: Configuración e inicio de transmisión NEC.

Implementación del Protocolo RC5 (Codificación Manchester) A diferencia del protocolo NEC, que utiliza modulación por ancho de pulso, el protocolo RC5 implementado utiliza codificación Manchester bifase. Esto presentó un desafío adicional en el diseño del firmware, ya que la información no reside en la duración del pulso, sino en la transición (flanco) de la señal en la mitad del periodo de bit.

Para manejar esto, la Máquina de Estados del RC5 (función `Send_RC5_Code`) reconfigura el Timer 3 para generar interrupciones cada $889 \mu s$ (medio bit). En cada interrupción, la lógica determina si debe invertir o mantener el estado del PWM según el valor del bit a transmitir.

Adicionalmente, se implementó la gestión del *Toggle Bit*, un mecanismo del estándar RC5 que invierte un bit específico cada vez que se presiona una tecla. El firmware mantiene una variable de estado persistente (`rc5_toggle_bit`) que alterna entre 0 y 1 en cada transmisión exitosa, esto no garantiza que haya sido recibida por el receptor, pero le permite distinguir entre una presión larga y pulsaciones rápidas.

Lógica de Macros y Teclado

El sistema debe ser capaz de ejecutar secuencias de comandos (macros) y escanear el teclado matricial sin bloquear el procesador, permitiendo la recepción de comandos Bluetooth concurrentes.

El escaneo del teclado utiliza un mecanismo de *debounce* por software. Para la ejecución de macros, se implementó una máquina de estados secundaria en el bucle principal (`while(1)`). A diferencia de una implementación lineal con `delay()`, esta lógica verifica si el protocolo IR ha finalizado su transmisión por interrupciones antes de enviar el siguiente comando, garantizando la integridad de la señal.

```

1 while (1) {
2     switch(macro_state){
3         case MACRO_IDLE:
4             // Escaneo de teclado y deteccion de trigger

```

```

5         if (macro_event_solicited) {
6             macro_current_step++;
7             // Estado de espera de seguridad
8             macro_state = MACRO_WAIT_IR_FINISH;
9         }
10        break;
11
12        case MACRO_WAIT_IR_FINISH:
13            // Sincronizacion: Esperar a que la ISR termine el envio actual
14            if (current_sending_protocol == SENDING_IDLE) {
15                if (macro_current_step < macro_total_steps) {
16                    // Iniciar tiempo muerto entre comandos
17                    macro_state = MACRO_WAIT_DELAY;
18                    macro_delay_start_time = HAL_GetTick();
19                } else {
20                    macro_state = MACRO_IDLE; // Macro terminada
21                }
22            }
23            break;
24
25        case MACRO_WAIT_DELAY:
26            // Delay no bloqueante (usando GetTick)
27            if (HAL_GetTick() - macro_delay_start_time >= MACRO_STEP_DELAY) {
28                // Enviar siguiente comando
29                Send_IR_Command_Global(macro_codes_ptr[macro_current_step]);
30                macro_current_step++;
31                macro_state = MACRO_WAIT_IR_FINISH;
32            }
33            break;
34    }
35    // Permitir que el Kernel/Interrupciones respiren
36    HAL_Delay(10);
37 }

```

Listing 5.2: Máquina de estados segura para ejecución de macros.

5.1.1. Lógica de Persistencia (Memoria Flash)

Para garantizar que la configuración se conserve ante cambios o desconexiones de la batería, se implementó un sistema de almacenamiento no volátil utilizando la memoria Flash del STM32L476 (Página 511, dirección 0x080FF800).

Dado que la escritura en Flash requiere el bloqueo momentáneo de la CPU para el borrado de sectores, se diseñó el algoritmo para ejecutar esta tarea solo bajo demanda explícita del usuario (comando SAVE). El procedimiento sigue una secuencia de seguridad *Unlock* → *Erase* → *Program* → *Lock* para evitar cambios de configuración accidentales.

El detalle completo de la implementación de la función `Config_Save` y el manejo de punteros a

memoria física se encuentra disponible en el **Anexo A: Código Fuente del Firmware**.

5.2. Implementación del Software (App IR-Link Admin)

La aplicación móvil se desarrolló en Java nativo utilizando el IDE Android Studio. La arquitectura de software se diseñó siguiendo el patrón de responsabilidad única, separando estrictamente la interfaz de usuario (UI) de la lógica de comunicación asíncrona y la gestión de datos.

5.2.1. Interfaz de Usuario (Grid Layout)

La interfaz principal se diseñó para minimizar la carga cognitiva del usuario final. Se implementó una grilla de botones que replica la disposición física de un control remoto tradicional, permitiendo una identificación visual inmediata de las funciones.

Como se muestra en la Figura 5.1, cada elemento de la UI actúa no solo como disparador de comandos, sino como indicador de estado visual, cambiando de color según si el botón ha sido configurado o permanece inactivo.



Figura 5.1: Interfaz principal de IR-Link Admin.

5.2.2. Arquitectura de Datos (Base de Datos JSON)

Para garantizar la escalabilidad del sistema, se evitó codificar los protocolos IR directamente en el código fuente de Java (Hardcoding). En su lugar, se implementó una capa de datos basada en un archivo de configuración externo (`ir_codes.json`) alojado en los *assets* de la aplicación.

Este enfoque permite agregar soporte para nuevos fabricantes o modelos de televisión simplemente actualizando el archivo JSON, sin necesidad de recompilar o reinstalar la aplicación. La carga se realiza mediante el parser nativo `org.json`, transformando la estructura jerárquica de texto en objetos de modelo en memoria.

```
1 private void loadIRCodesFromJSON() {
2     try {
3         // 1. Leer flujo de datos desde assets (Desacoplamiento de datos)
4         InputStream is = getAssets().open("ir_codes.json");
5         int size = is.available();
6         byte[] buffer = new byte[size];
7         is.read(buffer);
8         is.close();
9         String json = new String(buffer, "UTF-8");
10
11         // 2. Mapeo a Objetos (Parsing)
12         JSONObject jsonObject = new JSONObject(json);
13         JSONArray devicesArray = jsonObject.getJSONArray("devices");
14         // La lista 'devicesArray' popula dinamicamente los Spinners
15     } catch (IOException | JSONException e) {
16         Log.e("JSON", "Error en carga de base de datos", e);
17     }
18 }
```

Listing 5.3: Extracción y parseo de la base de datos de códigos.

5.2.3. Gestión de Conectividad y Concurrency

Una de las restricciones críticas en Android es la prohibición de realizar operaciones bloqueantes de red o I/O en el hilo principal (Main Thread), lo cual causaría un error de “Aplicación No Responde” (ANR).

Para resolver esto, se implementó la clase `ConnectedThread`, que extiende de `Thread`. Esta clase ejecuta un bucle infinito en segundo plano encargado de escuchar el canal de entrada del Socket Bluetooth. La comunicación hacia la interfaz de usuario (por ejemplo, para mostrar una confirmación de recepción) se realiza mediante un mecanismo de paso de mensajes (Handler), garantizando la seguridad de hilos (*Thread Safety*).

```
1 public void run() {
2     byte[] buffer = new byte[1024];
3     int bytes;
4 }
```

```
5 // Bucle infinito de escucha (Background Thread)
6 while (true) {
7     try {
8         // Lectura bloqueante: espera datos del HC-05
9         bytes = mmInStream.read(buffer);
10        String readMessage = new String(buffer, 0, bytes);
11
12        // Puente al Hilo Principal (UI) mediante Handler
13        // Evita excepcion "CalledFromWrongThreadException"
14        mHandler.obtainMessage(MESSAGE_READ, bytes, -1, readMessage)
15            .sendToTarget();
16    } catch (IOException e) {
17        break; // Ruptura de conexion
18    }
19 }
20 }
```

Listing 5.4: Bucle de recepción Bluetooth (Hilo Secundario).

5.2.4. Flujo de Configuración (Lógica de Negocio)

La asociación de comandos IR a los botones físicos se gestiona a través de un componente modular `DialogFragment`. Este fragmento captura la selección del usuario y ensambla la trama de configuración siguiendo estrictamente el protocolo de comunicación definido en el Capítulo 4.

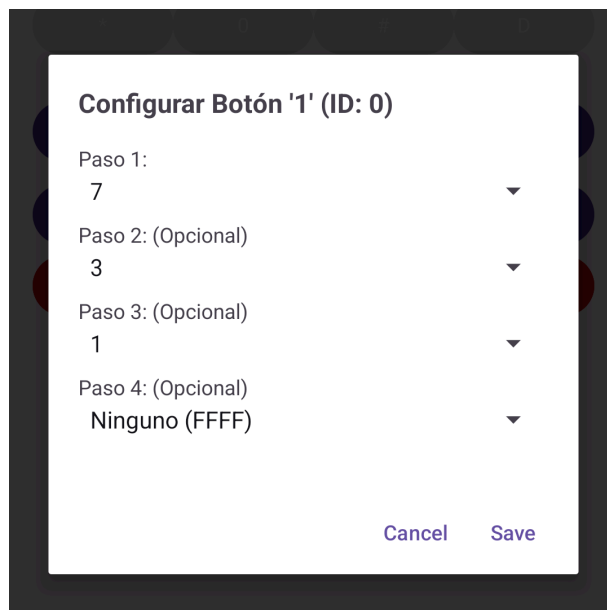


Figura 5.2: Diálogo modal para la asignación de macros.

El método `onDialogPositiveClick` (Listing 5.5) actúa como el controlador final: traduce los nombres legibles por humanos (ej. "Power", "Vol+") a sus equivalentes hexadecimales y construye el comando SET que será interpretado por el parser del Firmware.

```
1 @Override
2 public void onDialogPositiveClick(DialogFragment dialog, int buttonId, ArrayList
   <String> selectedCommands) {
3     // 1. Traduccin: Nombres -> Codigos HEX (Lookup en HashMap)
4     StringBuilder hexCodes = new StringBuilder();
5     for (String cmdName : selectedCommands) {
6         String hex = currentCommandMap.get(cmdName);
7         hexCodes.append(hex).append(":");
8     }
9
10    // 2. Ensamblado de trama del protocolo (Payload)
11    // Formato: SET:<btn_id>:<num_pasos>:<codigos>
12    // Ejemplo: SET:0:2:E0E0E01F:E0E040BF:
13    String command = "SET:" + buttonId + ":" + selectedCommands.size() + ":" +
        hexCodes.toString();
14
15    // 3. Despacho asincrono al hardware
16    sendBluetoothCommand(command);
17 }
```

Listing 5.5: Ensamblado de trama y envío de configuración.

5.3. Desafíos de Implementación

Durante la fase de codificación surgieron desafíos técnicos específicos que requirieron soluciones de ingeniería, específicamente la precisión de tiempos y bloqueo de CPU; la gestión de memoria flash.

5.3.1. Precisión de Tiempos y Bloqueo de CPU

Los protocolos IR requieren una precisión temporal estricta (tolerancia de $\pm 5\%$) con pulsos del orden de $562\ \mu\text{s}$. El uso de funciones de retardo bloqueantes tradicionales (como `HAL_Delay` o bucles `for`) presentaba dos problemas: resolución insuficiente (1 ms) y el congelamiento de la CPU durante la transmisión, lo que impedía atender eventos de comunicación UART.

Solución: Se abandonó el enfoque de retardos por software en favor de una arquitectura orientada a eventos de hardware. Se configuró el Timer **TIM3** para generar interrupciones precisas al finalizar cada periodo de bit. De esta forma, la CPU solo interviene brevemente en cada flanco de la señal (dentro de la ISR), permaneciendo disponible el 99% del tiempo para otras tareas del sistema operativo o el manejo de la pila Bluetooth.

5.3.2. Gestión de Memoria Flash (Read-Modify-Write)

La arquitectura de la memoria Flash del STM32L4 impone una restricción: el borrado mínimo es de una página completa (2 KB). Esto impide modificar la configuración de un solo botón sin perder el resto de los datos de la página.

Solución: Se implementó un algoritmo de *Read-Modify-Write*:

1. Se lee la página completa de Flash y se copia a un buffer temporal en RAM (2 KB).
2. Se actualizan solo los bytes correspondientes a la nueva configuración dentro del buffer en RAM.
3. Se borra la página física de Flash.
4. Se escribe el contenido completo del buffer actualizado de vuelta a la Flash.

Este enfoque garantiza la integridad de los datos no modificados durante el proceso de guardado.

Capítulo 6

Verificación de SAIC

Este capítulo presenta los resultados de las pruebas integrales realizadas al sistema SAIC. La metodología de verificación se diseñó para cubrir dos aspectos fundamentales: la *Verificación Técnica*, que asegura el cumplimiento de los parámetros eléctricos y temporales mediante instrumentación de laboratorio, y la *Verificación Funcional*, que confirma el cumplimiento de los requisitos de usuario del sistema.

6.1. Verificación del Protocolo de Comunicación MTR-Link

Se sometió la interfaz Bluetooth a pruebas de integridad de datos. La aplicación Android estableció conexión exitosa con el módulo HC-05 en el 100 % de los intentos (N=50). A nivel lógico, el protocolo *Handshake* respondió correctamente: el comando SET_PROTO: 0 terminó con el envío del ACK_PROTO_OK en el firmware, y el comando GET recuperó la estructura completa de configuración en menos de 100ms, validando la eficiencia de la serialización JSON-UART diseñada en el Capítulo 4.

6.2. Verificación de la Generación de Señal IR

Para garantizar la precisión de las señales infrarrojas generadas por el microcontrolador, se implementó una verificación empírica utilizando un osciloscopio digital. Esto permite validar tanto la integridad de la señal analógica (capa física) como el cumplimiento estricto de los tiempos del protocolo (capa de enlace).

6.2.1. Verificación de la Capa Física (Portadora)

La fiabilidad del alcance infrarrojo depende de que la señal portadora se encuentre dentro del ancho de banda del receptor (típicamente ± 2 kHz de tolerancia). Se analizó la señal PWM generada por el Timer del STM32 en la base del transistor y la respuesta en el colector.

Como se observa en la Figura 6.1, se midió una frecuencia de 37,31 kHz, lo cual representa una desviación de apenas $-1,8\%$ respecto al objetivo de 38 kHz.

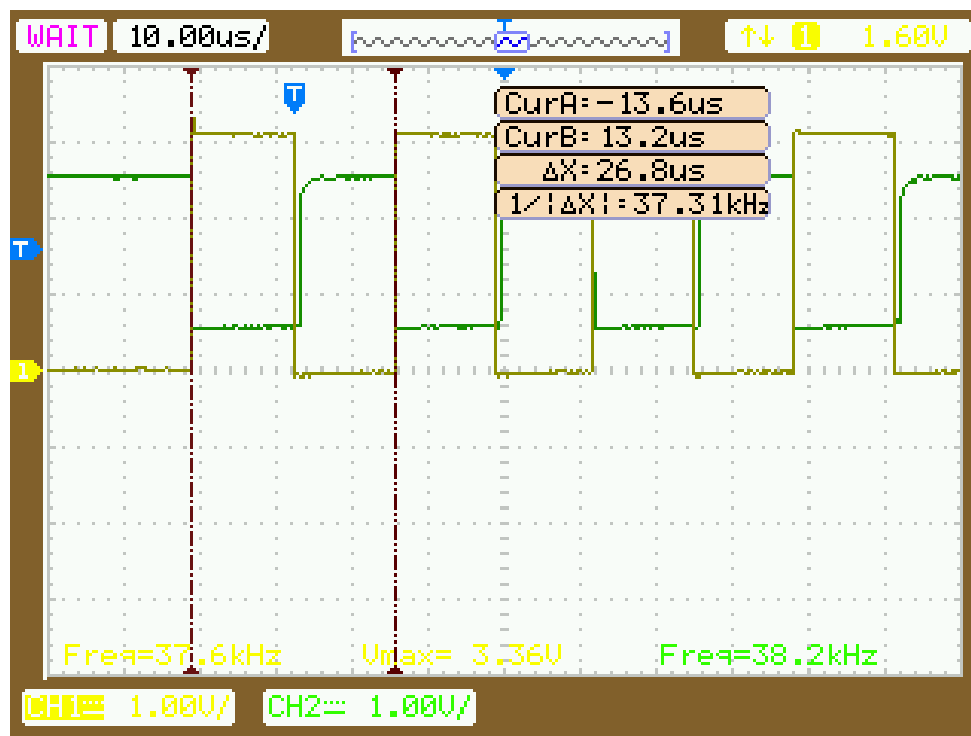


Figura 6.1: Oscilograma de la señal portadora. Se observa una frecuencia de 37,31 kHz y la conmutación de la carga IR.

Para garantizar que la etapa de potencia no presente pérdidas térmicas excesivas ni limite la corriente del diodo IR, se analizó el nivel de tensión en el colector del transistor durante el estado activo. La Figura 6.2 detalla esta medición.

