



Universidad Zaragoza

GRADO EN INGENIERÍA INFORMÁTICA

Memoria Técnica: Beat Hero

Asignatura: Proyecto Hardware

Autores:

Alejandro Lacosta Ramos (NIP: 870897)

Pablo Villa Camañes (NIP: 874773)

Curso 2025-2026
9 de enero de 2026

Resumen ejecutivo

Esta memoria describe el diseño y la implementación de Beat Hero, un sistema embebido de tiempo real desarrollado bajo principios de modularidad y eficiencia energética. El objetivo central del trabajo ha consistido en construir una arquitectura de software robusta y portable, capaz de operar indistintamente sobre dos plataformas hardware distintas el simulador del microcontrolador LPC2105 y la placa de desarrollo física nRF52840 DK. El proyecto se centra en la resolución de problemas habituales en sistemas empotrados, como la gestión de recursos limitados, la concurrencia y la minimización del consumo de energía.

Los objetivos fundamentales del proyecto se centran en el desarrollo de un sistema embebido portable y modular, capaz de ejecutar la aplicación *Beat Hero* tanto en el simulador LPC2105 como en la placa nRF52840. El trabajo persigue la implementación eficiente de mecanismos de concurrencia y gestión de periféricos, priorizando la optimización del rendimiento y la minimización del consumo energético. Asimismo, se busca garantizar la robustez del sistema y demostrar el dominio de las herramientas de desarrollo y depuración.

La metodología empleada se fundamenta en una arquitectura software por capas, diseñada para desacoplar completamente la lógica de la aplicación de los detalles del hardware. El núcleo del sistema funciona bajo un modelo reactivo orientado a eventos, organizado por un planificador que gestiona la ejecución asíncrona a través de una cola FIFO circular. Para garantizar la robustez en un entorno concurrente, se han implementado mecanismos de exclusión mutua (*secciones críticas*) que aseguran la integridad de los datos compartidos entre las rutinas de interrupción y el hilo principal. La validación del proyecto ha seguido una estrategia incremental, evolucionando desde pruebas unitarias de periféricos hasta la integración final, y verificando el cumplimiento de los requisitos temporales y energéticos mediante herramientas de simulación y medición física.

Finalmente, los resultados validan el éxito de la implementación respecto a los requisitos no funcionales de eficiencia y rendimiento. El análisis de consumo energético confirma una reducción del **89 %** en la corriente promedio respecto a implementaciones bloqueantes, logrando un consumo en modo de juego de apenas **2 mA** y un estado de apagado (*System OFF*) con consumo virtualmente nulo ($0 \mu A$).

El power profiler revela que el procesador permanece en estados de bajo consumo durante el **86 %** del tiempo total de una partida, despertando únicamente para atender eventos puntuales. La arquitectura orientada a eventos ha demostrado una latencia de respuesta inferior a **100 μs** , con una sobrecarga del Kernel por debajo del 17 % del tiempo de CPU activa. Asimismo, la robustez del sistema queda garantizada por un ratio de procesamiento de la cola FIFO de **2:1** (velocidad de extracción doble que la de inserción), asegurando la estabilidad del sistema y la ausencia de desbordamientos incluso bajo condiciones de estrés.

En definitiva, el proyecto valida que la aplicación rigurosa de principios de ingeniería de software —como la separación de responsabilidades, la programación asíncrona y la optimización energética— permite construir sistemas empotrados robustos, eficientes y portables, demostrando un dominio efectivo de las herramientas y técnicas de desarrollo para arquitecturas ARM.

Índice

1. Introducción	4
2. Objetivos	4
3. Metodología	5
3.1. Herramientas utilizadas	5
3.2. Estructura del proyecto	6
3.2.1. El Runtime y la Gestión de Eventos	8
3.3. Tests	9
4. Resultados	9
4.1. Tests y Anexo A.1	9
4.1.1. Evolución del Sistema y Gestión de Energía (Blinks)	9
4.1.2. Bit Counter Strike	10
4.2. Juego final: Beat Hero	11
4.2.1. Implementación del juego y cumplimiento de requisitos	12
4.2.2. Descripción del flujo de la partida	13
4.3. Mediciones energéticas del juego final	13
4.3.1. Tabla de medidas de consumo	14
4.3.2. Análisis Dinámico: Beat Hero	14
4.4. Análisis de Rendimiento	15
4.4.1. Escenario de Prueba	15
4.4.2. Análisis Global de Eficiencia	15
4.4.3. Análisis Temporal de Funciones Críticas	15
5. Conclusiones	19
A. Anexos	21
A.1. Diagramas de Arquitectura	21
A.2. (Mediciones Energéticas)	23
A.3. Interfaz	25
A.4. Código Fuente del Proyecto	26
A.4.1. Implementación del Juego y Estadísticas	26
A.4.2. Gestor de Eventos (Runtime)	38
A.4.3. Cola FIFO de Eventos	40
A.4.4. Drivers y Servicios	42
A.4.5. Capa de Abstracción de Hardware (HAL) y Concurrencia	45

Índice de figuras

1.	Arquitectura por capas del sistema.	6
2.	Diagrama de estados para la gestión de botones.	7
3.	Diagrama de estados del Bit Counter Strike.	11
4.	Diagrama de estados del juego Beat Hero.	12
5.	Captura del Power Profiler durante la ejecución de Beat Hero.	13
6.	Métricas arrojadas por el <i>Performance Analyzer</i>	15
7.	Detalle de llamadas a la función de "esperar" (IDLE). El tiempo activo es mínimo porque el reloj se detiene al entrar en reposo.	16
8.	Impacto de la transmisión UART. Se evidencia el coste del envío síncrono. . .	16
9.	Detalles de las funciones de la lógica de negocio. El consumo de CPU es despreciable.	16
10.	Detalles de los tiempos de las funciones del <i>Gestor de Eventos</i>	17
11.	Detalles de los tiempos de las funciones de la cola FIFO	17
12.	Latencia de procesamiento de entrada. El tiempo de evaluación es despreciable. .	18
13.	Detalles de los tiempos del servicio de alarmas. Se observa una correlación directa entre el número de llamadas y los ticks del sistema.	18
14.	Diagrama de secuencia detallado: Interacción ISR-FIFO-Dispatcher.	21
15.	Diagrama de bloques de arquitectura de la cola Fifo	22
16.	Consumo del Blink v2 (Espera Activa).	23
17.	Consumo del Blink v3.	23
18.	Consumo del Blink v3 bis.	24
19.	Tabla comparativa consumos.	24
20.	Interfaz para conocer los botones y leds de la placa nrf52840.	25