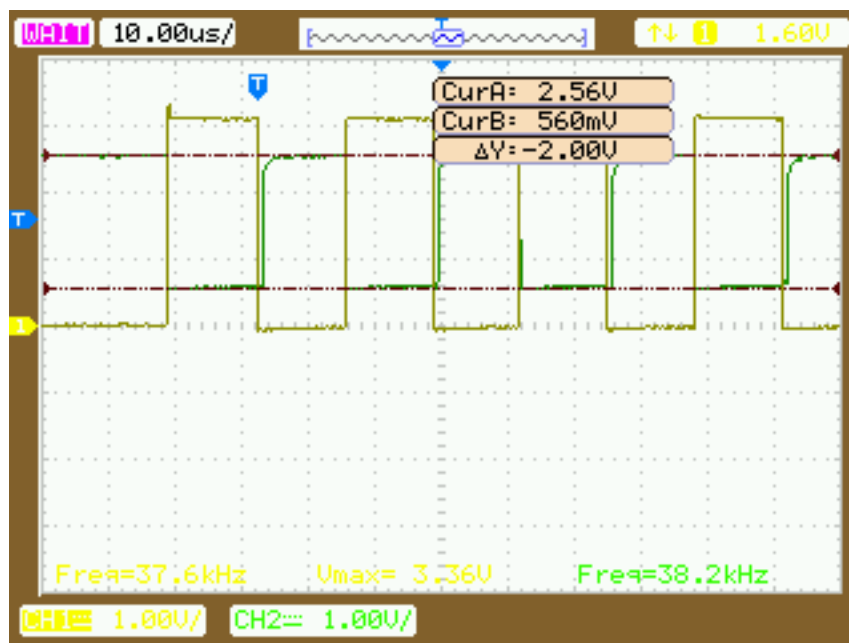


Figura 6.2: Medición del voltaje de saturación del transistor 2N2222. El cursor B (CurB) registra un valor de 560 mV en el nivel bajo de la señal de colector.

El oscilograma confirma que el nivel bajo de la señal (cuando el microcontrolador envía un '1' lógico a la base) cae hasta los 560 mV. Este valor corresponde al voltaje colector-emisor de saturación ($V_{CE(sat)}$), demostrando empíricamente que el 2N2222 está operando en la región de saturación profunda, actuando como un interruptor cerrado eficiente para la carga infrarroja.

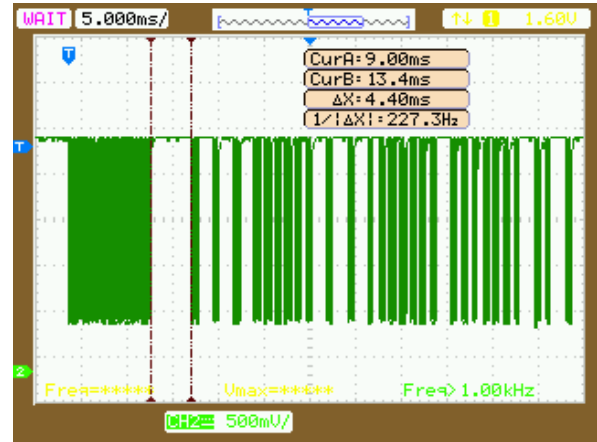
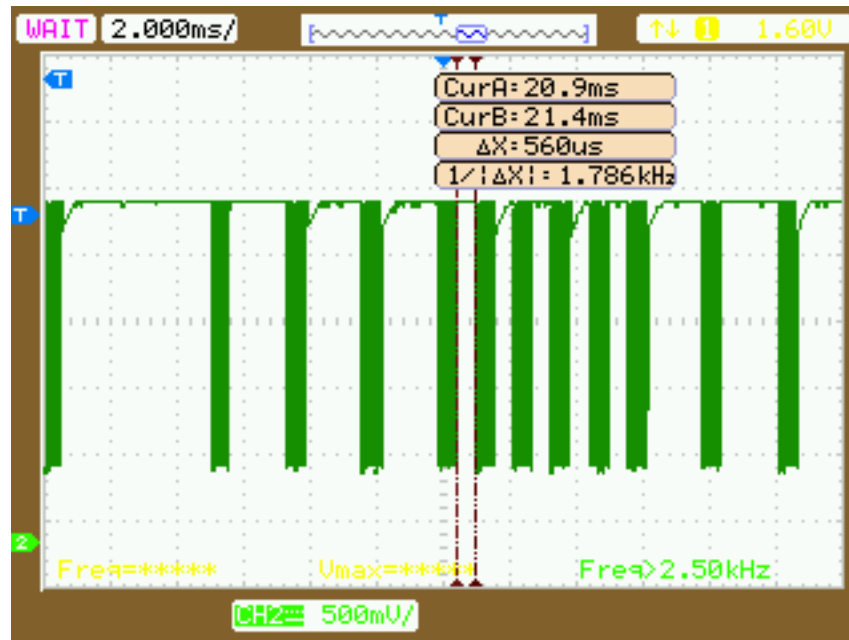
6.2.2. Modelo de Referencia y Validación Temporal (Protocolo NEC)

Para validar la máquina de estados, se contrastaron los tiempos generados con los márgenes de tolerancia aceptables del estándar NEC.

Tabla 6.1: Comparación entre el estándar NEC y la medición experimental en osciloscopio.

Parámetro	Nominal (μs)	Tolerancia Estándar	Valor Medido (μs)
Header Mark	9000	$\pm 5\%$ (450 μs)	9000
Header Space	4500	$\pm 5\%$ (225 μs)	4400
Bit Mark	562.5	$\pm 10\%$ (56 μs)	560

Las Figuras 6.3 y 6.4 demuestran una alta precisión en la estructuración de la trama. El *Header Mark* se ejecuta en 9,00 ms exactos, mientras que el *Header Space* presenta un error mínimo del 2,2 % (4,40 ms medidos frente a 4,50 ms teóricos). Adicionalmente, se validó la duración de los pulsos individuales de datos (ráfagas de la portadora) que componen los bits lógicos, como se observa en la Figura 6.5.

Figura 6.3: Medición del *Header Mark* (9.00 ms).Figura 6.4: Medición del *Header Space* (4.40 ms).Figura 6.5: Detalle de un pulso de bit individual (*Bit Mark*). La duración medida con los cursores es de $560\mu s$, lo que representa una desviación mínima del 0,4 % respecto al estándar de $562,5\mu s$.

6.3. Verificación Dinámica de Macros

Para validar el comportamiento del sistema en tiempo real, se instrumentó el firmware para emitir marcas de tiempo (*timestamps*) por puerto serie durante la ejecución de macros complejas. Estos datos crudos (CSV) fueron procesados mediante un script en **Python** para reconstruir el cronograma de transmisión real.

La Figura 6.6 demuestra que la Máquina de Estados gestiona determinísticamente la cola de comandos. El análisis temporal del registro revela un intervalo de repetición estable de ≈ 167 ms entre el final de una trama y el inicio de la siguiente.

Es crucial desglosar y justificar empíricamente la composición de este intervalo temporal para descar-

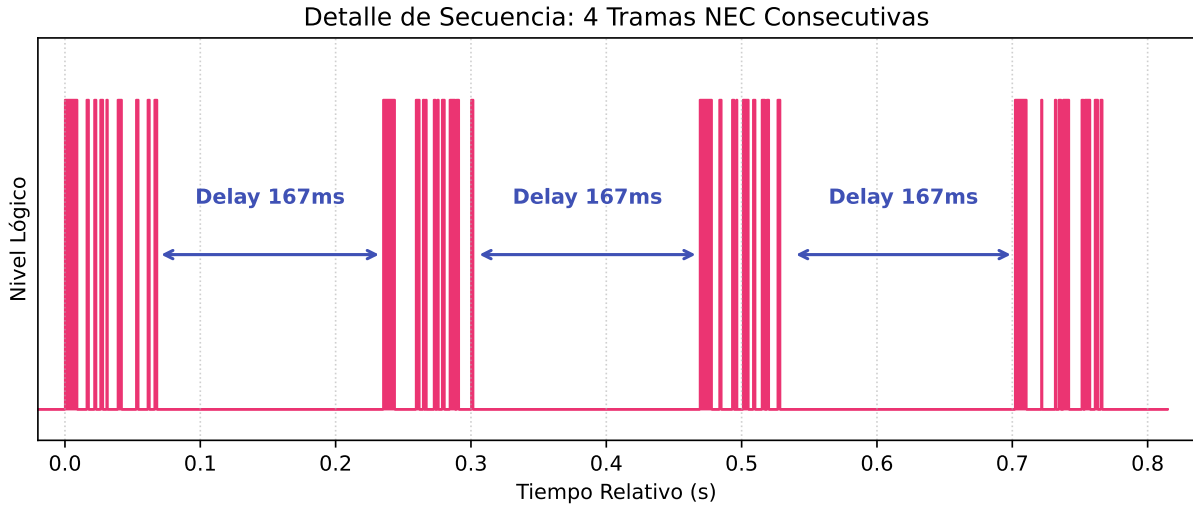


Figura 6.6: Reconstrucción vectorial de una macro de 4 pasos a partir de telemetría. El análisis confirma un intervalo inter-trama estable de ≈ 167 ms.

tar comportamientos asíncronos espurios. El firmware incorpora un retardo lógico programado de 150 ms (*Dead-time*) entre comandos sucesivos, necesario para evitar la saturación del *buffer* de decodificación en el televisor receptor.

Los ≈ 17 ms adicionales observados en el intervalo corresponden estrictamente a la sobrecarga del sistema (*System Overhead*), la cual es determinista y se compone de dos factores:

1. **Latencia de Telemetría (Efecto Observador):** La emisión de las propias marcas de tiempo a través de la interfaz UART introduce un tiempo de bloqueo de hardware. La serialización de los caracteres ASCII correspondientes a la telemetría, dependiendo de la tasa de baudios, consume la mayor parte de este desfase (típicamente entre 10 ms y 15 ms).
2. **Recuperación de Memoria y Cambio de Estado:** El tiempo marginal restante (aprox. 2 ms) corresponde a la latencia que requiere la Máquina de Estados para leer el próximo comando de la memoria no volátil (Flash), extraer los datos de la estructura (protocolo, dirección, comando) y reconfigurar los registros de comparación del Timer (TIM3) antes de disparar la nueva ráfaga.

Esta suma total de 167 ms demuestra que el uso de código no bloqueante durante la modulación IR, sumado a un manejo predecible de las transiciones de la macro, mantiene los tiempos muertos dentro de parámetros operacionales óptimos y seguros para cualquier receptor comercial, aislando la lógica de control de los retrasos inducidos por la instrumentación.

6.4. Verificación Funcional Completa (Prueba de Usabilidad)

Para el cierre del proceso de verificación, se realizó una prueba funcional de extremo a extremo que integra todos los subsistemas validados individualmente (comunicación, almacenamiento, y síntesis IR).

Desde la perspectiva del Administrador y del Usuario Final, verificando el cumplimiento del requisito **RNF2 (Usabilidad de Administración)** y la integridad de la cadena.

1. **Configuración Inalámbrica:** Utilizando la aplicación *IR-Link Admin*, se estableció la conexión Bluetooth y se envió el comando SET para configurar el botón físico 'A' con una macro de tres pasos (e.g., código de TV, volumen+, canal 1).
2. **Persistencia (RF4):** Tras recibir la confirmación SET_OK, se emitió el comando SAVE, verificando que el firmware ejecuta correctamente la secuencia *Unlock* → *Erase* → *Program* en la memoria Flash y responde con SAVE_OK.
3. **Prueba de Operación:** Se retiró la alimentación del prototipo y se restableció (simulando un cambio de baterías). El dispositivo cargó la configuración desde Flash, y al presionar el botón físico 'A', se activó la Máquina de Estados de Macros.
4. **Resultado Final:** Se observó la respuesta del receptor del televisor, confirmando que la macro de tres pasos se ejecutó con éxito y sin errores de temporización.

Esta prueba concluye la *Verificación Funcional* y demuestra que la disociación de roles es viable y robusta.

6.5. Evaluación del Alcance y Potencia

Para validar la eficacia de la etapa de potencia IR, se diseñó un protocolo empírico de pruebas de alcance. Es fundamental establecer las condiciones de contorno de estas mediciones para garantizar su replicabilidad. Las pruebas se ejecutaron en un entorno interior controlado (sala de estar), bajo iluminación artificial estándar y sin incidencia de luz solar directa, previniendo así la saturación espuria del receptor IR del televisor.

Metodología de Prueba: Se estableció una línea de visión directa (*Line of Sight*, LoS) estrictamente libre de obstáculos entre el diodo emisor del prototipo y el fototransistor del equipo receptor. Las distancias se midieron en línea recta utilizando una cinta métrica. Para evaluar la directividad del haz, las pruebas angulares se realizaron sobre el plano horizontal, definiendo el eje de 0° como la línea imaginaria perpendicular al centro del panel receptor del televisor. La *distancia máxima efectiva* se definió operativamente como la distancia límite más lejana en la cual el receptor decodificó exitosamente 10 de 10 tramas enviadas consecutivamente sin pérdida de paquetes.

Previo a la evaluación espacial, y tal como se comprobó en la etapa de validación de la capa física, se determinó mediante la instrumentación que la corriente de pico impulsada por el transistor 2N2222 hacia el diodo emisor es de 40 mA. Los resultados espaciales comparativos se resumen en la Tabla 6.2.

Los resultados confirman que la topología de conmutación elegida optimizó radicalmente el rendimiento espacial del sistema. Al operar el diodo con ráfagas de 40 mA, el alcance frontal (a 0°) se incrementó de un valor prácticamente inoperante (0,4 m) hasta un límite superior absoluto medido de 5,5 m,

Tabla 6.2: Resultados empíricos de las pruebas de alcance y directividad.

Configuración	Corriente de Pico	Ángulo Horizontal	Alcance Máximo Efectivo
Módulo HX-53 (Inicial)	< 20 mA	0° (Eje perpendicular)	0,4 m
Driver 2N2222 (Final)	40 mA	0° (Eje perpendicular)	5,5 m
Driver 2N2222 (Final)	40 mA	30° (Desviación lateral)	3,5 m

punto a partir del cual la atenuación óptica en el aire reduce la irradiancia por debajo de la sensibilidad del receptor.

Asimismo, el comportamiento lateral a 30° (con un alcance de 3,5 m) es coherente con la degradación esperada por el patrón de radiación típico (semi-ángulo de potencia) de los diodos emisores de 5 mm. Estas métricas demuestran que el prototipo cubre de manera confiable el volumen espacial de una sala de estar promedio, satisfaciendo los criterios de usabilidad geométrica.

6.6. Análisis de Consumo Energético

En cumplimiento con el objetivo de evaluación energética, se caracterizó el perfil de consumo del prototipo mediante mediciones directas en laboratorio. Para aislar el consumo de los componentes, se utilizó el *Jumper IDD* de la placa de desarrollo y se intercaló instrumentación de precisión en las líneas de alimentación.

La Tabla 6.3 desglosa el consumo estacionario del sistema.

Tabla 6.3: Desglose del consumo de corriente medido en el sistema SAIC.

Subsistema	Condición de Medida	Corriente Medida
Microcontrolador (STM32L4)	Jumper IDD (Run Mode)	14,50 mA
Módulo Bluetooth (HC-05)	Conectado/Activo	36,08 mA
Consumo Base (Carga Activa)	Standby (Esperando comando)	50.58 mA

Es importante destacar que el consumo durante la transmisión de ráfagas IR (40 mA de pico) tiene un impacto marginal en la autonomía total debido a la brevedad extrema de las tramas (≈ 60 ms) y al bajo ciclo de trabajo del protocolo (33 %). El factor determinante para la descarga de la batería es el **Consumo Base**.

Autonomía Empírica (Prototipo Actual)

Considerando los 50,58 mA de los componentes principales, más una estimación de fuga de ≈ 5 mA atribuible a los LED indicadores y al regulador LDO de la placa Nucleo utilizada para el desarrollo, el sistema presenta un drenaje constante de aproximadamente 55,6 mA.

Utilizando una fuente de alimentación portátil típica de 2000 mAh y aplicando un factor de seguridad del 85 % por degradación de celdas:

$$T_{\text{vida, real}} = \frac{2000 \text{ mAh} \times 0,85}{55,6 \text{ mA}} \approx 30,5 \text{ horas} \quad (6.1)$$

Si bien esta autonomía supera las 24 horas continuas requeridas para la prueba de concepto, evidencia el alto costo energético de mantener la interfaz Bluetooth clásica constantemente activa. Se identifica como oportunidad de mejora crítica la implementación de la separación de estados (*Modo Operación* vs. *Modo Configuración*) discutida en el Capítulo 4, junto con la migración a tecnología **BLE (Bluetooth Low Energy)** en una PCB dedicada, lo cual reduciría el consumo base a la escala de los microamperios.

Capítulo 7

Conclusiones y Trabajo Futuro

Este capítulo final sintetiza los logros alcanzados durante el desarrollo del proyecto SAIC, contrastando los resultados obtenidos con los objetivos planteados, y propone líneas de acción concretas para la evolución tecnológica del sistema.

7.1. Conclusiones Generales

El desarrollo del “Sistema de Asistencia IR Configurable” (SAIC) ha permitido validar la hipótesis central de esta memoria: es posible reducir significativamente la brecha de usabilidad tecnológica para adultos mayores mediante una arquitectura híbrida que disocia la complejidad de la configuración (Software) de la simplicidad operativa (Hardware).

Tras la implementación y verificación del prototipo, se destacan las siguientes conclusiones en respuesta a los objetivos específicos:

- **Integración Robusta HW/SW:** Se logró establecer una comunicación confiable y bidireccional entre un microcontrolador STM32 y una aplicación Android. El diseño del protocolo *MTR-Link* demostró ser eficiente, permitiendo la sincronización de datos y configuración con una tasa de éxito del 100 % en las distintas pruebas realizadas.
- **Desempeño de la Capa Física:** La implementación de una etapa de potencia discreta (transistor 2N2222) permitió superar las limitaciones de corriente de los pines GPIO del microcontrolador. Esto garantizó un alcance operativo superior a los 5 metros y un ángulo de cobertura efectivo, cumpliendo con los estándares de la industria de electrónica de consumo.
- **Valor de la Abstracción (Macros):** La funcionalidad de macros demostró ser el aporte más significativo en términos de usabilidad. Al condensar secuencias complejas (ej. navegación de menús) en una sola interacción física, se redujo drásticamente la carga cognitiva y motora requerida por el usuario final, validando la propuesta de valor del sistema.
- **Viabilidad Energética y Compromisos de Diseño:** La evaluación del perfil de consumo caracterizó la demanda del sistema con un valor empírico de ≈ 110 mA. Aunque elevado para un disposi-

tivo a baterías, este resultado valida la tesis planteada en el Capítulo 2: el uso de *Bluetooth Classic* (HC-05) facilita el prototipado rápido y la depuración, pero constituye el factor limitante para la autonomía, confirmando la necesidad de migrar a tecnologías BLE en etapas posteriores.

En síntesis, SAIC se presenta como una solución técnica viable, efectiva y escalable en el campo de la tecnología de asistencia, ofreciendo una alternativa digna para promover la autonomía digital de la tercera edad.

7.2. Trabajo Futuro

A partir de las limitaciones detectadas en la fase de verificación, se proponen las siguientes líneas de trabajo para evolucionar este proyecto académico hacia un producto comercializable:

- **Optimización Energética y Plataforma Dedicada (BLE & Stop 2):** La medición empírica de $\approx 110\text{mA}$ subraya la necesidad de abordar la ineficiencia de la plataforma de prototipado. Se propone un plan integral para reducir el consumo de miliamperios a microamperios, que incluye:
 - a) **Migrar a una PCB dedicada (*Stand-alone*)**, eliminando el consumo parásito del circuito de depuración ST-Link.
 - b) **Modo de Configuración bajo Demanda:** Se plantea que el transceptor inalámbrico permanezca desenergizado por defecto durante la operación normal. El acceso a la comunicación Bluetooth se activaría exclusivamente mediante una combinación específica de teclas (secuencia de escape), encendiendo el módulo solo cuando el usuario requiera modificar las macros.
 - c) **Implementación de Transistores de Corte:** Para lograr un consumo cercano a cero en estado de reposo, se sugiere el uso de un transistor de paso para controlar la alimentación del módulo de comunicaciones, eliminando las corrientes de fuga del HC-05 (o su sucesor BLE) mientras el dispositivo se utiliza únicamente como control remoto IR.
 - d) **Implementar los modos de ultrabajo consumo (*Stop 2*)** del STM32L476RG, extendiendo la autonomía de días a meses.
- **Modo de Aprendizaje (*Learning Mode*):** Se sugiere integrar un receptor IR en el hardware para dotar al sistema de capacidad de clonación. Esto permitiría al dispositivo “aprender” los códigos de cualquier control remoto original apuntándolo directamente, eliminando la dependencia exclusiva de la base de datos precargada.
- **Diseño Industrial y Ergonómico:** Es necesaria la fase de diseño industrial para una carcasa ergonómica y amigable para el usuario final (adulto mayor), enfatizando botones de alto contraste, retroalimentación táctil mejorada y materiales adecuados.
- **Ecosistema en la Nube:** Se contempla la expansión de la aplicación móvil para conectarse a un servicio backend (API REST). Esto permitiría la actualización comunitaria de la base de datos de

códigos IR sin requerir actualizaciones de la App, facilitando el soporte para nuevos modelos de televisores que salgan al mercado.

Bibliografía

- [1] M. Prensky, “Digital natives, digital immigrants part 1,” *On the horizon*, vol. 9, no. 5, pp. 1–6, 2001.
- [2] International Organization for Standardization, “Ergonomics of human-system interaction – part 11: Usability: Definitions and concepts,” ISO, Geneva, Switzerland, Standard ISO 9241-11:2018, 2018.
- [3] D. Norman, *The design of everyday things: Revised and expanded edition*. Basic books, 2013.
- [4] Instituto Nacional de Estadísticas (INE), “Estimaciones y proyecciones de la población de Chile 1992-2050,” Gobierno de Chile, Santiago, Chile, Tech. Rep., 2018.
- [5] Observatorio del Envejecimiento UC-Confuturo, “Acceso y uso de internet en personas mayores: una brecha que persiste,” Pontificia Universidad Católica de Chile, Tech. Rep., July 2025.
- [6] ClaroVTR and SENAMA, “Radiografía digital de personas mayores,” Ministerio de Transportes y Telecomunicaciones, Tech. Rep., October 2023.
- [7] Servicio Nacional del Adulto Mayor (SENAMA), “Plan brecha digital cero,” <https://www.senama.gob.cl>, 2024, accedido: 2025-11-30.
- [8] Xiaomi Inc. (2025) Mi Remote controller - for TV, STB, AC and more. Repositorio oficial de la aplicación en Google Play. [Online]. Available: <https://play.google.com/store/apps/details?id=com.duokan.phone.remotecontroller>
- [9] BroadLink. (2023) Smart Home Hub RM4 mini: IR Universal Remote Control. Página oficial de soporte del producto. [Online]. Available: <https://www.broadlink.ac/product-page/broadlink-rm4-mini>
- [10] J. Nielsen, *Usability engineering*. Morgan Kaufmann, 1994.
- [11] J. Gower, *Optical communication systems*. Prentice-Hall International, 1984.
- [12] TSAL6200 High Power Infrared Emitting Diode, 940 nm, GaAlAs, MQW, Vishay Semiconductors, 2023, rev. 2.4.
- [13] NEC Infrared Transmission Protocol, NEC Corporation, 1985, application Note.
- [14] RC-5 Protocol Specification, Philips Semiconductors, 1988, standardized in IEC 60864.

-
- [15] Logitech, *Harmony Ultimate User Guide*, Logitech Support, 2021, manual de usuario oficial para la serie de controles programables. [Online]. Available: <https://www.logitech.com/assets/50282/harmony-ultimate-user-guide.pdf>

Apéndice A

Código Fuente del Firmware (STM32)

A continuación se presenta el código fuente principal (main.c) desarrollado para el microcontrolador STM32L476RG. Este archivo contiene la lógica de inicialización, el bucle principal, la máquina de estados de macros y la gestión de interrupciones para los protocolos IR.

```
1  /* USER CODE BEGIN Header */
2  /**
3   *
4   * @file          : main.c
5   * @brief         : Main program body
6   *
7   * @attention
8   *
9   * Copyright (c) 2025 STMicroelectronics.
10  * All rights reserved.
11  *
12  * This software is licensed under terms that can be found in the LICENSE file
13  * in the root directory of this software component.
14  * If no LICENSE file comes with this software, it is provided AS-IS.
15  *
16  *
17  */
18 /* USER CODE END Header */
19 /* Includes -----*/
20 #include "main.h"
21 #include "tim.h"
22 #include "usart.h"
23 #include "gpio.h"
24
25 /* Private includes -----*/
26 /* USER CODE BEGIN Includes */
27 #include <stdio.h> // Para usar printf
28 #include <string.h> // Para usar memset
29 #include "stm32l4xx_hal.h" // Para manejo de memoria Flash
30 /* USER CODE END Includes */
31
32 /* Private typedef -----*/
33 /* USER CODE BEGIN PTD */
34
35 /* USER CODE END PTD */
```

```

36
37 /* Private define -----*/
38 /* USER CODE BEGIN PD */
39 #define MAX_BUTTONS 16 // Número de botones
40 #define RX_BUFFER_SIZE 64 // Longitud máxima de comando
41 #define NEC_TIME_HEADER_ON 9000
42 #define NEC_TIME_HEADER_OFF 4500
43 #define NEC_TIME_BIT_ON 562
44 #define NEC_TIME_BIT_0_OFF 562
45 #define NEC_TIME_BIT_1_OFF 1687
46 #define NEC_TIME_STOP_ON 562
47 #define NEC_ARR 2104 // (80,000,000 / 38,000) - 1
48 #define NEC_PULSE 1052 // (NEC_ARR + 1) / 2
49 #define RC5_ARR 2221 // (80,000,000 / 36,000) - 1
50 #define RC5_PULSE 1111 // (RC5_ARR + 1) / 2
51 #define RC5_TIME_HALF_BIT 889
52
53 #define MAX_MACRO_STEPS 4
54
55 #define MAX_IR_TIMES 100
56 #define MACRO_STEP_DELAY_MS 150
57
58 #define PROTOCOL_ID_SAMSUNG_NEC 0
59 #define PROTOCOL_ID_LG_NEC 1
60 #define PROTOCOL_ID_PHILIPS_RC5 2
61 #define PROTOCOL_ID_UNKNOWN 0xFF
62
63
64 #ifndef __GNUC__
65 #define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
66 #else
67 #define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
68 #endif /* __GNUC__ */
69 #define FLASH_CONFIG_ADDRESS ((uint32_t)0x080FF800)
70 #define FLASH_CONFIG_PAGE 511
71
72 /* USER CODE END PD */
73
74 /* Private macro -----*/
75 /* USER CODE BEGIN PM */
76
77 /* USER CODE END PM */
78
79 /* Private variables -----*/
80
81 /* USER CODE BEGIN PV */
82 typedef enum {
83     MACRO_IDLE,
84     MACRO_WAIT_IR_FINISH,
85     MACRO_WAIT_DELAY
86 } MacroState_t;
87
88 typedef enum{
89     SENDING_IDLE,
90     SENDING_NEC,
91     SENDING_RC5

```

```

92 } IR_Sending_Protocol_t;
93
94 typedef enum {
95     IR_IDLE = 0,
96     IR_SENDING_HEADER_OFF,
97     IR_SENDING_BIT_ON,
98     IR_SENDING_BIT_OFF,
99     IR_SENDING_STOP_ON
100 } IR_SendState_t;
101
102 typedef struct{
103     uint8_t num_steps;
104     uint16_t codes[MAX_MACRO_STEPS];
105 } ButtonConfig_t;
106
107 typedef struct{
108     uint8_t globalProtocolId;
109     ButtonConfig_t buttons[MAX_BUTTONS];
110 } SystemConfig_t;
111
112 SystemConfig_t currentConfig;
113
114 volatile IR_Sending_Protocol_t current_sending_protocol = SENDING_IDLE;
115
116 uint8_t uart_rx_buffer[RX_BUFFER_SIZE];
117 volatile uint16_t uart_rx_index = 0;
118 volatile uint8_t uart_cmd_received = 0; // Flag al recibir comando completo
119 uint8_t single_byte_rx;
120
121 volatile IR_SendState_t ir_state = IR_IDLE;
122 volatile uint32_t nec_data_to_send = 0;
123 volatile int8_t bit_index = -1; // Índice del bit actual (31 down to 0)}
124 volatile uint8_t start_NEC = 0;
125
126 volatile uint8_t rc5_bit_index = -1;
127 volatile uint8_t rc5_phase = 0;
128 uint8_t rc5_toggle_bit = 0;
129 volatile uint16_t rc5_data_to_send = 0;
130
131 volatile MacroState_t macro_state = MACRO_IDLE;
132 volatile int8_t macro_current_step = 0;
133 volatile uint8_t macro_total_steps = 0;
134 volatile uint16_t* macro_codes_ptr = NULL;
135 volatile uint32_t macro_delay_start_time = 0;
136 volatile uint8_t macro_event_solicited = 0;
137 /* USER CODE END PV */
138
139 /* Private function prototypes -----*/
140 void SystemClock_Config(void);
141 /* USER CODE BEGIN PFP */
142 char Scan_Keypad(void);
143 void Send_NEC_Code(uint8_t address, uint8_t command, uint8_t is_standard_nec);
144 void Send_RC5_Code(uint8_t address, uint8_t command);
145 HAL_StatusTypeDef Config_Save(SystemConfig_t* config);
146 void Config_Load(SystemConfig_t* config);
147 void Send_IR_Command_Global(uint16_t code);

```

```

148 /* USER CODE END PFP */
149
150 /* Private user code -----*/
151 /* USER CODE BEGIN 0 */
152
153 /* USER CODE END 0 */
154
155 /**
156  * @brief The application entry point.
157  * @retval int
158  */
159 int main(void)
160 {
161
162     /* USER CODE BEGIN 1 */
163
164     /* USER CODE END 1 */
165
166     /* MCU Configuration-----*/
167
168     /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
169     HAL_Init();
170
171     /* USER CODE BEGIN Init */
172
173     /* USER CODE END Init */
174
175     /* Configure the system clock */
176     SystemClock_Config();
177
178     /* USER CODE BEGIN SysInit */
179
180     /* USER CODE END SysInit */
181
182     /* Initialize all configured peripherals */
183     MX_GPIO_Init();
184     MX_USART2_UART_Init();
185     MX_USART1_UART_Init();
186     MX_TIM2_Init();
187     MX_TIM3_Init();
188     /* USER CODE BEGIN 2 */
189     HAL_GPIO_TogglePin(GPIOA, IR_TX_Pin);
190     printf("IR Remote Initializing...\r\n");
191
192     Config_Load(&currentConfig); //Cargar configuración desde FLASH
193
194     HAL_UART_Receive_IT(&huart1, &single_byte_rx, 1);
195     printf("IR Remote Ready\r\n");
196     /* USER CODE END 2 */
197
198     /* Infinite loop */
199     /* USER CODE BEGIN WHILE */
200     while (1)
201     {
202         /* USER CODE END WHILE */
203

```

```

204  /* USER CODE BEGIN 3 */
205  // BLUETOOTH
206  if(uart_cmd_received){
207      uart_cmd_received = 0;
208
209      printf("CMD RX: %s", uart_rx_buffer);
210
211      if(strncmp((char*)uart_rx_buffer, "SET_PROTO:", 10) == 0){
212          unsigned int proto_id;
213
214          int params_read = sscanf((char*)uart_rx_buffer, "SET_PROTO:%u\n", &proto_id);
215          if(params_read == 1 && (proto_id <= PROTOCOL_ID_PHILIPS_RC5)){
216              currentConfig.globalProtocolId = (uint8_t)proto_id;
217              printf("Protocolo Global Establecido a: %u\r\n", proto_id);
218              HAL_UART_Transmit(&huart1, (uint8_t*)"PROTO_OK\r\n", 10, HAL_MAX_DELAY);
219          }
220          else{
221              printf("Error: SET_PROTO formato o ID inválido.\r\n");
222              HAL_UART_Transmit(&huart1, (uint8_t*)"ERR_PROTO\r\n", 11, HAL_MAX_DELAY);
223          }
224      }
225      else if(strncmp((char*)uart_rx_buffer, "SET:", 4) == 0){
226          unsigned int btn_id, n_steps;
227          unsigned int hex[MAX_MACRO_STEPS] = {0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF};
228          int params_read = -1;
229
230          sscanf((char*)uart_rx_buffer, "SET:%u:%u:", &btn_id, &n_steps);
231
232          if (n_steps == 1) {
233              params_read = sscanf((char*)uart_rx_buffer, "SET:%u:%u:%x\n", &btn_id, &n_steps
↵ , &hex[0]);
234          } else if (n_steps == 2) {
235              params_read = sscanf((char*)uart_rx_buffer, "SET:%u:%u:%x:%x\n", &btn_id, &
↵ n_steps, &hex[0], &hex[1]);
236          } else if (n_steps == 3) {
237              params_read = sscanf((char*)uart_rx_buffer, "SET:%u:%u:%x:%x:%x\n", &btn_id, &
↵ n_steps, &hex[0], &hex[1], &hex[2]);
238          } else if (n_steps == 4) {
239              params_read = sscanf((char*)uart_rx_buffer, "SET:%u:%u:%x:%x:%x:%x\n", &btn_id,
↵ &n_steps, &hex[0], &hex[1], &hex[2], &hex[3]);
240          } else if (n_steps == 0){ // Borrar configuración del botón
241              params_read = sscanf((char*)uart_rx_buffer, "SET:%u:%u\n", &btn_id, &n_steps);
242          }
243
244          if(params_read == (n_steps + 2) && btn_id < MAX_BUTTONS){
245              currentConfig.buttons[btn_id].num_steps = (uint8_t)n_steps;
246              for(int i = 0; i < MAX_MACRO_STEPS; i++){
247                  currentConfig.buttons[btn_id].codes[i] = (uint16_t)hex[i];
248              }
249              printf("Botón %u configurado con %u pasos.\r\n", btn_id, n_steps);
250              HAL_UART_Transmit(&huart1, (uint8_t*)"SET_OK\r\n", 8, HAL_MAX_DELAY);
251          }
252          else{
253              printf("Error: SET formato inválido. (params_read=%d)\r\n", params_read);
254              HAL_UART_Transmit(&huart1, (uint8_t*)"ERR_SET_FORMAT\r\n", 16, HAL_MAX_DELAY);
255          }