1. Introducción

Esta memoria recoge el trabajo desarrollado en la asignatura Proyecto Hardware del quinto semestre del Grado en Ingeniería Informática de la Universidad de Zaragoza, durante el curso 2025-2026. A lo largo de la asignatura se aborda el diseño de sistemas empotrados mediante una metodología basada en capas, cuyo propósito es separar la lógica del hardware para obtener sistemas modulares, portables y robustos. Bajo este enfoque, el proyecto final consiste en implementar el juego *Beat Hero*, una aplicación completa que integra todos los módulos construidos en prácticas previas y que debe funcionar tanto en el simulador LPC2105 como en la placa NRF52840 DK.

El desarrollo de *Beat Hero* permite poner en práctica los objetivos de la asignatura, gestionar periféricos mediante interfaces abstractas, evitar condiciones de carrera mediante el uso correcto de interrupciones y secciones críticas, diseñar máquinas de estados eficientes, optimizar el consumo energético, y validar el comportamiento global del sistema bajo diferentes configuraciones. Además, el proyecto exige la integración de componentes adicionales como un generador de números aleatorios, un planificador basado en alarmas y mecanismos de depuración, lo que completa la construcción de un sistema embebido funcional y autónomo.

El documento está organizado para presentar de forma ordenada el trabajo realizado. En primer lugar, se describe el contexto y los objetivos del proyecto. A continuación, se expone la metodología seguida para el diseño por capas y la integración de los distintos módulos funcionales. Después, se detallan los resultados obtenidos, incluyendo el funcionamiento final del sistema, las pruebas realizadas y las métricas de rendimiento. Finalmente, se presentan las conclusiones del proyecto y una reflexión sobre los problemas encontrados y sus soluciones, completando así la documentación del desarrollo de *Beat Hero* sobre ambas plataformas.

2. Objetivos

- Desarrollar un sistema embebido completo capaz de ejecutar el juego *Beat Hero* en dos plataformas distintas (simulador LPC2105 y placa NRF52840 DK), garantizando la portabilidad del código y la abstracción del hardware.
- Diseñar una arquitectura modular basada en capas que permita separar la lógica del juego de los detalles específicos de cada dispositivo, facilitando la reutilización y el mantenimiento del software.
- Implementar periféricos y mecanismos de concurrencia propios de sistemas empotrados, trabajando sin sistema operativo y gestionando interrupciones, alarmas, temporizadores y posibles condiciones de carrera.
- Optimizar el rendimiento del sistema mediante la medición y el análisis del tiempo de ejecución, incluyendo la identificación de cuellos de botella y la toma de decisiones fundamentadas en cuanto a eficiencia del código.
- Minimizar el consumo energético del sistema embebido evaluando el impacto de las decisiones de diseño y configuraciones hardware/software, en línea con los objetivos de sostenibilidad digital y eficiencia energética de la asignatura.
- Cumplir los requisitos y restricciones propias del entorno embebido, tales como la escasez de recursos, la limitación de memoria, la necesidad de respuesta determinista y la ausencia de un sistema operativo.

- Asegurar la robustez del juego garantizando un funcionamiento estable, evitando fallos por mal uso de interrupciones, gestionando correctamente estados internos y asegurando un comportamiento predecible en ambas plataformas.
- Demostrar el dominio de las herramientas de desarrollo cruzado con Keil μ Vision para procesadores ARM, realizando tareas de depuración eficiente, inspección del estado del sistema y validación del comportamiento final.
- Aplicar los conocimientos teóricos adquiridos en asignaturas previas para crear un diseño funcional y defendible, siendo capaces de justificar las decisiones tomadas en la implementación del proyecto.
- Documentar adecuadamente el trabajo realizado mediante la elaboración de una memoria técnica completa, clara y coherente, siguiendo los criterios establecidos en la asignatura Proyecto Hardware del Grado en Ingeniería Informática.

3. Metodología

Para el desarrollo del proyecto se ha seguido una metodología de trabajo basada en capas y modularidad, en la que cada módulo tiene una responsabilidad específica según el principio de *Single Responsibility*. Esto ha permitido aislar la funcionalidad de cada componente del sistema embebido, facilitando su comprensión, prueba y mantenimiento. Cada módulo realiza únicamente las operaciones que le corresponden, asegurando la coherencia y la robustez del sistema.

Para el proyecto se ha aplicado una programación orientada a eventos, en la que los eventos generados por interrupciones o entradas externas modifican el flujo de ejecución del programa. Este enfoque permite un control eficiente de los periféricos y garantiza un comportamiento determinista en tiempo real. Además, se ha implementado una metodología incremental, donde cada módulo se probaba de manera independiente y progresiva a través de ejemplos sencillos (blinks V1 a V4, bit counter strike) antes de integrarlo en el juego final *Beat Hero*. Esta aproximación asegura que cada componente funcione correctamente antes de pasar a etapas más complejas del proyecto.

Se han realizado tests exhaustivos para cada módulo, verificando tanto la funcionalidad como la robustez del sistema. Estos tests permiten detectar fallos, cubrir la mayor parte posible del código y asegurar que el juego final funcione correctamente en ambas plataformas: simulador LPC2105 y placa nRF52840 DK. Los resultados de estas pruebas y los fragmentos de código más relevantes se encuentran documentados en los Anexos, y opcionalmente el proyecto completo puede consultarse en el repositorio disponible.

3.1. Herramientas utilizadas

Para el desarrollo y prueba del proyecto se han empleado diversas herramientas que han permitido emular, depurar y medir el comportamiento del sistema en distintas plataformas:

- **Keil μ Vision5:** Ha sido utilizado como entorno de desarrollo integrado (IDE) para programar y depurar el código. Además, permite emular el procesador LPC2105, lo que fue fundamental para trabajar sin disponer de la placa física. Gracias a Keil se pudieron simular periféricos como GPIO, LEDs, y visualizar eventos y logs mediante la consola del UART.