```

```

256     }
257     else if(strncmp((char*)uart_rx_buffer, "GET\n",4) == 0){
258         printf("GET command received. Enviando config...\r\n");
259         char response_buffer[60];
260
261         sprintf(response_buffer, "GLOBAL_PROTO:%d\r\n", currentConfig.globalProtocolId);
262         HAL_UART_Transmit(&huart1, (uint8_t*)response_buffer, strlen(response_buffer),
↪ HAL_MAX_DELAY);
263
264         for (int i = 0; i < MAX_BUTTONS; i++) {
265             if (currentConfig.buttons[i].num_steps > 0) { // Enviar solo si está
↪ configurado
266                 sprintf(response_buffer, "BTN:%d:%d:%04X:%04X:%04X:%04X\r\n",
267                     i,
268                     currentConfig.buttons[i].num_steps,
269                     currentConfig.buttons[i].codes[0],
270                     currentConfig.buttons[i].codes[1],
271                     currentConfig.buttons[i].codes[2],
272                     currentConfig.buttons[i].codes[3]);
273                 HAL_UART_Transmit(&huart1, (uint8_t*)response_buffer, strlen(response_buffer)
↪ , HAL_MAX_DELAY);
274             }
275         }
276         HAL_UART_Transmit(&huart1, (uint8_t*)"GET_END\r\n", 9, HAL_MAX_DELAY);
277         printf("Configuración enviada.\r\n");
278     }
279     else if(strncmp((char*)uart_rx_buffer, "SAVE\n", 5) == 0){
280         printf("SAVE command received. Saving config to Flash...\r\n");
281         HAL_StatusTypeDef save_status = Config_Save(&currentConfig);
282         if(save_status == HAL_OK){
283             printf("Configuración guardada correctamente.\r\n");
284             char* ok_msg = "SAVE_OK\r\n";
285             HAL_UART_Transmit(&huart1, (uint8_t*)ok_msg, strlen(ok_msg), HAL_MAX_DELAY);
286         }
287         else{
288             printf("ERROR al guardar la configuración.\r\n");
289             char* err_msg = "ERR_SAVE\r\n";
290             HAL_UART_Transmit(&huart1, (uint8_t*)err_msg, strlen(err_msg), HAL_MAX_DELAY);
291         }
292     }
293     else if(strncmp((char*)uart_rx_buffer, "CLR_CONFIG\n", 11) == 0){
294         printf("Borrando Configuración...");
295         SystemConfig_t blankConfig;
296         blankConfig.globalProtocolId = currentConfig.globalProtocolId;
297         for(int i=0; i < MAX_BUTTONS; i++){
298             blankConfig.buttons[i].num_steps = 0;
299             for(int j = 0; j < MAX_MACRO_STEPS; j++){
300                 blankConfig.buttons[i].codes[j] = 0xFFFF;
301             }
302         }
303         HAL_StatusTypeDef clr_status = Config_Save(&blankConfig);
304         if(clr_status == HAL_OK){
305             Config_Load(&currentConfig);
306             printf("Configuración borrada.\r\n");
307             HAL_UART_Transmit(&huart1, (uint8_t*)"CLR_OK\r\n", 8, HAL_MAX_DELAY);
308         }

```

```

309     else{
310         printf("Error al borrar la configuración.\r\n");
311         HAL_UART_Transmit(&huart1, (uint8_t*)"ERR_CLR\r\n", 9, HAL_MAX_DELAY);
312     }
313 }
314 else{
315     printf("Comando desconocido: %s\r\n", (char*)uart_rx_buffer);
316     char error_buffer[RX_BUFFER_SIZE + 16];
317     snprintf(error_buffer, sizeof(error_buffer), "ERR_UNKNOWN: %s\r\n", (char*)
↪ uart_rx_buffer);
318     HAL_UART_Transmit(&huart1, (uint8_t*) error_buffer, strlen(error_buffer),
↪ HAL_MAX_DELAY);
319 }
320
321 memset(uart_rx_buffer, 0, RX_BUFFER_SIZE);
322 uart_rx_index = 0;
323
324 HAL_UART_Receive_IT(&huart1, &single_byte_rx, 1);
325 }
326
327 // TECLADO
328 char key = Scan_Keypad();
329 if(key != '\0' && macro_state == MACRO_IDLE && current_sending_protocol ==
↪ SENDING_IDLE){
330     int8_t button_id = -1; // Default invalid
331     if (key >= '1' && key <= '3') button_id = key - '1';
332     else if (key == 'A') button_id = 3;
333     else if (key >= '4' && key <= '6') button_id = key - '4' + 4;
334     else if (key == 'B') button_id = 7;
335     else if (key >= '7' && key <= '9') button_id = key - '7' + 8;
336     else if (key == 'C') button_id = 11;
337     else if (key == '*') button_id = 12;
338     else if (key == '0') button_id = 13;
339     else if (key == '#') button_id = 14;
340     else if (key == 'D') button_id = 15;
341
342     if (button_id != -1 && currentConfig.buttons[button_id].num_steps > 0)
343     {
344         macro_total_steps = currentConfig.buttons[button_id].num_steps;
345         macro_codes_ptr = currentConfig.buttons[button_id].codes;
346
347         macro_event_solicited = 1; // Solicitar inicio
348     }
349     else if (button_id != -1) {
350         printf("Key '%c' (ID %d) not configured.\r\n", key, button_id);
351     }
352 }
353
354 switch(macro_state){
355 case MACRO_IDLE:
356     macro_current_step = 0;
357     if (macro_event_solicited == 1) {
358         macro_event_solicited = 0;
359
360         Send_IR_Command_Global(macro_codes_ptr[macro_current_step]);
361

```

```

362     macro_current_step++;
363     macro_state = MACRO_WAIT_IR_FINISH;
364 }
365 break;
366
367 case MACRO_WAIT_IR_FINISH:
368     if (current_sending_protocol == SENDING_IDLE) {
369         if (macro_current_step < macro_total_steps) {
370             macro_state = MACRO_WAIT_DELAY;
371             macro_delay_start_time = HAL_GetTick();
372         } else {
373             macro_state = MACRO_IDLE;
374         }
375     }
376     break;
377
378 case MACRO_WAIT_DELAY:
379     if (HAL_GetTick() - macro_delay_start_time >= MACRO_STEP_DELAY_MS) {
380         Send_IR_Command_Global(macro_codes_ptr[macro_current_step]);
381         macro_current_step++;
382         macro_state = MACRO_WAIT_IR_FINISH;
383     }
384     break;
385
386 default:
387     macro_current_step = 0;
388     macro_event_solicited = 0;
389     macro_state = MACRO_IDLE;
390     break;
391 }
392 HAL_Delay(10);
393 }
394 /* USER CODE END 3 */
395 }
396
397 /**
398  * @brief System Clock Configuration
399  * @retval None
400  */
401 void SystemClock_Config(void)
402 {
403     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
404     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
405
406     /** Configure the main internal regulator output voltage
407     */
408     if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
409     {
410         Error_Handler();
411     }
412
413     /** Initializes the RCC Oscillators according to the specified parameters
414     * in the RCC_OscInitTypeDef structure.
415     */
416     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
417     RCC_OscInitStruct.HSISState = RCC_HSI_ON;

```

```

418 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
419 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
420 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
421 RCC_OscInitStruct.PLL.PLLM = 1;
422 RCC_OscInitStruct.PLL.PLLN = 10;
423 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
424 RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
425 RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
426 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
427 {
428     Error_Handler();
429 }
430
431 /** Initializes the CPU, AHB and APB buses clocks
432 */
433 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
434                               |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
435 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
436 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
437 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
438 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
439
440 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
441 {
442     Error_Handler();
443 }
444 }
445
446 /* USER CODE BEGIN 4 */
447 PUTCHAR_PROTOTYPE
448 {
449     HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
450     return ch;
451 }
452
453
454 /**
455  * @brief Callback de recepción. Acumula bytes hasta \n.
456  * @param huart UART handle.
457  * @retval None
458  */
459 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
460     if(huart->Instance == USART1){
461         if (uart_cmd_received == 1) {
462             HAL_UART_Receive_IT(&huart1, &single_byte_rx, 1);
463             return;
464         }
465         if(uart_rx_index >= (RX_BUFFER_SIZE -1)){
466             uart_rx_index = 0;
467         }
468         else{
469             uart_rx_buffer[uart_rx_index] = single_byte_rx;
470             if(uart_rx_buffer[uart_rx_index] == '\n'){
471                 uart_rx_buffer[uart_rx_index + 1] = '\0';
472                 uart_cmd_received = 1;
473                 uart_rx_index = 0;

```

```

474     }
475     else{
476         uart_rx_index++;
477     }
478 }
479 HAL_UART_Receive_IT(&huart1, &single_byte_rx, 1);
480
481 }
482 }
483
484 /**
485  * @brief Escanea el teclado matricial 4x4 y devuelve el carácter del botón presionado.
486  * @retval Carácter ('1' a '9', '0', 'A'-'D', '*', '#') si se presiona un botón, '\0' si
487  * ↪ no.
488  */
489 char Scan_Keypad(void) {
490     // Mapa de caracteres para teclado 4x4
491     const char keypad_map[4][4] = {
492         {'1', '2', '3', 'A'},
493         {'4', '5', '6', 'B'},
494         {'7', '8', '9', 'C'},
495         {'*', '0', '#', 'D'}
496     };
497     GPIO_TypeDef* row_ports[] = {GPIOC, GPIOC, GPIOC, GPIOC};
498     uint16_t row_pins[] = {GPIO_PIN_0, GPIO_PIN_1, GPIO_PIN_2, GPIO_PIN_3};
499
500     GPIO_TypeDef* col_ports[] = {GPIOC, GPIOC, GPIOC, GPIOC};
501     uint16_t col_pins[] = {GPIO_PIN_4, GPIO_PIN_5, GPIO_PIN_6, GPIO_PIN_7};
502
503     char pressed_key = '\0';
504
505     for (int r = 0; r < 4; r++){
506         HAL_GPIO_WritePin(row_ports[r], row_pins[r], GPIO_PIN_SET);
507     }
508
509     for (int r = 0; r < 4; r++) {
510         HAL_GPIO_WritePin(row_ports[r], row_pins[r], GPIO_PIN_RESET);
511
512         for (int c = 0; c < 4; c++) {
513             if (HAL_GPIO_ReadPin(col_ports[c], col_pins[c]) == GPIO_PIN_RESET) {
514                 pressed_key = keypad_map[r][c];
515                 // Debouncer
516                 HAL_Delay(50);
517                 if (HAL_GPIO_ReadPin(col_ports[c], col_pins[c]) == GPIO_PIN_RESET) {
518                     while(HAL_GPIO_ReadPin(col_ports[c], col_pins[c]) == GPIO_PIN_RESET);
519                     HAL_GPIO_WritePin(row_ports[r], row_pins[r], GPIO_PIN_SET);
520                     goto key_found;
521                 } else {
522                     pressed_key = '\0';
523                 }
524             }
525         }
526         HAL_GPIO_WritePin(row_ports[r], row_pins[r], GPIO_PIN_SET);
527     }
528 }

```

```

529 key_found:
530     for (int r = 0; r < 4; r++){
531         HAL_GPIO_WritePin(row_ports[r], row_pins[r], GPIO_PIN_SET);
532     }
533
534     return pressed_key;
535 }
536
537 void Send_IR_Command_Global(uint16_t code){
538     if(code == 0xFFFF) return;
539
540     uint8_t addr = (code >> 8) & 0xFF;
541     uint8_t cmd = code & 0xFF;
542
543     switch(currentConfig.globalProtocolId){
544     case PROTOCOL_ID_SAMSUNG_NEC:
545         printf("Sending Samsung (NEC-Var): A=0x%02X, C=0x%02X\r\n", addr, cmd);
546         Send_NEC_Code(addr, cmd, 0);
547         break;
548     case PROTOCOL_ID_LG_NEC:
549         printf("Sending LG (NEC-Std): A=0x%02X, C=0x%02X\r\n", addr, cmd);
550         Send_NEC_Code(addr, cmd, 1);
551         break;
552     case PROTOCOL_ID_PHILIPS_RC5:
553         uint8_t addr = (code >> 6) & 0x1F;
554         uint8_t cmd = code & 0x3F;
555         printf("Sending Philips (RC5): A=%u, C=%u\r\n", addr, cmd);
556         Send_RC5_Code(addr, cmd);
557         break;
558     default:
559         printf("UNKNOWN PROTOCOL");
560     }
561 }
562
563 /**
564  * @brief Inicia el envío de un código NEC IR.
565  * @param address: Dirección del dispositivo (8 bits).
566  * @param command: Comando a enviar (8 bits).
567  * @retval None
568  */
569 void Send_NEC_Code(uint8_t address, uint8_t command, uint8_t is_standard_nec){
570     if(current_sending_protocol != SENDING_IDLE) return;
571
572     HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
573     __HAL_TIM_SET_AUTORELOAD(&htim2, NEC_ARR);
574     __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, NEC_PULSE);
575
576     // Datos invertidos para uso en protocolo
577     uint8_t addr_byte_2 = (is_standard_nec) ? ~address : address;
578     uint8_t inv_command = ~command;
579
580     // Formato de envío de información
581     nec_data_to_send = ((uint32_t)address << 24) |
582         ((uint32_t)addr_byte_2 << 16) |
583         ((uint32_t)command << 8) |
584         (uint32_t)inv_command;

```

```

585
586 bit_index = 31; //Lógica MSB
587 start_NEC = 1;
588 // Detener y resetear contador de TIM3
589 HAL_TIM_Base_Stop_IT(&htim3);
590 __HAL_TIM_SET_COUNTER(&htim3, 0);
591
592 // Interrupción a los 9 ms usando TIM3
593 __HAL_TIM_SET_AUTORELOAD(&htim3, NEC_TIME_HEADER_ON -1);
594 __HAL_TIM_CLEAR_IT(&htim3, TIM_IT_UPDATE);
595 HAL_TIM_Base_Start_IT(&htim3);
596
597 // **CORRECCIÓN: Establecer el estado de protocolo de envío aquí.**
598 current_sending_protocol = SENDING_NEC;
599
600 // Comenzar portadora dse 38 kHz
601 HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
602 }
603
604 /**
605  * @brief Inicia el envío de un código RC5 IR.
606  * @param address: Dirección del dispositivo (5 bits).
607  * @param command: Comando a enviar (6 bits).
608  * @retval None
609  */
610 void Send_RC5_Code(uint8_t address, uint8_t command){
611     if(current_sending_protocol != SENDING_IDLE){
612         return;
613     }
614
615     HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
616     __HAL_TIM_SET_AUTORELOAD(&htim2, RC5_ARR);
617     __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, RC5_PULSE);
618
619     address &= 0x1F;
620     command &= 0x3F;
621
622     rc5_toggle_bit = 1-rc5_toggle_bit;
623
624     rc5_data_to_send = (1 << 13) |
625                       (1 << 12) |
626                       (rc5_toggle_bit << 11) |
627                       ((uint16_t)address << 6) |
628                       (uint16_t)command;
629     rc5_bit_index = 13;
630     rc5_phase = 0;
631     current_sending_protocol = SENDING_RC5;
632
633     HAL_TIM_Base_Stop_IT(&htim3);
634     __HAL_TIM_SET_COUNTER(&htim3, 0);
635     __HAL_TIM_SET_AUTORELOAD(&htim3, RC5_TIME_HALF_BIT -1);
636     __HAL_TIM_CLEAR_IT(&htim3, TIM_IT_UPDATE);
637     HAL_TIM_Base_Start_IT(&htim3);
638 }
639
640

```

```

641 /**
642  * @brief Callback que se ejecuta cuando el período del timer TIM3 expira.
643  * Controla la secuencia de envío del código NEC IR.
644  * @param htim: handle del timer que generó la interrupción.
645  * @retval None
646  */
647 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
648     if(htim->Instance == TIM3){
649         HAL_TIM_Base_Stop_IT(&htim3);
650         __HAL_TIM_SET_COUNTER(&htim3, 0);
651
652         uint16_t next_delay_us = 0; // Duración del próximo estado (ON/OFF)
653         uint8_t start_pwm_next = 0; // Flag para saber si hay que encender la PWM en el pró
        ↪ ximo estado
654
655         if(current_sending_protocol == SENDING_RC5){
656             uint16_t current_bit_value = (rc5_data_to_send >> rc5_bit_index) & 1;
657
658             if(rc5_phase == 0){
659                 rc5_phase = 1;
660                 if(current_bit_value == 1){
661                     HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
662                 }
663                 else{
664                     HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
665                 }
666                 __HAL_TIM_SET_AUTORELOAD(&htim3, RC5_TIME_HALF_BIT -1);
667                 __HAL_TIM_CLEAR_IT(&htim3, TIM_IT_UPDATE);
668                 HAL_TIM_Base_Start_IT(&htim3);
669             }
670             else{ // rc5_phase == 1, transición al inicio del siguiente bit (rc5_phase = 0)
671                 rc5_bit_index--;
672                 rc5_phase = 0;
673                 if(rc5_bit_index < 0){
674                     current_sending_protocol = SENDING_IDLE;
675                     HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
676                 }
677                 else{
678                     current_bit_value = (rc5_data_to_send >> rc5_bit_index) & 1;
679
680                     if(current_bit_value == 1){
681                         // Bit '1' (ON-OFF): La fase 0 debe ser ON
682                         HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
683                     }
684                     else{
685                         // Bit '0' (OFF-ON): La fase 0 debe ser OFF
686                         HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
687                     }
688                     __HAL_TIM_SET_AUTORELOAD(&htim3, RC5_TIME_HALF_BIT -1);
689                     __HAL_TIM_CLEAR_IT(&htim3, TIM_IT_UPDATE);
690                     HAL_TIM_Base_Start_IT(&htim3);
691                 }
692             }
693         }
694         else if(current_sending_protocol == SENDING_NEC){
695             if(ir_state == IR_IDLE){

```

```

696     ir_state = IR_SENDING_HEADER_OFF;
697     next_delay_us = NEC_TIME_HEADER_OFF;
698     start_pwm_next = 0;
699     HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
700 }
701 else {
702     switch(ir_state) {
703     case IR_SENDING_HEADER_OFF: // Terminó OFF de 4.5 ms
704         ir_state = IR_SENDING_BIT_ON;
705         next_delay_us = NEC_TIME_BIT_ON;
706         start_pwm_next = 1; //siguiente estado es ON
707         break;
708     case IR_SENDING_BIT_ON: //Terminó ON de 562 us
709         HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
710         ir_state = IR_SENDING_BIT_OFF;
711         if((nec_data_to_send >> bit_index) & 1){
712             next_delay_us = NEC_TIME_BIT_1_OFF;
713         }
714         else{
715             next_delay_us = NEC_TIME_BIT_0_OFF;
716         }
717         start_pwm_next = 0; //Siguiente estado es OFF
718         break;
719
720     case IR_SENDING_BIT_OFF: // Terminó el OFF de un bit
721         bit_index--;
722         if(bit_index < 0){ //Comprobar si se enviaron los 32 bits
723             ir_state = IR_SENDING_STOP_ON;
724             next_delay_us = NEC_TIME_STOP_ON;
725             start_pwm_next = 1; //Siguiente estado es ON
726         }
727         else{
728             ir_state = IR_SENDING_BIT_ON;
729             next_delay_us = NEC_TIME_BIT_ON;
730             start_pwm_next = 1; // Siguiente estado es ON
731         }
732         break;
733
734     case IR_SENDING_STOP_ON:
735         HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
736         ir_state = IR_IDLE;
737         current_sending_protocol = SENDING_IDLE;
738         return;
739
740     case IR_IDLE: // Ahora esto solo significa que hubo un error de estado
741         current_sending_protocol = SENDING_IDLE;
742         HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
743         break;
744
745     default:
746         ir_state = IR_IDLE;
747         current_sending_protocol = SENDING_IDLE;
748         HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_1);
749         return;
750     }
751 }

```

```

752
753     if(ir_state != IR_IDLE){
754         // Configurar TIM 3 según ir_state
755         __HAL_TIM_SET_AUTORELOAD(&htim3, next_delay_us -1);
756         __HAL_TIM_CLEAR_IT(&htim3, TIM_IT_UPDATE);
757         HAL_TIM_Base_Start_IT(&htim3);
758
759         if(start_pwm_next){
760             HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
761         }
762     }
763 }
764 }
765 }
766
767 /**
768  * @brief Guarda la configuración actual en la memoria Flash.
769  * @param config: Puntero a la estructura SystemConfig_t en RAM que se guardará.
770  * @retval HAL_StatusTypeDef: Estado de la operación HAL (HAL_OK si éxito).
771  */
772 HAL_StatusTypeDef Config_Save(SystemConfig_t* config){
773     HAL_StatusTypeDef status = HAL_ERROR;
774     uint32_t page_error = 0;
775     FLASH_EraseInitTypeDef erase_init;
776     status = HAL_FLASH_Unlock(); // Desbloquear Flash
777     if(status != HAL_OK){
778         printf("Error: Flash Unlock failed!\r\n");
779         return status;
780     }
781
782     erase_init.TypeErase = FLASH_TYPEERASE_PAGES;
783     erase_init.Page = FLASH_CONFIG_PAGE;
784     erase_init.NbPages = 1;
785
786     status = HAL_FLASHEx_Erase(&erase_init, &page_error);
787     if(status != HAL_OK){
788         printf("Error: Flash Erase failed! Page error: %lu\r\n", page_error);
789         HAL_FLASH_Lock();
790         return status;
791     }
792     uint32_t start_address = FLASH_CONFIG_ADDRESS;
793     uint64_t* data_ptr = (uint64_t*) config;
794     uint32_t size_in_bytes = sizeof(SystemConfig_t);
795     uint32_t num_double_words = (size_in_bytes + 7)/8;
796
797     for(uint32_t i = 0; i < num_double_words; i++){
798         status = HAL_FLASH_Program(FLASH_TYPEPROGRAM_DOUBLEWORD, start_address + (i*8),
799             ↪ data_ptr[i]);
800         if(status != HAL_OK){
801             printf("ERROR: Flash Program failed at address 0x%lX\r\n", start_address+(i*8));
802             HAL_FLASH_Lock();
803             return status;
804         }
805     }
806     status = HAL_FLASH_Lock();
807     if(status != HAL_OK){

```

```

807     printf("ERROR: Flash Lock failed!\r\n");
808 }
809 else{
810     printf("Configuración guardada en Flash. \r\n");
811 }
812 return status;
813 }
814
815 /**
816  * @brief Carga la configuración desde la memoria Flash a la RAM.
817  * @param config: Puntero a la estructura SystemConfig_t en RAM donde se cargará la
818  *               ↪ configuración.
819  * @retval None. Si no hay config válida, carga valores por defecto.
820  */
821 void Config_Load(SystemConfig_t* config){
822     SystemConfig_t* flash_config = (SystemConfig_t*)FLASH_CONFIG_ADDRESS;
823     if (flash_config->globalProtocolId == 0xFF) { // Si la Flash está borrada
824         printf("No se encontró config válida. Cargando defaults.\r\n");
825         config->globalProtocolId = PROTOCOL_ID_SAMSUNG_NEC;
826         for(int i = 0; i < MAX_BUTTONS; i++) {
827             config->buttons[i].num_steps = 0; // 0 pasos = no configurado
828             for (int j = 0; j < MAX_MACRO_STEPS; j++) {
829                 config->buttons[i].codes[j] = 0xFFFF; // Llenar con códigos nulos
830             }
831         }
832     }
833     else{
834         memcpy(config, flash_config, sizeof(SystemConfig_t));
835         printf("Configuración cargada desde Flash.\r\n");
836     }
837 }
838
839 /* USER CODE END 4 */
840
841 /**
842  * @brief This function is executed in case of error occurrence.
843  * @retval None
844  */
845 void Error_Handler(void)
846 {
847     /* USER CODE BEGIN Error_Handler_Debug */
848     /* User can add his own implementation to report the HAL error return state */
849     __disable_irq();
850     while (1)
851     {
852     }
853     /* USER CODE END Error_Handler_Debug */
854 }
855 #ifdef USE_FULL_ASSERT
856 /**
857  * @brief Reports the name of the source file and the source line number
858  *        where the assert_param error has occurred.
859  * @param file: pointer to the source file name
860  * @param line: assert_param error line source number
861  * @retval None

```

```

862  */
863 void assert_failed(uint8_t *file, uint32_t line)
864 {
865     /* USER CODE BEGIN 6 */
866     /* User can add his own implementation to report the file name and line number,
867        ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
868     /* USER CODE END 6 */
869 }
870 #endif /* USE_FULL_ASSERT */

```

Listing A.1: Archivo principal del firmware (main.c)

Apéndice B

Código Fuente de la App (MainActivity.java)

Código de la actividad principal de Android. Gestiona la conexión Bluetooth, la interfaz de grilla de botones y el procesamiento de los comandos recibidos desde el hardware.

```
1 package com.example.ir_remote_config;
2
3 import android.app.AlertDialog;
4 import android.os.Bundle;
5
6 import androidx.activity.EdgeToEdge;
7 import androidx.annotation.NonNull;
8 import androidx.appcompat.app.AppCompatActivity;
9 import androidx.core.graphics.Insets;
10 import androidx.core.view.ViewCompat;
11 import androidx.core.view.WindowInsetsCompat;
12
13 import androidx.appcompat.app.AppCompatActivity;
14 import android.os.Bundle;
15 import android.util.Log;
16 import android.view.View; // Import View
17 import android.widget.AdapterView;
18 import android.widget.Button; // Import Button
19 import android.widget.TextView;
20 import android.widget.Toast; // Import Toast for messages
21
22 import android.bluetooth.BluetoothAdapter;
23 import android.bluetooth.BluetoothManager;
24 import android.content.Intent;
25 import android.os.Build; // For version checks
26 import androidx.activity.result.ActivityResultLauncher; // For requesting permissions/
    ↪ enable BT
27 import androidx.activity.result.contract.ActivityResultContracts;
28 import androidx.core.app.ActivityCompat; // For checking/requesting permissions
29 import android.Manifest; // To reference permission strings
30 import android.content.pm.PackageManager; // To check permissions
31 import android.bluetooth.BluetoothDevice; // Importar BluetoothDevice
32 import android.bluetooth.BluetoothSocket; // Importar BluetoothSocket
```

```

33 import java.io.IOException; // Para manejo de errores de conexión
34 import java.io.OutputStream; // Para enviar datos
35 import java.util.Arrays;
36 import java.util.List;
37 import java.util.Set; // Para la lista de dispositivos
38 import java.util.UUID; // Para el UUID de SPP
39 import android.widget.AdapterView; // Para mostrar lista (opcional)
40 import android.widget.Spinner; // Para mostrar lista (opcional)
41 import java.util.ArrayList; // Para la lista de dispositivos
42
43 import android.os.Handler;
44 import android.os.Looper;
45 import android.os.Message;
46 import java.io.InputStream; // For reading data
47 import java.nio.charset.StandardCharsets; // For converting bytes to string
48
49 import org.json.JSONArray; // Para parsear JSON
50 import org.json.JSONObject; // Para parsear JSON
51 import java.io.InputStream; // Para leer assets
52 import java.util.List; // Para listas
53 import java.util.ArrayList; // Para listas
54 import java.util.Map; // Para mapas
55 import java.util.HashMap; // Para mapas
56
57 public class MainActivity extends AppCompatActivity implements
    ↪ ConfigureButtonDialogFragment.ConfigureButtonDialogListener{
58     Button button1, button2, button3, buttonA;
59     Button button4, button5, button6, buttonB;
60     Button button7, button8, button9, buttonC;
61     Button buttonStar, button0, buttonHash, buttonD;
62
63     Button buttonRetryConnection;
64
65     Button buttonGetConfig, buttonSaveConfig, buttonClearConfig;
66     TextView textViewSelectedButtonInfo;
67     TextView textViewConnectionStatus;
68
69     Spinner spinnerSelectTV;
70     private static final int PROTOCOL_ID_SAMSUNG_NEC = 0;
71     private static final int PROTOCOL_ID_LG_NEC = 1;
72     private static final int PROTOCOL_ID_PHILIPS_RC5 = 2;
73     private final String[] tvProtocolNames = {"Samsung (NEC Var)", "LG (NEC Std)", "
    ↪ Phillips (RC5)"};
74     private int currentGlobalProtocolId = -1;
75     private boolean programmaticSpinnerChange = false;
76     private BluetoothAdapter bluetoothAdapter;
77     private static final int REQUEST_ENABLE_BT = 1;
78     private static final int REQUEST_BLUETOOTH_PERMISSIONS = 2;
79
80     private ActivityResultLauncher<Intent> enableBtLauncher;
81     private ActivityResultLauncher<String[]> requestPermissionsLauncher;
82
83     private ArrayList<BluetoothDevice> pairedDevicesList = new ArrayList<>();
84     private ArrayAdapter<String> pairedDeviceArrayAdapter;
85     private BluetoothDevice hc05Device = null;
86     private BluetoothSocket btSocket = null;

```

```

87 private OutputStream outputStream = null;
88 private static final UUID MY_UUID_SPP = UUID.fromString("
↪ 00001101-0000-1000-8000-00805F9B34FB");
89
90 private ConnectedThread connectedThread = null;
91 private Handler uiHandler;
92
93 public static final int MESSAGE_READ = 0;
94 public static final int MESSAGE_TOAST = 1;
95 public static final int MESSAGE_CONNECTION_LOST = 2;
96 public static final int MESSAGE_CONNECTION_FAILED = 3;
97
98 private static final int MAX_BUTTONS_APP = 16;
99 private static final int MAX_MACRO_STEPS_APP = 4;
100 private static class AppButtonConfig {
101     int numSteps = 0;
102     String[] hexCodes = new String[MAX_MACRO_STEPS_APP];
103 }
104
105 private AppButtonConfig[] appButtonConfigs = new AppButtonConfig[MAX_BUTTONS_APP];
106 private List<IRDevice> irDeviceList = new ArrayList<>();
107 private List<String> deviceBrandNames = new ArrayList<>();
108 private Map<String, String> currentCommandMap = new HashMap<>();
109 private ArrayList<String> currentCommandNames = new ArrayList<>();
110 private final String CMD_NONE = "Ninguno (FFFF)";
111
112
113 @Override
114 protected void onCreate(Bundle savedInstanceState) {
115     super.onCreate(savedInstanceState);
116     setContentView(R.layout.activity_main);
117
118     spinnerSelectTV = findViewById(R.id.spinnerSelectTV);
119     buttonRetryConnection = findViewById(R.id.buttonRetryConnection);
120     loadIRCodesFromJson();
121
122     for (int i = 0; i < MAX_BUTTONS_APP; i++) {
123         appButtonConfigs[i] = new AppButtonConfig();
124         appButtonConfigs[i].numSteps = 0;
125         Arrays.fill(appButtonConfigs[i].hexCodes, "FFFF");
126     }
127
128     // Botones
129     button1 = findViewById(R.id.button1);
130     button2 = findViewById(R.id.button2);
131     button3 = findViewById(R.id.button3);
132     buttonA = findViewById(R.id.buttonA);
133     button4 = findViewById(R.id.button4);
134     button5 = findViewById(R.id.button5);
135     button6 = findViewById(R.id.button6);
136     buttonB = findViewById(R.id.buttonB);
137     button7 = findViewById(R.id.button7);
138     button8 = findViewById(R.id.button8);
139     button9 = findViewById(R.id.button9);
140     buttonC = findViewById(R.id.buttonC);
141     buttonStar = findViewById(R.id.buttonStar);

```

```

142     button0 = findViewById(R.id.button0);
143     buttonHash = findViewById(R.id.buttonHash);
144     buttonD = findViewById(R.id.buttonD);
145
146     buttonClearConfig = findViewById(R.id.buttonClearConfig);
147     buttonGetConfig = findViewById(R.id.buttonSendGET);
148     buttonSaveConfig = findViewById(R.id.buttonSendSAVE);
149     textViewSelectedButtonInfo = findViewById(R.id.textViewSelectedButtonInfo);
150     textViewConnectionStatus = findViewById(R.id.textViewConnectionStatus);
151
152     buttonRetryConnection.setOnClickListener(v -> {
153         Log.d("Bluetooth", "Botón Reintentar presionado.");
154         buttonRetryConnection.setVisibility(View.GONE);
155         textViewConnectionStatus.setText("Estado: Reintentando conexión...");
156         listPairedDevices();
157     });
158
159     textViewConnectionStatus.setText("Estado: Desconectado");
160     buttonRetryConnection.setVisibility(View.VISIBLE);
161
162     ArrayAdapter<String> tvSpinnerAdapter = new ArrayAdapter<>(this,
163         android.R.layout.simple_spinner_item, deviceBrandNames);
164     tvSpinnerAdapter.setDropDownViewResource(android.R.layout.
165 ↪ simple_spinner_dropdown_item);
166     spinnerSelectTV.setAdapter(tvSpinnerAdapter);
167
168     spinnerSelectTV.setOnItemClickListener(new AdapterView.OnItemClickListener
169 ↪ () {
170         @Override
171         public void onItemClick(AdapterView<?> parent, View view, int position,
172 ↪ long id) {
173             if (programmaticSpinnerChange) {
174                 return;
175             }
176             // Obtener el dispositivo seleccionado de nuestra lista
177             IRDevice selectedDevice = irDeviceList.get(position);
178             int newProtocolId = selectedDevice.protocolId;
179
180             // Actualizar las listas de comandos actuales
181             currentCommandMap.clear();
182             currentCommandNames.clear();
183             for (IRCommand cmd : selectedDevice.commands) {
184                 currentCommandMap.put(cmd.name, cmd.hex);
185                 currentCommandNames.add(cmd.name);
186             }
187             Log.d("Spinner", "TV Seleccionado: " + selectedDevice.brand);
188             if (newProtocolId != currentGlobalProtocolId) {
189                 currentGlobalProtocolId = newProtocolId;
190                 String command = "SET_PROTO:" + currentGlobalProtocolId + "\n";
191                 Log.d("Bluetooth", "Spinner selected by user. Sending: " + command.
192 ↪ trim());
193
194                 sendBluetoothCommand(command);
195                 clearLocalConfigAndUI();
196             }
197         }
198     });