- **Placa de desarrollo nRF52840 DK:** Suministrada en el laboratorio, permitió ejecutar el proyecto sobre hardware real, garantizando la validación del juego *Beat Hero* y su comportamiento bajo condiciones reales de ejecución.
- **JLink:** Depurador utilizado para conectarse a la placa nRF52840, permitiendo la depuración paso a paso y la inspección de registros y memoria durante la ejecución.
- **nRF Connect for Desktop:** Una aplicación con herramientas de Nordic Semiconductor utilizada para monitorizar la placa y medir consumo energético. Esta app incluye el programa *nRF Power Profiler*, que permitió evaluar el consumo energético del sistema en distintos estados de ejecución, y un terminal que se usó para recibir logs del sistema en tiempo real.
- **Documentación técnica oficial:** Se empleó documentación técnica oficial tanto de NXP como de Nordic para comprender en profundidad la arquitectura y los periféricos de cada plataforma. Entre los documentos consultados están los manuales de LPC2105, la guía genérica del ARM Cortex-M4, el nRF52840 DK User Guide y la especificación completa del nRF52840 (Product Specification v1.10). Estos recursos fueron muy útiles para el diseño correcto de los HAL y la configuración de periféricos.

Gracias a esta metodología de trabajo, incremental, modular y orientada a eventos, se ha logrado implementar un juego completo, funcional y robusto, cumpliendo los objetivos de portabilidad, eficiencia y control de consumo energético planteados en la asignatura.

3.2. Estructura del proyecto

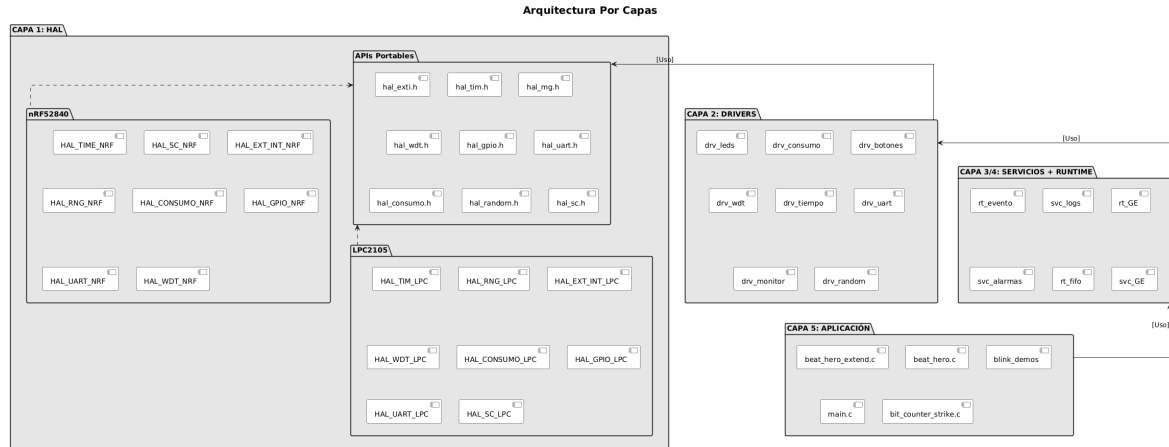


Figura 1: Arquitectura por capas del sistema.

El proyecto se organiza en distintos niveles de abstracción, comenzando por los HAL (*Hardware Abstraction Layer*), que son los módulos encargados de enlazar el software con el hardware de forma transparente para el resto del sistema. Su función es ofrecer una interfaz común que permita acceder a elementos como LEDs, botones, temporizadores, watchdog, reloj del sistema o mecanismos de bajo consumo sin que los niveles superiores tengan que conocer cómo se implementa cada periférico en la placa real o en el emulador.

En este proyecto se desarrollaron todos los HAL requeridos, implementados para las dos plataformas objetivo: LPC2105 en el entorno de emulación de Keil, y nRF52840 en la placa real de Nordic. En ambos casos se incluyeron las funcionalidades necesarias para mostrar

logs, adaptadas a las capacidades de cada entorno. Además, en el caso de la nRF52840 se implementaron opcionalmente dos temporizadores adicionales: el SysTick y el RTC, con el objetivo de mejorar la precisión temporal y optimizar aún más el consumo energético en los modos de bajo consumo. Estas implementaciones adicionales permitieron experimentar con alternativas más eficientes sin modificar el resto de capas.

Sobre los HAL, se construyen los **drivers**, que abstraen esa interacción y ofrecen funcionalidades limpias, coherentes y reutilizables sin importar qué hardware haya debajo. Es decir, de cambiar la placa, solo ha de reescribirse el HAL, mientras que los drivers y todo el resto del proyecto seguirán funcionando igual.

Los primeros drivers desarrollados fueron el driver de LEDs y el driver de tiempo, porque constituyen la base de cualquier interacción visible y de toda la temporización interna del juego. Una vez creados y testeados, se incorporaron el driver de consumo y el driver de monitores, que permiten entrar automáticamente en modos de bajo consumo, medir el rendimiento y optimizar el comportamiento energético del sistema.

Después se añadió el driver de botones, responsable de gestionar interrupciones externas y de permitir que el procesador despierte desde un estado de reposo. Este driver es especialmente relevante porque su funcionamiento se modeló mediante un diagrama de estados de Mealy, que describe con precisión las transiciones y acciones asociadas a cada estado del botón, garantizando detección robusta y fiable de las pulsaciones.

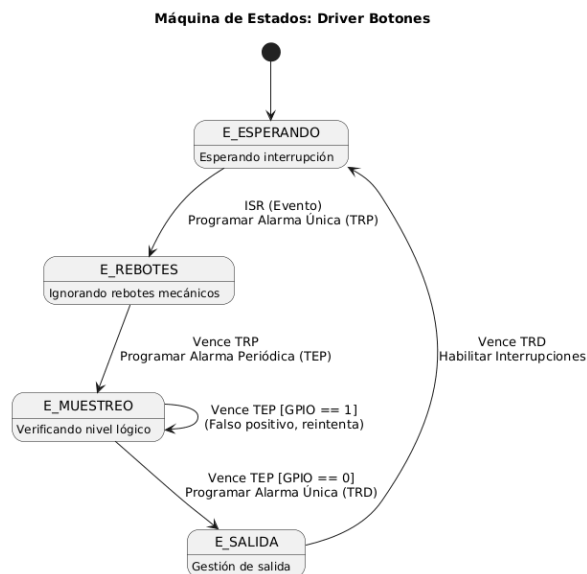


Figura 2: Diagrama de estados para la gestión de botones.