```

```

194         @Override public void onNothingSelected(AdapterView<?> parent) { }
195     });
196
197     // Listeners
198     button1.setOnClickListener(v ->{
199         int btnId = 0;
200         displayButtonInfo(btnId);
201     });
202
203     button1.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
204 ↪ del botón
205         @Override
206         public boolean onLongClick(View v){
207             int btnId = 0;
208             String btnLabel = ((Button) v).getText().toString();
209
210             String[] currentCodes = appButtonConfigs[btnId].hexCodes;
211
212             ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
213 ↪ newInstance(
214                 btnId, btnLabel,
215                 currentCodes, // Pasa el array de códigos hex
216                 currentCommandNames
217             );
218             dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
219 ↪ ;
220             return true;
221         }
222     });
223
224     button2.setOnClickListener(v ->{
225         int btnId = 1;
226         displayButtonInfo(btnId);
227     });
228
229     button2.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
230 ↪ del botón
231         @Override
232         public boolean onLongClick(View v){
233             int btnId = 1;
234             String btnLabel = ((Button) v).getText().toString();
235             String[] currentCodes = appButtonConfigs[btnId].hexCodes;
236
237             ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
238 ↪ newInstance(
239                 btnId, btnLabel,
240                 currentCodes, // Pasa el array de códigos hex
241                 currentCommandNames
242             );
243             dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
244 ↪ ;
245             return true;
246         }
247     });
248
249     button3.setOnClickListener(v ->{

```

```

244         int btnId = 2;
245         displayButtonInfo(btnId);
246     });
247
248     button3.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
249 ↪ del botón
250         @Override
251         public boolean onLongClick(View v){
252             int btnId = 2;
253             String btnLabel = ((Button) v).getText().toString();
254
255             String[] currentCodes = appButtonConfigs[btnId].hexCodes;
256
257             ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
258 ↪ newInstance(
259                 btnId, btnLabel,
260                 currentCodes, // Pasa el array de códigos hex
261                 currentCommandNames
262             );
263             dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
264 ↪ ;
265             return true;
266         }
267     });
268
269     buttonA.setOnClickListener(v ->{
270         int btnId = 3;
271         displayButtonInfo(btnId);
272     });
273
274     buttonA.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
275 ↪ del botón
276         @Override
277         public boolean onLongClick(View v){
278             int btnId = 3;
279             String btnLabel = ((Button) v).getText().toString();
280
281             String[] currentCodes = appButtonConfigs[btnId].hexCodes;
282
283             ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
284 ↪ newInstance(
285                 btnId, btnLabel,
286                 currentCodes, // Pasa el array de códigos hex
287                 currentCommandNames
288             );
289             dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
290 ↪ ;
291             return true;
292         }
293     });
294
295     button4.setOnClickListener(v ->{
296         int btnId = 4;
297         displayButtonInfo(btnId);
298     });

```

```

294         button4.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
295             @Override
296             public boolean onLongClick(View v){
297                 int btnId = 4;
298                 String btnLabel = ((Button) v).getText().toString();
299
300                 String[] currentCodes = appButtonConfigs[btnId].hexCodes;
301
302                 ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
303                     btnId, btnLabel,
304                     currentCodes, // Pasa el array de códigos hex
305                     currentCommandNames
306                 );
307                 dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
308                 return true;
309             }
310         });
311
312         button5.setOnClickListener(v ->{
313             int btnId = 5;
314             displayButtonInfo(btnId);
315         });
316
317         button5.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
318             @Override
319             public boolean onLongClick(View v){
320                 int btnId = 5;
321                 String btnLabel = ((Button) v).getText().toString();
322
323                 String[] currentCodes = appButtonConfigs[btnId].hexCodes;
324
325                 ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
326                     btnId, btnLabel,
327                     currentCodes, // Pasa el array de códigos hex
328                     currentCommandNames
329                 );
330                 dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
331                 return true;
332             }
333         });
334
335         button6.setOnClickListener(v ->{
336             int btnId = 6;
337             displayButtonInfo(btnId);
338         });
339
340         button6.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
341             @Override
342             public boolean onLongClick(View v){

```

```

343         int btnId = 6;
344         String btnLabel = ((Button) v).getText().toString();
345
346         String[] currentCodes = appButtonConfigs[btnId].hexCodes;
347
348         ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
349             btnId, btnLabel,
350             currentCodes, // Pasa el array de códigos hex
351             currentCommandNames
352         );
353         dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
354         return true;
355     }
356 });
357
358 buttonB.setOnClickListener(v ->{
359     int btnId = 7;
360     displayButtonInfo(btnId);
361 });
362
363 buttonB.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
364     @Override
365     public boolean onLongClick(View v){
366         int btnId = 7;
367         String btnLabel = ((Button) v).getText().toString();
368
369         String[] currentCodes = appButtonConfigs[btnId].hexCodes;
370
371         ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
372             btnId, btnLabel,
373             currentCodes, // Pasa el array de códigos hex
374             currentCommandNames
375         );
376         dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
377         return true;
378     }
379 });
380
381 button7.setOnClickListener(v ->{
382     int btnId = 8;
383     displayButtonInfo(btnId);
384 });
385
386 button7.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
387     @Override
388     public boolean onLongClick(View v){
389         int btnId = 8;
390         String btnLabel = ((Button) v).getText().toString();
391
392         String[] currentCodes = appButtonConfigs[btnId].hexCodes;

```

```

393
394         ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
395             btnId, btnLabel,
396             currentCodes, // Pasa el array de códigos hex
397             currentCommandNames
398         );
399         dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
400         return true;
401     }
402 });
403
404     button8.setOnClickListener(v ->{
405         int btnId = 9;
406         displayButtonInfo(btnId);
407     });
408
409     button8.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
410         @Override
411         public boolean onLongClick(View v){
412             int btnId = 9;
413             String btnLabel = ((Button) v).getText().toString();
414
415             String[] currentCodes = appButtonConfigs[btnId].hexCodes;
416
417             ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
418                 btnId, btnLabel,
419                 currentCodes, // Pasa el array de códigos hex
420                 currentCommandNames
421             );
422             dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
423             return true;
424         }
425     });
426
427     button9.setOnClickListener(v ->{
428         int btnId = 10;
429         displayButtonInfo(btnId);
430     });
431
432     button9.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
433         @Override
434         public boolean onLongClick(View v){
435             int btnId = 10;
436             String btnLabel = ((Button) v).getText().toString();
437
438             String[] currentCodes = appButtonConfigs[btnId].hexCodes;
439
440             ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
441                 btnId, btnLabel,

```

```

442         currentCodes, // Pasa el array de códigos hex
443         currentCommandNames
444     );
445     dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
446     return true;
447 }
448 });
449
450 buttonC.setOnClickListener(v ->{
451     int btnId = 11;
452     displayButtonInfo(btnId);
453 });
454
455 buttonC.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
456     @Override
457     public boolean onLongClick(View v){
458         int btnId = 11;
459         String btnLabel = ((Button) v).getText().toString();
460
461         String[] currentCodes = appButtonConfigs[btnId].hexCodes;
462
463         ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
464             btnId, btnLabel,
465             currentCodes, // Pasa el array de códigos hex
466             currentCommandNames
467         );
468         dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
↪ ;
469         return true;
470     }
471 });
472
473 buttonStar.setOnClickListener(v ->{
474     int btnId = 12;
475     displayButtonInfo(btnId);
476 });
477
478 buttonStar.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
↪ del botón
479     @Override
480     public boolean onLongClick(View v){
481         int btnId = 12;
482         String btnLabel = ((Button) v).getText().toString();
483
484         String[] currentCodes = appButtonConfigs[btnId].hexCodes;
485
486         ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
↪ newInstance(
487             btnId, btnLabel,
488             currentCodes, // Pasa el array de códigos hex
489             currentCommandNames
490         );
491         dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")

```

```

492         ↪ ;
493         return true;
494     }
495 });
496
497 button0.setOnClickListener(v ->{
498     int btnId = 13;
499     displayButtonInfo(btnId);
500 });
501
502 button0.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
503     ↪ del botón
504     @Override
505     public boolean onLongClick(View v){
506         int btnId = 13;
507         String btnLabel = ((Button) v).getText().toString();
508
509         String[] currentCodes = appButtonConfigs[btnId].hexCodes;
510
511         ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
512     ↪ newInstance(
513             btnId, btnLabel,
514             currentCodes, // Pasa el array de códigos hex
515             currentCommandNames
516         );
517         dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
518     ↪ ;
519         return true;
520     }
521 });
522
523 buttonHash.setOnClickListener(v ->{
524     int btnId = 14;
525     displayButtonInfo(btnId);
526 });
527
528 buttonHash.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
529     ↪ del botón
530     @Override
531     public boolean onLongClick(View v){
532         int btnId = 14;
533         String btnLabel = ((Button) v).getText().toString();
534
535         String[] currentCodes = appButtonConfigs[btnId].hexCodes;
536
537         ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
538     ↪ newInstance(
539             btnId, btnLabel,
540             currentCodes, // Pasa el array de códigos hex
541             currentCommandNames
542         );
543         dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
544     ↪ ;
545         return true;
546     }
547 });

```

```

541
542     buttonD.setOnClickListener(v ->{
543         int btnId = 15;
544         displayButtonInfo(btnId);
545     });
546
547     buttonD.setOnLongClickListener(new View.OnLongClickListener(){ //Presión larga
548 ↪ del botón
549         @Override
550         public boolean onLongClick(View v){
551             int btnId = 15;
552             String btnLabel = ((Button) v).getText().toString();
553
554             String[] currentCodes = appButtonConfigs[btnId].hexCodes;
555
556             ConfigureButtonDialogFragment dialog = ConfigureButtonDialogFragment.
557 ↪ newInstance(
558                 btnId, btnLabel,
559                 currentCodes, // Pasa el array de códigos hex
560                 currentCommandNames
561             );
562             dialog.show(getSupportFragmentManager(), "ConfigureButtonDialogFragment")
563 ↪ ;
564             return true;
565         }
566     });
567
568     buttonGetConfig.setOnClickListener(v -> {
569         Log.d("Bluetooth", "Botón GET presionado. Enviando GET...");
570         sendBluetoothCommand("GET\n");
571     });
572
573     buttonSaveConfig.setOnClickListener(v -> {
574         Log.d("Bluetooth", "Botón SAVE presionado. Enviando SAVE...");
575         sendBluetoothCommand("SAVE\n");
576     });
577
578     buttonClearConfig.setOnClickListener(v -> {
579         // Opcional: Pedir confirmación antes de borrar
580         new AlertDialog.Builder(MainActivity.this)
581             .setTitle("Borrar Configuración")
582             .setMessage("¿Estás seguro de que quieres borrar toda la configuració
583 ↪ n guardada en el control remoto?")
584             .setPositiveButton("Borrar", (dialog, which) -> {
585                 Log.d("Bluetooth", "Clear button confirmed. Sending CLR_CONFIG...
586 ↪ ");
587                 sendBluetoothCommand("CLR_CONFIG\n");
588             })
589             .setNegativeButton("Cancelar", null)
590             .show();
591     });
592
593     BluetoothManager bluetoothManager = getSystemService(BluetoothManager.class);
594     if(bluetoothManager != null){
595         bluetoothAdapter = bluetoothManager.getAdapter();
596     }

```

```

592
593     if(blueetoothAdapter == null){
594         showToast("El dispositivo no soporta Bluetooth");
595         return;
596     }
597
598     setupActivityResultLaunchers();
599     checkBluetoothStateAndPermissions();
600
601     uiHandler = new Handler(Looper.getMainLooper()){
602         @Override
603         public void handleMessage(@NonNull Message msg){
604             switch(msg.what){
605                 case MESSAGE_READ:
606                     byte[] readBuf = (byte[]) msg.obj;
607                     String receivedData = new String(readBuf, 0, msg.arg1,
↪ StandardCharsets.UTF_8);
608                     String[] lines = receivedData.split("\\r?\\n");
609
610                     for(String line: lines) {
611                         if (line.isEmpty()) continue;
612                         Log.d("BluetoothRCV", "Procesando Línea: " + line);
613                         if (line.startsWith("GLOBAL_PROTO:")) {
614                             String[] parts = line.split(":");
615                             if (parts.length == 2) { // GLOBAL_PROTO:id
616                                 try {
617                                     int protoId = Integer.parseInt(parts[1]);
618                                     if (protoId >= 0 && protoId < tvProtocolNames.
↪ length) {
619                                         currentGlobalProtocolId = protoId;
620                                         Log.i("ConfigUpdate", "Global Proto received:
↪ " + protoId + " (" + tvProtocolNames[protoId] + ")");
621
622                                         programmaticSpinnerChange = true;
623                                         spinnerSelectTV.setSelection(protoId);
624                                         programmaticSpinnerChange = false;
625
626                                     } else {
627                                         Log.w("ConfigParse", "Invalid Global Protocol
↪ ID received: " + line);
628                                     }
629                                     } catch (NumberFormatException e) {
630                                         Log.e("ConfigParse", "Error parsing GLOBAL_PROTO:
↪ " + line, e);
631                                     }
632                                 } else {
633                                     Log.w("ConfigParse", "Incorrect GLOBAL_PROTO format:
↪ " + line);
634                                 }
635                             }
636                             else if (line.startsWith("BTN:")) {
637                                 // Formato: BTN:<id>:<n_steps>:<c1>:<c2>:<c3>:<c4>
638                                 String[] parts = line.split(":");
639                                 if (parts.length == 7) { // Esperar 7 partes (BTN, id, n,
↪ c1, c2, c3, c4)
640                                     try {

```

```

641         int buttonId = Integer.parseInt(parts[1]);
642         int numSteps = Integer.parseInt(parts[2]);
643
644         if (buttonId >= 0 && buttonId < 16) {
645             appButtonConfigs[buttonId].numSteps =
↪ numSteps;
646             appButtonConfigs[buttonId].hexCodes[0] =
↪ parts[3].toUpperCase();
647             appButtonConfigs[buttonId].hexCodes[1] =
↪ parts[4].toUpperCase();
648             appButtonConfigs[buttonId].hexCodes[2] =
↪ parts[5].toUpperCase();
649             appButtonConfigs[buttonId].hexCodes[3] =
↪ parts[6].toUpperCase();
650             Log.i("ConfigUpdate", "Stored Btn " +
↪ buttonId + ": N=" + numSteps);
651         }
652         else {
653             Log.w("ConfigParse", "Data recibida inválida:
↪ " + line);
654         }
655         } catch (NumberFormatException e) {
656             Log.e("ConfigParse", "Error parsing Numbers: " +
↪ line, e);
657         }
658         } else {
659             Log.w("ConfigParse", "Formato recibido incorrecto: "
↪ + line);
660         }
661         } else if(line.startsWith("GET_END")){
662             Log.i("ConfigUpdate", "GET_END recibido. Actualizando UI
↪ de los botones.");
663             updateAllButtonUIs();
664             textViewSelectedButtonInfo.setText("Seleccione un botón."
↪ );
665         }
666         else if(line.startsWith("SET_OK")){
667             showToast("Configuración exitosa");
668         }
669         else if(line.startsWith("SAVE_OK")){
670             showToast("Configuración guardada");
671         }
672         else if(line.startsWith("PROTO_OK")){
673             Log.i("Bluetooth", "Protocolo actualizado en STM32.
↪ Solicitando config...");
674             sendBluetoothCommand("GET\n");
675         }
676         else if(line.startsWith("ERR_")){
677             showToast("Error del dispositivo: " + line);
678         }
679         else if(line.startsWith("CLR_OK")){
680             showToast("Configuración borrada exitosamente.");
681             clearLocalConfigAndUI();
682         }
683         else{
684             Log.d("BluetoothRCV", "Mensaje: " + line);

```

```

685         }
686     }
687
688     break;
689     case MESSAGE_TOAST:
690         showToast(msg.getData().getString("toast"));
691         break;
692     case MESSAGE_CONNECTION_LOST:
693         showToast("Bluetooth connection lost.");
694         textViewConnectionStatus.setText("Estado: Conexión Perdida");
695         buttonRetryConnection.setVisibility(View.VISIBLE);
696         break;
697     case MESSAGE_CONNECTION_FAILED:
698         showToast("Bluetooth connection failed.");
699         textViewConnectionStatus.setText("Estado: Conexión Fallida");
700         buttonRetryConnection.setVisibility(View.VISIBLE);
701         break;
702     }
703 }
704 };
705 }
706
707 private void setupActivityResultLaunchers(){
708     enableBtLauncher = registerForActivityResult(
709         new ActivityResultContracts.StartActivityForResult(),
710         result -> {
711             if(result.getResultCode() == RESULT_OK){
712                 showToast("Bluetooth habilitado");
713             }
714             else{
715                 showToast("Habilitación de Bluetooth cancelada");
716             }
717         });
718
719     requestPermissionsLauncher = registerForActivityResult(
720         new ActivityResultContracts.RequestMultiplePermissions(), isGrantedMap ->
721         ↪ {
722             boolean allGranted = true;
723             for(Boolean granted : isGrantedMap.values()){
724                 if(!granted){
725                     allGranted = false;
726                     break;
727                 }
728             }
729
730             if(allGranted){
731                 showToast("Permisos de Bluetooth concedidos");
732                 checkBluetoothEnabled();
733             }
734             else{
735                 showToast("Permisos de Bluetooth denegados");
736             }
737         });
738
739     private void checkBluetoothStateAndPermissions(){