Por último, el driver del watchdog aporta tolerancia a fallos, asegurando que el sistema pueda reiniciarse de manera segura ante condiciones inesperadas.

Además de los drivers, se han desarrollado **services**, que son módulos de software independientes del hardware y diseñados para ser reutilizables a lo largo del proyecto. Mientras que los drivers se encargan de gestionar directamente los periféricos o recursos específicos del hardware (como LEDs, botones, temporizadores o el watchdog), los services proporcionan funcionalidades de más alto nivel que utilizan los drivers para implementar lógica común o coordinar distintos módulos.

Entre los services implementados destacan:

- El servicio de logs, que permite registrar eventos y métricas de ejecución para depuración y análisis.
- El servicio de alarmas, encargado de generar y gestionar eventos únicos o periódicos, facilitando la planificación de acciones en el sistema.
- El servicio del gestor de eventos, que centraliza la suscripción y cancelación de eventos, permitiendo la programación orientada a eventos dentro de la arquitectura modular del sistema.

La principal diferencia con los drivers es que los services no interactúan directamente con el hardware, sino que se apoyan en los drivers para realizar tareas complejas de manera portable y coordinada.

3.2.1. El Runtime y la Gestión de Eventos

El Runtime del proyecto, compuesto por el Gestor de Eventos (`rt_GE`) y la cola FIFO (`rt_FIFO`), constituye el corazón del sistema embebido. Su función principal es coordinar toda la actividad del juego mediante un patrón productor-consumidor, encolando y despachando eventos de manera eficiente y determinista.

El sistema abstrae las interacciones hardware y software mediante el tipo enumerado `EVENTO_T`, que unifica fuentes heterogéneas como la pulsación de un usuario (`ev_PULSAR_BOTON`) o el vencimiento de un temporizador (`ev_T_PERIODICO`). Cada evento se almacena en la cola acompañado de un *payload* genérico de 32 bits que parametriza el evento (`uint32_t aux_data`), y una marca de tiempo que registra el instante exacto en el que se produce la interrupción física (*timestamp*). Este diseño permite que el despachador procese la lógica sin conocer los detalles de bajo nivel del periférico que generó la señal.

El gestor se encarga de activar alarmas, alimentar el watchdog para tolerancia a fallos y poner el procesador en modo de bajo consumo cuando no hay eventos pendientes. El flujo detallado de interacción entre las interrupciones hardware y el despacho de eventos se encuentra documentado en el diagrama de secuencia del Anexo A.1 (Fig. 14).

La cola FIFO almacena eventos con sus datos y marcas temporales, permitiendo procesarlos en estricto orden de llegada. Esta estructura asegura que cada módulo (drivers y services) reciba los eventos correspondientes sin depender del hardware específico. De esta forma, el runtime garantiza la modularidad, la portabilidad y la robustez del sistema, permitiendo que todas las funciones del juego y del sistema se ejecuten de manera coordinada y eficiente. Se puede observar el funcionamiento de la cola fifo en Anexo A.1 (Fig. 15).

Gestión de Concurrencia y Secciones Críticas: Un desafío fundamental en el diseño de la `rt_FIFO` es la gestión de la concurrencia. Dado que la cola es un recurso compartido accedido tanto desde el contexto de interrupción (productores, ej. pulsación de botón) como desde el bucle principal (consumidor), existe riesgo de condiciones de carrera si una interrupción interrumpe la actualización de los punteros de lectura/escritura.

Para mitigar este riesgo y asegurar la integridad de los datos, se han implementado secciones críticas (`hal_SC_entrar` / `hal_SC_salir`). Estas funciones deshabilitan temporalmente las interrupciones durante las operaciones de encolado y desencolado, garantizando que la modi-

ficación de los índices y el contador de elementos sea una operación atómica. Esta protección es imprescindible para evitar la corrupción de la memoria y asegurar la robustez del sistema ante ráfagas de eventos asíncronos.

3.3. Tests

Todos los módulos, tanto drivers como services, cuentan con tests específicos que verifican su correcto funcionamiento en ambas plataformas: el emulador LPC2105 y la placa nRF52840. Se prueban las funciones de cada driver y service de forma aislada, ya que contienen la lógica principal del juego y del sistema. Los HAL, al ser dependientes del hardware, también se comprueban, pero su correcta implementación no es suficiente para garantizar el funcionamiento completo: sin drivers y services bien probados, el sistema no funciona.

Los driver de LEDs, tiempo, consumo, botones o watchdog se han testado en escenarios que reproducen las condiciones reales de ejecución del juego. De igual modo, los servicios de logs, alarmas y gestión de eventos han sido verificados para asegurar que los eventos se encolan, despachan y procesan correctamente. Incluso el runtime y la FIFO cuentan con tests que confirman el orden, la integridad de los datos y la gestión de overflow, asegurando así la robustez de toda la arquitectura del proyecto.

4. Resultados

Durante el desarrollo del proyecto se han ido obteniendo resultados parciales e incrementales, derivados tanto de la arquitectura modular implementada como del enfoque de trabajo basado en pruebas y programas intermedios. A medida que se avanzaba en la asignatura, cada nuevo módulo, driver o service se validaba mediante test específicos, se probaba en pequeños programas funcionales y finalmente se integraba en aplicaciones más complejas, como los juegos *Bit Counter Strike* y *Beat Hero*. Los resultados que se presentan a continuación recogen este proceso progresivo, desde la verificación de los componentes del sistema hasta el funcionamiento completo del juego final y su perfil energético medido sobre la nRF52840.