```

```

740         if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.S){
741             if(ActivityCompat.checkSelfPermission(this, Manifest.permission.
↪ BLUETOOTH_SCAN) != PackageManager.PERMISSION_GRANTED ||
742                 ActivityCompat.checkSelfPermission(this, Manifest.permission.
↪ BLUETOOTH_CONNECT) != PackageManager.PERMISSION_GRANTED){
743                 requestPermissionsLauncher.launch(new String[]{
744                     Manifest.permission.BLUETOOTH_SCAN,
745                     Manifest.permission.BLUETOOTH_CONNECT
746                 });
747                 return;
748             }
749         }
750         else{
751             if(ActivityCompat.checkSelfPermission(this, Manifest.permission.
↪ ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED ||
752                 ActivityCompat.checkSelfPermission(this, Manifest.permission.
↪ BLUETOOTH_ADMIN) != PackageManager.PERMISSION_GRANTED){
753                 requestPermissionsLauncher.launch(new String[]{
754                     Manifest.permission.ACCESS_FINE_LOCATION,
755                     Manifest.permission.BLUETOOTH_ADMIN
756                 });
757                 return;
758             }
759
760         }
761         checkBluetoothEnabled();
762     }
763
764     private void checkBluetoothEnabled(){
765         if(!bluetoothAdapter.isEnabled()){
766             Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
767
768             if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.S){
769                 if(ActivityCompat.checkSelfPermission(this, Manifest.permission.
↪ BLUETOOTH_CONNECT) != PackageManager.PERMISSION_GRANTED){
770                     showToast("Se requiere permiso BLUETOOTH_CONNECT para habilitar
↪ Bluetooth");
771                     return;
772                 }
773             }
774             enableBtLauncher.launch(enableBtIntent);
775         }
776         else{
777             showToast("Bluetooth ya se encuentra habilitado y con los permisos concedidos
↪ ");
778             listPairedDevices();
779             //TODO: Conectarse al HC-05
780         }
781     }
782
783     @Override
784     public void onDialogPositiveClick(int buttonId, String command1Name, String
↪ command2Name, String command3Name, String command4Name) {          String hex1 =
↪ currentCommandMap.getOrDefault(command1Name, "FFFF");
785         String hex2 = currentCommandMap.getOrDefault(command2Name, "FFFF");
786     }

```

```

787     String hex3 = currentCommandMap.getOrDefault(command3Name, "FFFF");
788     String hex4 = currentCommandMap.getOrDefault(command4Name, "FFFF");
789
790     int numSteps = 0;
791     if (!hex1.equals("FFFF")) numSteps++;
792     if (!hex2.equals("FFFF")) numSteps++;
793     if (!hex3.equals("FFFF")) numSteps++;
794     if (!hex4.equals("FFFF")) numSteps++;
795
796
797     if (command1Name.equals(CMD_NONE) || hex1.equals("FFFF")) numSteps = 0;
798     if (command2Name.equals(CMD_NONE) || hex2.equals("FFFF")) { hex2 = "FFFF"; if (
↪ numSteps > 1) numSteps = 1; }
799     if (command3Name.equals(CMD_NONE) || hex3.equals("FFFF")) { hex3 = "FFFF"; if (
↪ numSteps > 2) numSteps = 2; }
800     if (command4Name.equals(CMD_NONE) || hex4.equals("FFFF")) { hex4 = "FFFF"; if (
↪ numSteps > 3) numSteps = 3; }
801
802     String command = String.format(java.util.Locale.US, "SET:%d:%d:%s:%s:%s:%s\n",
803         buttonId, numSteps, hex1, hex2, hex3, hex4);
804
805     showToast("Enviando: " + command.trim());
806     sendBluetoothCommand(command);
807
808     if (buttonId >= 0 && buttonId < 16) {
809         appButtonConfigs[buttonId].numSteps = numSteps;
810         appButtonConfigs[buttonId].hexCodes[0] = hex1;
811         appButtonConfigs[buttonId].hexCodes[1] = hex2;
812         appButtonConfigs[buttonId].hexCodes[2] = hex3;
813         appButtonConfigs[buttonId].hexCodes[3] = hex4;
814         updateButtonUI(buttonId);
815     }
816 }
817
818 private void listPairedDevices(){
819     if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.S &&
820         ActivityCompat.checkSelfPermission(this, Manifest.permission.BLUETOOTH_CONNECT
↪ ) != PackageManager.PERMISSION_GRANTED){
821         showToast("Se requiere permiso BLUETOOTH_CONNECT para listar dispositivos");
822         return;
823     }
824
825     Set<BluetoothDevice> pairedDevices = bluetoothAdapter.getBondedDevices();
826     pairedDevicesList.clear();
827
828     if(!pairedDevices.isEmpty()){
829         Log.d("Bluetooth", "Paired devices found");
830         for(BluetoothDevice device : pairedDevices){
831             pairedDevicesList.add(device);
832             String deviceName = device.getName();
833             String deviceAddress = device.getAddress();
834             Log.d("Bluetooth", "- " + deviceName + "[" + deviceAddress + "]");
835
836             if(deviceName != null && deviceName.equals("HC-05")){
837                 hc05Device = device;
838                 Log.d("Bluetooth", ">>> ;HC-05 encontrado <<<!");

```

```

839         break;
840     }
841 }
842 }
843 else{
844     Log.d("Bluetooth", "No se han encontrado dispositivos emparejados");
845     showToast("No se han encontrado dispositivos emparejados, por favor, vincule
↪ el HC-05 primero");
846     hc05Device = null;
847 }
848
849 if(hc05Device != null){
850     connectToDevice();
851 }
852 }
853
854 private void connectToDevice(){
855     if(hc05Device == null){
856         showToast("No se pudo encontrar el control entre los dispositivos vinculados"
↪ );
857         return;
858     }
859
860     if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.S &&
861 ActivityCompat.checkSelfPermission(this, Manifest.permission.BLUETOOTH_CONNECT
↪ ) != PackageManager.PERMISSION_GRANTED){
862         showToast("Se requiere el permiso de conexión Bluetooth");
863         return;
864     }
865
866     BluetoothSocket tmpSocket = null;
867     try{
868         tmpSocket = hc05Device.createRfcommSocketToServiceRecord(MY_UUID_SPP);
869     }
870     catch(IOException e){
871         Log.e("Bluetooth", "Método create() de Socket falló", e);
872         showToast("Error creando el Socket de conexión");
873         return;
874     }
875     btSocket = tmpSocket;
876     bluetoothAdapter.cancelDiscovery();
877
878     runOnUiThread(() -> {
879         buttonRetryConnection.setVisibility(View.GONE);
880         textViewConnectionStatus.setText("Estado: Conectando a " + hc05Device.getName
↪ () + "...");
881     });
882
883     try{
884         btSocket.connect();
885         Log.d("Bluetooth", "Conexión exitosa");
886         runOnUiThread(() -> {
887             showToast("Conectado a HC-05");
888             textViewConnectionStatus.setText("Estado: Conectado a " + hc05Device.
↪ getName());
889             buttonRetryConnection.setVisibility(View.GONE);

```

```

890     });
891
892     connectedThread = new ConnectedThread(btSocket);
893     connectedThread.start();
894     new Handler(Looper.getMainLooper()).postDelayed(new Runnable(){
895         @Override
896         public void run(){
897             Log.d("Bluetooth", "Solicitando configuración inicial...");
898             sendBluetoothCommand("GET\n");
899         }
900     }, 500);
901 }
902 catch(IOException connectException){
903     Log.e("Bluetooth", "Could not connect", connectException);
904     // --- CONEXIÓN FALLIDA ---
905     runOnUiThread(() -> {
906         showToast("Conexión Fallida: " + connectException.getMessage());
907         textViewConnectionStatus.setText("Estado: Conexión Fallida");
908         buttonRetryConnection.setVisibility(View.VISIBLE);
909     });
910     if(connectedThread != null){
911         connectedThread = null;
912     }
913     outputStream = null;
914     try{
915         btSocket.close();
916         btSocket = null;
917         outputStream = null;
918     }
919     catch(IOException closeException){
920         Log.e("Bluetooth", "No se pudo cerrar el socket del cliente",
921 ↪ closeException);
922     }
923     return;
924 }
925
926 private void sendBluetoothCommand(String command){
927     if (connectedThread != null) { // Check if thread exists
928         connectedThread.write(command.getBytes()); // Send through thread's write
929 ↪ method
930         Log.d("Bluetooth", "Sent: " + command.trim());
931     } else {
932         showToast("Not connected to HC-05.");
933     }
934 }
935
936 private void disconnectBluetooth(){
937     if(connectedThread != null){
938         connectedThread.cancel();
939         connectedThread = null;
940     }
941     hc05Device = null;
942     runOnUiThread(() -> {
943         showToast("Bluetooth Desconnected");
944         textViewConnectionStatus.setText("Estado: Desconectado");
945     });

```

```

944         buttonRetryConnection.setVisibility(View.VISIBLE); // Mostrar
945     });
946 }
947
948 private void clearLocalConfigAndUI() {
949     Log.d("ConfigUpdate", "Clearing local configuration and resetting UI.");
950     for (int i = 0; i < MAX_BUTTONS_APP; i++) {
951         appButtonConfigs[i].numSteps = 0;
952         Arrays.fill(appButtonConfigs[i].hexCodes, "FFFF");
953
954         updateButtonUI(i);
955     }
956     textViewSelectedButtonInfo.setText("Seleccione un botón");
957 }
958
959 private void displayButtonInfo(int buttonId){
960     if (buttonId >= 0 && buttonId < 16) {
961         AppButtonConfig config = appButtonConfigs[buttonId];
962         String buttonLabel = getButtonLabelFromId(buttonId);
963         String info;
964         String protocolName = (currentGlobalProtocolId >= 0 &&
↪ currentGlobalProtocolId < tvProtocolNames.length) ?
965             tvProtocolNames[currentGlobalProtocolId] : "Unknown";
966
967         if (config.numSteps > 0) {
968             info = "Btn ' " + buttonLabel + "' | Proto: " + protocolName;
969             for (int i = 0; i < config.numSteps; i++) {
970                 String cmdName = findNameByHexInMap(config.hexCodes[i]);
971                 info += String.format(java.util.Locale.US, "\nCmd %d: %s (%s)", (i+1)
↪ , cmdName, config.hexCodes[i]);
972             }
973         } else {
974             info = "Btn ' " + buttonLabel + "' - No Configurado | Proto: " +
↪ protocolName;
975         }
976         textViewSelectedButtonInfo.setText(info);
977     }
978 }
979
980 private String getButtonLabelFromId(int buttonId){
981     switch (buttonId) {
982         case 0: return "1"; case 1: return "2"; case 2: return "3"; case 3: return "A
↪ ";
983         case 4: return "4"; case 5: return "5"; case 6: return "6"; case 7: return "B
↪ ";
984         case 8: return "7"; case 9: return "8"; case 10: return "9"; case 11: return
↪ "C";
985         case 12: return "*"; case 13: return "0"; case 14: return "#"; case 15:
↪ return "D";
986         default: return "?";
987     }
988 }
989
990 private void updateButtonUI(int buttonId){
991     Button targetButton = null;
992

```

```

993         switch(buttonId){
994             case 0: targetButton = button1; break;
995             case 1: targetButton = button2; break;
996             case 2: targetButton = button3; break;
997             case 3: targetButton = buttonA; break;
998             case 4: targetButton = button4; break;
999             case 5: targetButton = button5; break;
1000            case 6: targetButton = button6; break;
1001            case 7: targetButton = buttonB; break;
1002            case 8: targetButton = button7; break;
1003            case 9: targetButton = button8; break;
1004            case 10: targetButton = button9; break;
1005            case 11: targetButton = buttonC; break;
1006            case 12: targetButton = buttonStar; break;
1007            case 13: targetButton = button0; break;
1008            case 14: targetButton = buttonHash; break;
1009            case 15: targetButton = buttonD; break;
1010        }
1011
1012        if (targetButton != null) {
1013            if (appButtonConfigs[buttonId].numSteps > 0) {
1014                targetButton.setBackgroundColor(getResources().getColor(R.color.
↪ button_configured_background, null));
1015            } else {
1016                targetButton.setBackgroundColor(getResources().getColor(R.color.
↪ button_default_background, null));
1017            }
1018        }
1019    }
1020
1021    private void updateAllButtonUIs(){
1022        for(int i = 0; i < MAX_BUTTONS_APP; i++){
1023            updateButtonUI(i);
1024        }
1025    }
1026
1027    private void loadIRCodesFromJson(){
1028        String jsonString;
1029        try{
1030            InputStream is = getAssets().open("ir_codes.json");
1031            int size = is.available();
1032            byte[] buffer = new byte[size];
1033            is.read(buffer);
1034            is.close();
1035            jsonString = new String(buffer, "UTF-8");
1036
1037            JSONObject root = new JSONObject(jsonString);
1038            JSONArray devices = root.getJSONArray("devices");
1039
1040            irDeviceList.clear();
1041            deviceBrandNames.clear();
1042
1043            for(int i = 0; i < devices.length(); i++){
1044                JSONObject deviceObj = devices.getJSONObject(i);
1045                IRDevice device = new IRDevice();
1046                device.brand = deviceObj.getString("brand");

```

```

1047         device.protocolId = deviceObj.getInt("protocolId");
1048         deviceBrandNames.add(device.brand);
1049
1050         JSONArray commands = deviceObj.getJSONArray("commands");
1051         for(int j = 0; j < commands.length(); j++){
1052             JSONObject cmdObj = commands.getJSONObject(j);
1053             device.commands.add(new IRCommand(
1054                 cmdObj.getString("name"),
1055                 cmdObj.getString("hex")
1056             ));
1057         }
1058         irDeviceList.add(device);
1059     }
1060     Log.d("JSONParser", "Cargados " + irDeviceList.size() + " dispositivos desde
↪ JSON");
1061     }
1062     catch(Exception e){
1063         Log.e("JSONParser", "Error cargando ir_codes.json", e);
1064         showToast("Error: No se pudo cargar la base de datos de códigos.");
1065     }
1066 }
1067
1068 public Map<String, String> getCurrentCommandMap() {
1069     return currentCommandMap;
1070 }
1071
1072 private String findNameByHexInMap(String hexCode) {
1073     if (hexCode == null || hexCode.equals("FFFF") || currentCommandMap == null) {
1074         return "Ninguno";
1075     }
1076
1077     for (Map.Entry<String, String> entry : currentCommandMap.entrySet()) {
1078         if (entry.getValue().equals(hexCode)) {
1079             return entry.getKey();
1080         }
1081     }
1082
1083     return "Desconocido (" + hexCode + ")";
1084 }
1085
1086 @Override
1087 protected void onDestroy(){
1088     super.onDestroy();
1089     disconnectBluetooth();
1090 }
1091
1092 private void showToast(String message){
1093     Toast.makeText(MainActivity.this, message, Toast.LENGTH_SHORT).show();
1094 }
1095
1096 private class ConnectedThread extends Thread{
1097     private final BluetoothSocket mmSocket;
1098     private final InputStream mmInStream;
1099     private final OutputStream mmOutStream;
1100
1101     public ConnectedThread(BluetoothSocket socket){

```

```

1102         mmSocket = socket;
1103         InputStream tmpIn = null;
1104         OutputStream tmpOut = null;
1105
1106         try{
1107             tmpIn = socket.getInputStream();
1108             tmpOut = socket.getOutputStream();
1109         }
1110         catch(IOException e){
1111             Log.e("BluetoothThread", "Error obteniendo streams", e);
1112             textViewConnectionStatus.setText("Estado: Conexión Perdida");
1113             Message errorMsg = uiHandler.obtainMessage(MESSAGE_CONNECTION_FAILED);
1114             uiHandler.sendMessage(errorMsg);
1115         }
1116
1117         mmInStream = tmpIn;
1118         mmOutStream = tmpOut;
1119         outputStream = mmOutStream;
1120     }
1121
1122     @Override
1123     public void run(){
1124         byte[] buffer = new byte[1024];
1125         int bytes;
1126         StringBuilder readMessageBuilder = new StringBuilder();
1127
1128         Log.d("BluetoothThread", "Comenzado loop de lectura...");
1129         while (true){
1130             try{
1131                 bytes = mmInStream.read(buffer);
1132                 if(bytes > 0){
1133                     String receivedChunk = new String(buffer, 0, bytes,
1134 ↪ StandardCharsets.UTF_8);
1135                     readMessageBuilder.append(receivedChunk);
1136                     int newLineIndex;
1137                     while((newLineIndex = readMessageBuilder.indexOf("\n")) != -1){
1138                         String completeMessage = readMessageBuilder.substring(0,
1139 ↪ newLineIndex + 1);
1140                         Message readMsg = uiHandler.obtainMessage(MESSAGE_READ);
1141                         readMsg.arg1 = completeMessage.getBytes().length;
1142                         readMsg.obj = completeMessage.getBytes();
1143                         readMsg.sendToTarget();
1144                         readMessageBuilder.delete(0, newLineIndex + 1);
1145                     }
1146                 }
1147                 else if(bytes == -1) {
1148                     Log.d("BluetoothThread", "InputStream llegó al end-of-stream");
1149                     Message lostMsg = uiHandler.obtainMessage(MESSAGE_CONNECTION_LOST
1150 ↪ );
1151                     uiHandler.sendMessage(lostMsg);
1152                     textViewConnectionStatus.setText("Estado: Conexión Perdida");
1153                     break;
1154                 }
1155             }
1156             catch(IOException e){
1157                 Log.e("BluetoothThread", "InputStream se desconectó");