4.1. Tests y Anexo A.1

Un primer resultado clave es la creación de un conjunto completo de test, uno por cada módulo del sistema. Estos test permiten verificar que cada HAL, driver y service se comporta exactamente como se espera antes de integrarlo en la aplicación final. La retroalimentación de cada prueba se muestra mediante LEDs y logs, encendiendo patrones específicos para indicar éxito o fallo. Para facilitar su ejecución se generaron targets independientes dentro del proyecto, lo que permitió probar cada módulo de forma aislada. Gracias a esto, la integración progresiva del sistema se desarrolló sin errores acumulados y con trazabilidad completa.

4.1.1. Evolución del Sistema y Gestión de Energía (Blinks)

Los programas Blinks son pruebas que validan la evolución de la arquitectura del software, desde un modelo bloqueante hasta un sistema orientado a eventos. Con cada blink se ha conseguido probar los diferentes módulos de la arquitectura verificando su correcto funcionamiento. Además, contamos con el programa bit counter que funciona como un test de integración, ya que por primera vez se hace una versión mínima del juego final y se prueba el sistema entero en funcionamiento.

- **Blink v1 (Espera Activa CPU):** Implementa el parpadeo mediante bucles de retardo vacíos que consumen ciclos de CPU. Su valor reside en establecer la línea base de peor eficiencia energética, demostrando el alto coste de mantener el procesador ocupado al 100 % sin realizar trabajo útil.
- **Blink v2 (Driver de Tiempo Bloqueante):** Valida el funcionamiento correcto del `drv_tiempo`. Aunque mejora la precisión temporal respecto a la v1, mantiene la CPU en modo activo durante las esperas, confirmando la necesidad de mecanismos no bloqueantes para el ahorro de energía.
 - **Medidas:** Se registra un consumo medio de **5.64 mA** (ver Tabla comparativa 4.3.1), cuyo valor ha sido medido con el (*Power Profiler*) Anexo A.2 (Fig. 16). Este valor confirma la ineficiencia de los mecanismos de espera activa, ya que la totalidad de la energía se disipa ejecutando instrucciones de bucle sin aportar funcionalidad al sistema.
- **Blink v3 (Interrupciones y System Idle):** Introduce el uso de temporizadores hardware con callbacks asíncronos. Permite validar que el sistema puede liberar la CPU entre eventos para entrar en modo Idle, logrando la primera reducción significativa de consumo al detener el reloj de la CPU cuando no es necesaria.
 - **Medidas:** Según se observa en la Tabla 4.3.1, el consumo medio desciende a **633 μA** , logrando una mejora del **89 %**. El uso de la instrucción `__WFI` permite detener el reloj del núcleo en los periodos de inactividad, manteniendo solo los temporizadores activos. Consultar la medición del consumo con el (*Power Profiler*) en Anexo A.2 (Fig. 17)
- **Blink v4 (Arquitectura Orientada a Eventos):** Prueba fundamental que valida la cola de eventos `rt_FIFO` y el despachador de eventos, demostrando que el sistema puede gestionar tareas de forma totalmente asíncrona y no bloqueante, base necesaria para el juego final.
- **Blink v3 bis (Deep Sleep):** Variante del Blink v3 para validar la gestión de energía. Tras una serie de ciclos, fuerza al sistema a entrar en modo Deep Sleep. Verifica la capacidad del microcontrolador para dormir casi totalmente y su correcta reactivación, simulando el comportamiento de apagado automático del producto final.
 - **Medidas:** La Tabla 4.3.1 confirma un consumo en reposo virtualmente nulo ($\approx 0 \mu\text{A}$). Con una carga de solo **8.52 nC** en 10 segundos, se valida el diseño para dispositivos a batería, ya que el sistema no consume energía mientras está apagado. Consultar la medición del consumo con el (*Power Profiler*) en Anexo A.2 (Fig. 18)

4.1.2. Bit Counter Strike

El módulo *Bit Counter Strike* se desarrolló como un paso intermedio de integración antes de abordar el juego final. Es una versión simplificada de un juego de memoria ("Simón Dice") que utiliza una secuencia fija de LEDs.

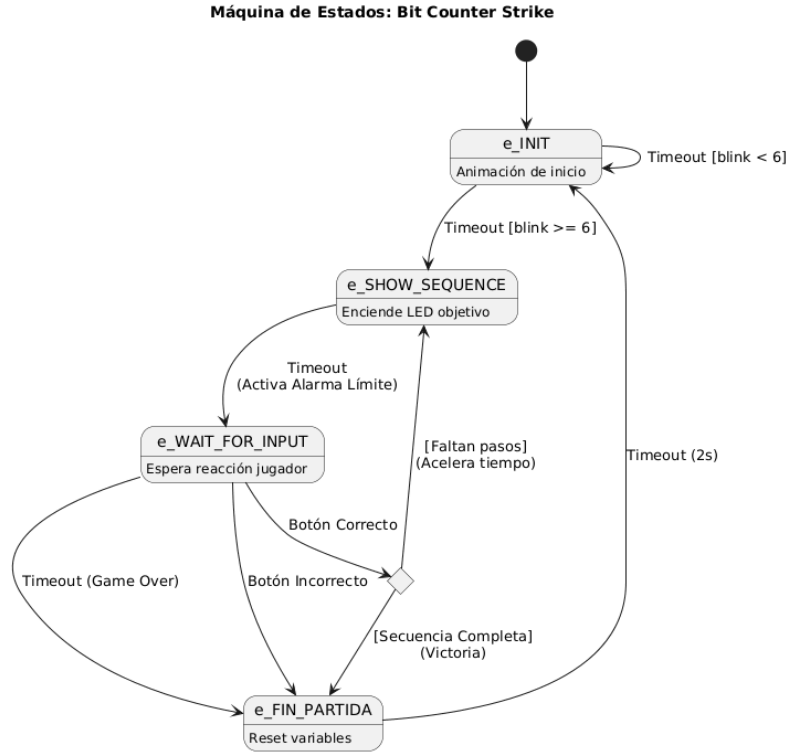


Figura 3: Diagrama de estados del Bit Counter Strike.

Este módulo aporta un valor crítico al proyecto al servir como prueba de la arquitectura diseñada, ya que valida la coordinación correcta entre la Máquina de Estados, el servicio de alarmas para la gestión de tiempos y ventanas de respuesta, y la lectura de entradas de usuario. Permite asegurar la lógica funcional sin bloqueos antes de añadir la complejidad de las partituras aleatorias y el módulo de estadísticas.

4.2. Juego final: Beat Hero

El proyecto ha alcanzado su objetivo principal con el desarrollo del juego *Beat Hero*, cumpliendo con las especificaciones funcionales y los requisitos energéticos establecidos, tanto en el emulador LPC2105 como en la plataforma física nRF52840. La implementación no se ha limitado solo en obtener el juego operativo, sino que se ha validado su robustez y corrección mediante un conjunto de tests unitarios, programas de prueba específicos y mediciones reales de consumo, todo ello soportado por una arquitectura modular basada en eventos.

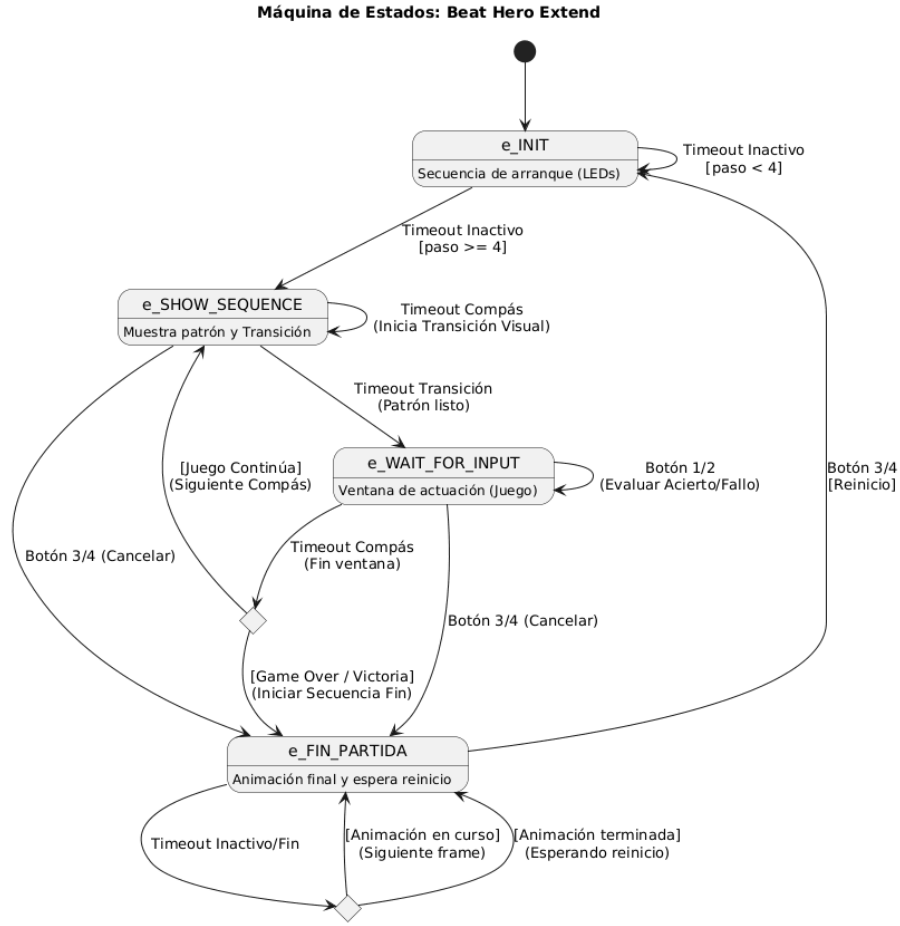


Figura 4: Diagrama de estados del juego Beat Hero.

4.2.1. Implementación del juego y cumplimiento de requisitos

La lógica del juego ha sido diseñada para dar respuesta a los Requisitos Funcionales (RF) y No Funcionales (RNF) del enunciado mediante una arquitectura de software robusta.

Para garantizar un funcionamiento ordenado (RF1), el sistema se estructura sobre una Máquina de Estados Finitos (FSM) que define claramente el ciclo de vida de la aplicación. Se han implementado estados exclusivos para el arranque (**e_INIT**) y el fin (**e_FIN_PARTIDA**), los cuales ejecutan secuencias visuales de luces obligatorias.

Esto asegura que el usuario siempre reciba feedback visual del estado del sistema y evita que el juego inicie o termine de manera desordenada.

Desde el punto de vista de la arquitectura, el sistema es reactivo (RF2). Se ha eliminado cualquier espera activa o bucles de bloqueo; el control del flujo y la temporización dependen exclusivamente de la suscripción al evento **ev_PULSAR_BOTON** y del uso del servicio de alarmas para gestionar los timeouts. Esta gestión eficiente se complementa con el mecanismo de seguridad del Watchdog (RF3). Este reinicio solo se dispara ante un bloqueo real del software donde el planificador no procesa ningún evento durante 1 segundo. La inactividad del jugador no provoca el reset, puesto que las alarmas internas del juego siguen generando eventos periódicos que mantienen al gestor funcionando y al Watchdog alimentado, diferenciando así entre un usuario pasivo y un error de sistema.

Para estructurar el despliegue del proyecto, se han definido los targets NRF Blink, NRF Bit-Counter, NRF Test y NRF Production, permitiendo aislar los módulos de prueba de la versión definitiva. Esta organización se refuerza con configuraciones de compilación diferenciadas: un modo Debug, que habilita la monitorización de estadísticas vía UART (cumpliendo el RF5), y un modo Producción, que elimina la instrumentación de depuración para maximizar la eficiencia energética en la entrega final.

4.2.2. Descripción del flujo de la partida

El desarrollo de una partida sigue un flujo secuencial. Al iniciar el sistema, la máquina de estados se sitúa en el estado de arranque, donde ejecuta una secuencia de inicialización de LEDs. Esta animación confirma que el procesador ha despertado correctamente y resetea las variables de juego.

Una vez en el bucle principal, el sistema entra en el estado de presentación `e_SHOW_SEQUENCE`, donde genera aleatoriamente el patrón del compás actual y lo muestra en los LEDs tras una breve transición visual. Inmediatamente después, se transita al estado de espera `e_WAIT_FOR_INPUT`, abriendo la ventana de actuación para el jugador. En esta fase se evalúa la precisión del usuario; el sistema discrimina entre pulsaciones Perfectas y Aciertos normales, asignando la puntuación correspondiente, o penalizando los fallos y omisiones. Simultáneamente, se gestiona la dificultad dinámica, incrementando la variable de nivel cada cuatro compases completados.