```

```

1155         Message lostMsg = uiHandler.obtainMessage(MESSAGE_CONNECTION_LOST);
1156         uiHandler.sendMessage(lostMsg);
1157         textViewConnectionStatus.setText("Estado: Conexión Perdida");
1158         break;
1159     }
1160 }
1161 Log.d("BluetoothThread", "Loop de lectura finalizado");
1162 }
1163
1164 public void write(byte[] bytes){
1165     try{
1166         mmOutputStream.write(bytes);
1167     }
1168     catch(IOException e){
1169         Log.e("BluetoothThread", "Ocurrió un error al enviar el mensaje", e);
1170         Message writeErrorMsg = uiHandler.obtainMessage(MESSAGE_TOAST);
1171         Bundle bundle = new Bundle();
1172         bundle.putString("toast", "No se pudo enviar la data");
1173         writeErrorMsg.setData(bundle);
1174         uiHandler.sendMessage(writeErrorMsg);
1175     }
1176 }
1177
1178 public void cancel(){
1179     try{
1180         if(mmInStream != null) mmInStream.close();
1181         if(mmOutputStream != null) mmOutputStream.close();
1182         mmSocket.close();
1183         Log.d("BluetoothThread", "Socket cerrado");
1184     }
1185     catch(IOException e){
1186         Log.e("BluetoothThread", "No se pudo cerrar el Socket", e);
1187     }
1188 }
1189 }
1190
1191 private static class IRCommand{
1192     String name;
1193     String hex;
1194
1195     public IRCommand(String name, String hex){
1196         this.name = name;
1197         this.hex = hex;
1198     }
1199
1200     @NonNull
1201     @Override
1202     public String toString(){
1203         return name;
1204     }
1205 }
1206
1207 private static class IRDevice{
1208     String brand;
1209     int protocolId;
1210     List<IRCommand> commands = new ArrayList<>();

```

```
1211     }  
1212 }
```

Listing B.1: Actividad Principal (MainActivity.java)

Apéndice C

Código Fuente del Diálogo (ConfigureButtonDialogFragment.java)

Clase encargado de mostrar la ventana emergente para la configuración de macros.

```
1 package com.example.ir_remote_config;
2
3 import android.app.Dialog;
4 import android.content.Context;
5 import android.content.DialogInterface;
6 import android.os.Bundle;
7 import android.view.LayoutInflater;
8 import android.view.View;
9 import android.widget.AdapterView;
10 import android.widget.EditText;
11 import android.widget.Spinner;
12 import android.widget.TextView;
13 import androidx.annotation.NonNull;
14 import androidx.annotation.Nullable;
15 import androidx.appcompat.app.AlertDialog;
16 import androidx.fragment.app.DialogFragment;
17 import android.util.Log;
18 import android.widget.Toast;
19 import android.widget.AdapterView;
20 import java.util.ArrayList;
21
22
23 import android.app.Dialog;
24 import android.content.Context;
25 import android.content.DialogInterface;
26 import android.os.Bundle;
27 import android.util.Log;
28 import android.view.LayoutInflater;
29 import android.view.View;
30 import android.widget.AdapterView;
31 import android.widget.EditText;
32 import android.widget.Spinner;
33 import android.widget.TextView;
34 import android.widget.Toast;
35
```

```

36 import androidx.annotation.NonNull;
37 import androidx.annotation.Nullable;
38 import androidx.appcompat.app.AlertDialog;
39 import androidx.fragment.app.DialogFragment;
40
41 import java.util.ArrayList;
42 import java.util.Arrays;
43 import java.util.HashMap;
44 import java.util.Map;
45
46 public class ConfigureButtonDialogFragment extends DialogFragment {
47     public interface ConfigureButtonDialogListener {
48         void onDialogPositiveClick(int buttonId, String cmd1, String cmd2, String cmd3,
49             ↪ String cmd4);
50     }
51
52     ConfigureButtonDialogListener listener;
53     private int buttonId;
54     private String buttonLabel;
55     private static final int MAX_MACRO_STEPS_APP = 4;
56     private String[] currentCodesHex = new String[MAX_MACRO_STEPS_APP];
57     private ArrayList<String> commandNames; // Lista de nombres (ej. "Power")
58     private Map<String, String> commandMap; // Mapa <Nombre, Hex>
59
60     private Spinner spinnerCmd1, spinnerCmd2, spinnerCmd3, spinnerCmd4;
61     private final String CMD_NONE = "Ninguno (FFFF)"; // Constante para "no comando"
62
63     public static ConfigureButtonDialogFragment newInstance(int buttonId, String
64         ↪ buttonLabel,
65
66         String[] currentCodesHex,
67         ArrayList<String>
68         ↪ commandNames) {
69         ConfigureButtonDialogFragment fragment = new ConfigureButtonDialogFragment();
70         Bundle args = new Bundle();
71         args.putInt("BUTTON_ID", buttonId);
72         args.putString("BUTTON_LABEL", buttonLabel);
73         args.putStringArray("CURRENT_CODES", currentCodesHex);
74         args.putStringArrayList("COMMAND_NAMES", commandNames);
75         fragment.setArguments(args);
76         return fragment;
77     }
78
79     @Override
80     public void onAttach(@NonNull Context context){
81         super.onAttach(context);
82         try {
83             listener = (ConfigureButtonDialogListener) context;
84             // También necesitamos el Mapa de Comandos desde MainActivity
85             if (context instanceof MainActivity) {
86                 commandMap = ((MainActivity) context).getCurrentCommandMap();
87             } else {
88                 commandMap = new HashMap<>(); // Vacío si falla
89             }
90         } catch (ClassCastException e) {
91             throw new ClassCastException(getActivity().toString() + " must implement
92             ↪ ConfigureButtonDialogListener");

```

```

88     }
89 }
90
91 @NonNull
92 @Override
93 public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
94     AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
95     LayoutInflater inflater = requireActivity().getLayoutInflater();
96     View view = inflater.inflate(R.layout.dialog_configure_button, null);
97
98     if (getArguments() != null) {
99         buttonId = getArguments().getInt("BUTTON_ID", -1);
100         buttonLabel = getArguments().getString("BUTTON_LABEL", "?");
101         currentCodesHex = getArguments().getStringArray("CURRENT_CODES");
102         commandNames = getArguments().getStringArrayList("COMMAND_NAMES");
103     } else {
104         commandNames = new ArrayList<>();
105         Arrays.fill(currentCodesHex, "FFFF");
106     }
107
108     TextView title = view.findViewById(R.id.textViewDialogTitle);
109     spinnerCmd1 = view.findViewById(R.id.spinnerCommand1);
110     spinnerCmd2 = view.findViewById(R.id.spinnerCommand2);
111     spinnerCmd3 = view.findViewById(R.id.spinnerCommand3);
112     spinnerCmd4 = view.findViewById(R.id.spinnerCommand4);
113
114     title.setText("Configurar Botón " + buttonLabel + " (ID: " + buttonId + ")");
115
116     ArrayList<String> commandNamesWithNone = new ArrayList<>();
117     commandNamesWithNone.add(CMD_NONE);
118     commandNamesWithNone.addAll(commandNames);
119
120     ArrayAdapter<String> adapter1 = new ArrayAdapter<>(getContext(),
121         android.R.layout.simple_spinner_item, commandNames);
122     adapter1.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
123     spinnerCmd1.setAdapter(adapter1);
124
125     ArrayAdapter<String> adapter234 = new ArrayAdapter<>(getContext(),
126         android.R.layout.simple_spinner_item, commandNamesWithNone);
127     adapter234.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
128     ↪ ;
129     spinnerCmd2.setAdapter(adapter234);
130     spinnerCmd3.setAdapter(adapter234);
131     spinnerCmd4.setAdapter(adapter234);
132
133     spinnerCmd1.setSelection(Math.max(0, commandNames.indexOf(findNameByHex(
134     ↪ currentCodesHex[0]))));
135     spinnerCmd2.setSelection(Math.max(0, commandNamesWithNone.indexOf(findNameByHex(
136     ↪ currentCodesHex[1]))));
137     spinnerCmd3.setSelection(Math.max(0, commandNamesWithNone.indexOf(findNameByHex(
138     ↪ currentCodesHex[2]))));
139     spinnerCmd4.setSelection(Math.max(0, commandNamesWithNone.indexOf(findNameByHex(
140     ↪ currentCodesHex[3]))));
141
142     builder.setView(view)
143         .setPositiveButton("Save", (dialog, id) -> {

```

```

139         String cmd1Name = (String) spinnerCmd1.getSelectedItem();
140         String cmd2Name = (String) spinnerCmd2.getSelectedItem();
141         String cmd3Name = (String) spinnerCmd3.getSelectedItem();
142         String cmd4Name = (String) spinnerCmd4.getSelectedItem();
143
144         listener.onDialogPositiveClick(buttonId, cmd1Name, cmd2Name, cmd3Name
145 ↪ , cmd4Name);
146     })
147     .setNegativeButton("Cancel", null);
148     return builder.create();
149 }
150
151 private String findNameByHex(String hexCode) {
152     if (hexCode == null || hexCode.equals("FFFF") || commandMap == null) {
153         return CMD_NONE;
154     }
155     for (Map.Entry<String, String> entry : commandMap.entrySet()) {
156         if (entry.getValue().equals(hexCode)) {
157             return entry.getKey();
158         }
159     }
160     return CMD_NONE; // No encontrado
161 }
162 }

```

Listing C.1: Diálogo de Configuración

Apéndice D

Base de Datos de Códigos (ir_codes.json)

Estructura del archivo JSON alojado en la aplicación, que contiene los códigos hexadecimales para los diferentes modelos soportados.

```
1 {
2   "devices": [
3     {
4       "brand": "Samsung",
5       "protocolId": 0,
6       "commands": [
7         {
8           "name": "Power (E040)",
9           "hex": "E040 "
10        },
11        {
12          "name": "Vol+",
13          "hex": "E0E0 "
14        },
15        {
16          "name": "Vol-",
17          "hex": "E0D0 "
18        },
19        {
20          "name": "Mute",
21          "hex": "E0F0 "
22        },
23        {
24          "name": "Ch+",
25          "hex": "E048 "
26        },
27        {
28          "name": "Ch-",
29          "hex": "E008 "
30        },
31        {
32          "name": "Input/Source",
```

```

33     "hex": "E080 "
34 },
35 {
36     "name": "PREV-CHN",
37     "hex": "E0C8 "
38 },
39 {
40     "name": "Menu",
41     "hex": "E058 "
42 },
43 {
44     "name": "SMARTHUB",
45     "hex": "E09E "
46 },
47 {
48     "name": "TOOLS",
49     "hex": "E0D2 "
50 },
51 {
52     "name": "UP",
53     "hex": "E006 "
54 },
55 {
56     "name": "DOWN",
57     "hex": "E086 "
58 },
59 {
60     "name": "LEFT",
61     "hex": "E0A6 "
62 },
63 {
64     "name": "RIGHT",
65     "hex": "E046 "
66 },
67 {
68     "name": "MIDDLE",
69     "hex": "E016 "
70 },
71 {
72     "name": "RETURN",
73     "hex": "E01A "
74 },
75 {
76     "name": "EXIT",
77     "hex": "E0B4 "
78 },
79 {
80     "name": "TOOLS",
81     "hex": "E0D2 "

```

```

82     },
83     {
84         "name": "1",
85         "hex": "E020 "
86     },
87     {
88         "name": "2",
89         "hex": "E0A0 "
90     },
91     {
92         "name": "3",
93         "hex": "E060 "
94     },
95     {
96         "name": "4",
97         "hex": "E010 "
98     },
99     {
100        "name": "5",
101        "hex": "E090 "
102    },
103    {
104        "name": "6",
105        "hex": "E050 "
106    },
107    {
108        "name": "7",
109        "hex": "E030 "
110    },
111    {
112        "name": "8",
113        "hex": "E0B0 "
114    },
115    {
116        "name": "9",
117        "hex": "E070 "
118    },
119    {
120        "name": "0",
121        "hex": "E088 "
122    }
123 ]
124 },
125 {
126     "brand": "LG",
127     "protocolId": 1,
128     "commands": [
129         {
130             "name": "Power (2010)",

```

```

131         "hex": "2010"
132     },
133     {
134         "name": "Vol+",
135         "hex": "2040"
136     },
137     {
138         "name": "Vol-",
139         "hex": "20C0"
140     },
141     {
142         "name": "Mute",
143         "hex": "2090"
144     },
145     {
146         "name": "Ch+",
147         "hex": "2000"
148     },
149     {
150         "name": "Ch-",
151         "hex": "2080"
152     },
153     {
154         "name": "Input/Source",
155         "hex": "20D0"
156     },
157     {
158         "name": "1",
159         "hex": "2088"
160     },
161     {
162         "name": "2",
163         "hex": "2048"
164     },
165     {
166         "name": "3",
167         "hex": "20C8"
168     },
169     {
170         "name": "4",
171         "hex": "2028"
172     },
173     {
174         "name": "5",
175         "hex": "20A8"
176     },
177     {
178         "name": "6",
179         "hex": "2068"

```

```

180     },
181     {
182         "name": "7",
183         "hex": "20E8"
184     },
185     {
186         "name": "8",
187         "hex": "2018"
188     },
189     {
190         "name": "9",
191         "hex": "2098"
192     },
193     {
194         "name": "0",
195         "hex": "2008"
196     }
197 ]
198 },
199 {
200     "brand": "Philips",
201     "protocolId": 2,
202     "commands": [
203         {
204             "name": "Power",
205             "hex": "000C"
206         },
207         {
208             "name": "Vol+",
209             "hex": "0010"
210         },
211         {
212             "name": "Vol-",
213             "hex": "0011"
214         },
215         {
216             "name": "Mute",
217             "hex": "000D"
218         },
219         {
220             "name": "Ch+",
221             "hex": "0020"
222         },
223         {
224             "name": "Ch-",
225             "hex": "0021"
226         },
227         {
228             "name": "1",

```

```
229         "hex": "0001"
230     },
231     {
232         "name": "2",
233         "hex": "0002"
234     },
235     {
236         "name": "0",
237         "hex": "0000"
238     }
239 ]
240 }
241 ]
242 }
```

Listing D.1: Base de datos de códigos IR

Apéndice E

Esquema Eléctrico del Hardware

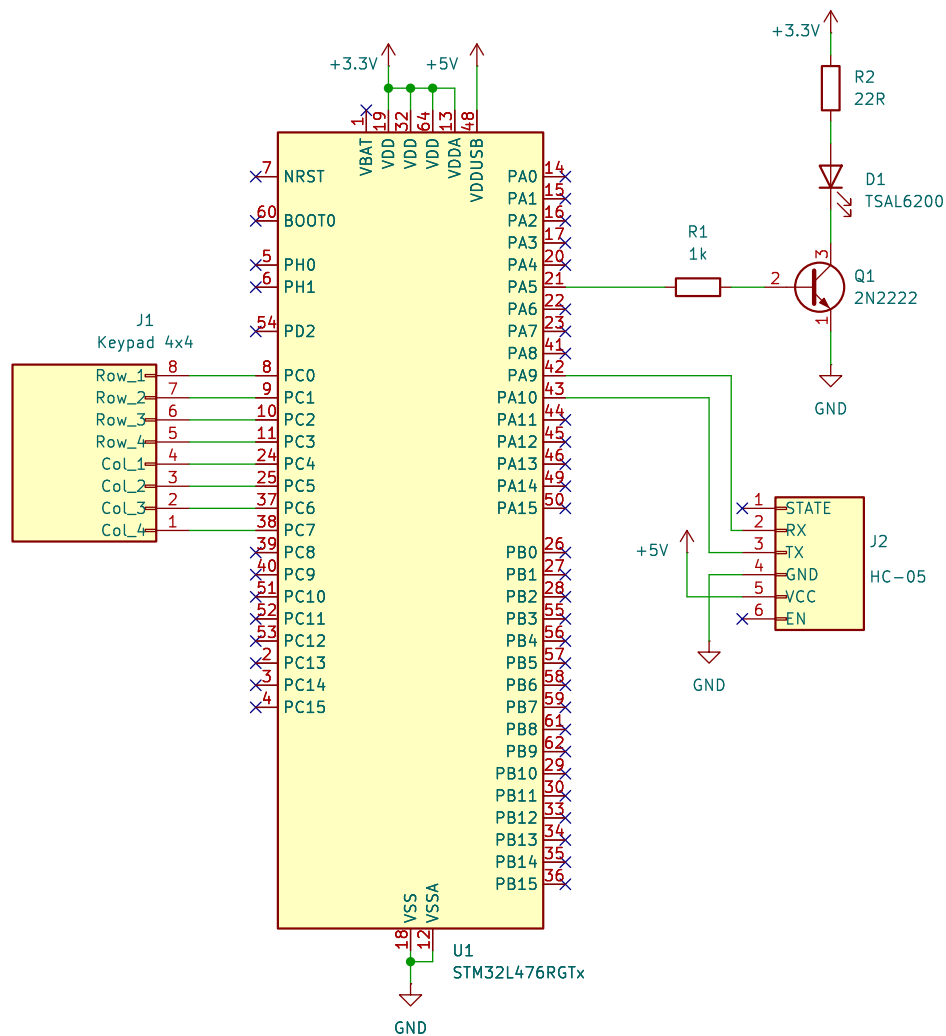


Figura E.1: Esquema Eléctrico del Hardware