Finalmente, el flujo termina en el estado `e_FIN_PARTIDA`, ya sea por completar los 30 compases o por perder debido a una puntuación baja o por cancelación del usuario. En este punto se ejecuta una animación visual de fin y, al terminar, el sistema invoca la función `drv_consumo_dormir` para entrar en modo Sleep profundo. El microcontrolador permanecerá en este estado de mínimo consumo hasta que una interrupción hardware en los botones 3 o 4 Anexo C (Fig. 20) solicite una nueva partida. Como medida de seguridad adicional, se ha implementado un temporizador de inactividad que fuerza esta suspensión tras 10 segundos sin interacción.

4.3. Mediciones energéticas del juego final



Figura 5: Captura del Power Profiler durante la ejecución de Beat Hero.

El análisis de consumo muestra los picos correspondientes a la activación de LEDs y procesamiento de eventos, los tramos estables de espera (Idle) y la caída final al estado de suspensión (Deep Sleep).

4.3.1. Tabla de medidas de consumo

A continuación se presenta el resumen de los valores obtenidos en las distintas etapas de desarrollo:

Versión	Estado CPU	Consumo	Mejora vs v2	Carga (10s)	Análisis Técnico
Blink v2	Espera activa.	5.64 mA	Ref.	672 μ A	Línea base (Peor Caso). La CPU ejecuta instrucciones inútiles continuamente. No explota modos de bajo consumo.
Blink v3	Sleep (WFI).	633 μ A	89 %	30.2 μ A	El Timer genera interrupciones periódicas. Entre ellas, la CPU duerme, reduciendo el consumo un orden de magnitud respecto a la espera activa.
Blink v3 bis	System OFF.	$\approx 0 \mu$ A (Sleep) 0.85 mA (Activo)	85 %	8.52 nC	Optimiza la inactividad total. Entra en apagado profundo, consumiendo mínima energía. Solo despierta por interrupción externa (botón).
Beat Hero	Juego Activo.	0.9-2.1 mA	84 %	Var.	Actividad intermitente: Coexisten LEDs, timers, botones y watchdog. Gracias a la arquitectura modular, el consumo se mantiene bajo (1mA) pese a la carga de trabajo.

Tabla 1: Comparativa de rendimiento y consumo energético por versión

4.3.2. Análisis Dinámico: Beat Hero

Más allá de los valores medios recogidos en la tabla, el comportamiento energético del juego final es altamente dinámico. Como se observa en la captura del *Power Profiler* adjunta en el Anexo B (Fig. A.2), el perfil de consumo se descompone en tres componentes claros:

1. **Corriente Base (≈ 0.9 mA):** Corresponde al "suelo" de consumo cuando el procesador está en modo *Sleep* (WFI) esperando la siguiente interrupción del temporizador o los botones. Valida que la arquitectura reactiva funciona correctamente incluso dentro de la partida.
2. **Sobrecarga de Procesamiento (≈ 2.1 mA):** Pequeños incrementos sostenidos que ocurren cuando el Gestor de Eventos (`rt_GE`) despierta para procesar lógica de juego, generar números aleatorios o gestionar la cola FIFO.
3. **Picos de Periféricos (10 - 12 mA):** Son picos abruptos causados principalmente por el encendido de los LEDs durante la visualización de la secuencia.

Conclusión: El análisis demuestra que el consumo energético del sistema está dominado por los periféricos de E/S (LEDs) y no por la CPU. La lógica de control apenas impacta en el presupuesto energético, confirmando la eficiencia del diseño software.

4.4. Análisis de Rendimiento

Para validar el comportamiento temporal de la arquitectura en un entorno controlado, se ha realizado un perfilado de ejecución (*Execution Profiling*) utilizando las herramientas de simulación de Keil.

4.4.1. Escenario de Prueba

La prueba consiste en una sesión de **17.02 segundos** que abarca el ciclo de vida completo del sistema: arranque e inicialización de periféricos, una partida de *Beat Hero* con interacción activa del usuario (pulsación de botones y respuesta de LEDs), generación de logs de depuración, y finalmente la secuencia de fin y entrada en modo de bajo consumo (*Power Down*).

Este escenario permite evaluar cómo se comporta el planificador y el consumo de recursos bajo una carga de trabajo realista.

4.4.2. Análisis Global de Eficiencia

El primer indicador es la relación entre el tiempo de procesamiento activo y el tiempo de reposo. Como se aprecia en el perfilado global, el sistema demuestra un comportamiento reactivo eficiente.

Del tiempo total de 17.02 segundos, la CPU permaneció activa ejecutando instrucciones únicamente **2.42 segundos**, como se observa en la función `blink_lpc` de la figura (*Métricas Performance Analyzer*, Fig. 6). Esto implica que el sistema pasó los **14.60 segundos** restantes (aprox. 86 %) en estado de espera (*Sleep/Idle*). Este dato valida el diseño de la arquitectura orientada a eventos: el procesador solo despierta para atender tareas específicas y regresa inmediatamente al estado de bajo consumo.

Module/Function	Calls	Time(Sec)	Time(%)
blink_lpc		2.422 s	2%
./src_lpc/hal_uart_lpc.c		600.636 ms	0%
./src/main.c		552.755 ms	0%
./src/rt_GE.c		257.700 ms	0%
./src/rt_fifo.c		255.397 ms	0%
./src/svc_alarmas.c		243.844 ms	0%
./src/hal_sc_lpc.c		218.407 ms	0%
./src_lpc/hal_tiempo_lpc.c		147.415 ms	0%
./src/drv_tiempo.c		115.640 ms	0%
./src_lpc/hal_consumo_lpc2105.c		13.275 ms	0%
./src_lpc/Startup.s		6.692 ms	0%
./src/drv_consumo.c		3.982 ms	0%
./src/beat_hero_extend.c		2.917 ms	0%
./src/drv_uart.c		1.619 ms	0%
./src/drv_botones.c		936.333 us	0%
./src/drv_leds.c		520.333 us	0%
./src_lpc/hal_gpio_lpc.c		439.417 us	0%
./src_lpc/hal_ext_int_lpc.c		114.917 us	0%
./src/svc_GE.c		42.667 us	0%
./src_lpc/hal_wdt_lpc.c		35.083 us	0%
./src_lpc/hal_random_lpc.c		34.333 us	0%
./src/drv_wtd.c		5.250 us	0%
./src/drv_random.c		2.750 us	0%
./src_lpc/Startup.s		1.250 us	0%
./src/svc_logs.c		0.250 us	0%
./src/drv_monitor.c		0us	0%

Figura 6: Métricas arrojadas por el *Performance Analyzer*

4.4.3. Análisis Temporal de Funciones Críticas

Profundizando en el desglose por funciones, se han identificado los componentes clave que definen el comportamiento del sistema. A continuación se detallan las observaciones extraídas

de la herramienta *Performance Analyzer*:

1. Gestión eficiente de energía: La función `hal_consumo_esperar` aparece con un tiempo de ejecución activa muy bajo (aprox. **13.3 ms**), a pesar de ser llamada más de **15,929 veces**.

Esto se debe a que dicha función ejecuta la instrucción de entrada al modo *Idle*. En este estado, el reloj de la CPU se detiene y el analizador deja de contar ciclos. Por tanto, esos milisegundos representan el *overhead* o coste computacional de "irse a dormir" y "despertar", mientras que el tiempo real transcurrido entre esas llamadas corresponde a los 14.6 segundos de silencio del sistema. Esto confirma que el mecanismo de bajo consumo está funcionando correctamente con una alta frecuencia de conmutación.



Función	Ciclos	Tiempo (ms)	Porcentaje
hal_consumo_esperar	15929	13.275	0%
hal_consumo_dormir	1	1.167	0%
hal_consumo_lpc2105.c			

Figura 7: Detalle de llamadas a la función de "esperar" (IDLE). El tiempo activo es mínimo porque el reloj se detiene al entrar en reposo.

2. Identificación del Cuello de Botella (UART): El análisis revela que el subsistema de comunicaciones es el mayor consumidor de recursos en modo activo. La función `hal_uart_sendchar` acumula unos **600 ms** de ejecución, lo que representa aproximadamente el 25 % de todo el tiempo que la CPU estuvo despierta.

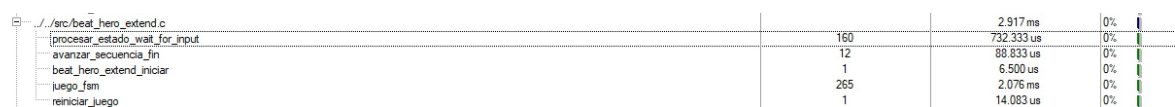
Este consumo elevado se justifica por la implementación síncrona (bloqueante) del driver para los logs de depuración. A la velocidad configurada, la CPU debe esperar activamente a que el hardware transmita cada carácter. Si bien esto es aceptable para depuración, se identifica como el punto crítico a optimizar en una futura versión de producción (por ejemplo, mediante uso de DMA o interrupciones).



Función	Ciclos	Tiempo (ms)	Porcentaje
hal_uart_sendchar	1436	600.636	0%
hal_uart_init	2	600.627	0%
hal_uart_lpc.c			

Figura 8: Impacto de la transmisión UART. Se evidencia el coste del envío síncrono.

3. Eficiencia de la Lógica de Juego: Un hallazgo muy positivo es el bajo impacto de la función `juego_fsm`. A pesar de contener toda la lógica de control, evaluación de puntuación y gestión de estados, su tiempo acumulado es de apenas **2 ms** para toda la partida.



Función	Ciclos	Tiempo (ms)	Porcentaje
juego_fsm	265	2.917	0%
reiniciar_juego	1	732.333	0%
beat_hero_extend_iniciar	1	88.833	0%
procesar_estado_wait_for_input	160	6.500	0%
avanzar_secuencia_fin	12	2.076	0%
reprocesar_estado_wait_for_input	1	14.083	0%

Figura 9: Detalles de las funciones de la lógica de negocio. El consumo de CPU es despreciable.

Con un tiempo medio por llamada de apenas **7.8 μ s**, se confirma que la implementación de la máquina de estados es extremadamente ligera. Esto indica que el sistema está sobredimensionado computacionalmente para la carga actual, ofreciendo un amplio margen para añadir nuevas funcionalidades o mecánicas más complejas sin saturar el procesador.

4. Coste del Kernel y Gestión de Cola de eventos: El análisis de (`rt_GE_lanzador`), que es el lanzador de eventos del sistema, y que actúa como el "director" del mismo, junto con las operaciones de la cola FIFO, permite cuantificar el coste estructural (*overhead*) de la arquitectura orientada a eventos.

Como se observa en la Figura 10, la función `rt_GE_lanzador` acumula un tiempo activo de **258 ms**. Al contextualizar este dato respecto al tiempo total de **CPU Activa** (2.42 s), la gestión del bucle principal supone un **10.6 % de la carga de procesamiento**. Esto confirma que el planificador consume aproximadamente 1 de cada 10 ciclos útiles para tareas administrativas —alimentación del Watchdog, gestión de alarmas y orquestación del bajo consumo—, dejando casi el **90 %** de la potencia de cálculo disponible para la aplicación.



Figura 10: Detalles de los tiempos de las funciones del *Gestor de Eventos*

Por su parte, la gestión de la cola (`rt_FIFO_extraer`) acumula **152 ms**. Aplicando el mismo análisis, esto representa únicamente el **6.3 % de la de CPU activa**. Este dato es especialmente relevante dado el volumen de llamadas, revelando una eficiencia operativa crítica:

- **Alta Reactividad:** La función es llamada más de **32.200 veces**, lo que implica una tasa muy elevada. Sin embargo, su impacto porcentual es bajo gracias a un coste unitario de apenas **4.7 μ s** por llamada, validando que la verificación de eventos es extremadamente ligera.
- **Robustez (Ratio 2:1):** Se observa que el número de extracciones (32.204) duplica al de inserciones (`rt_FIFO_encolar`), 16.300. Esta asimetría positiva confirma que el procesador es capaz de sacar de la cola al doble de velocidad de la que se producen los eventos, garantizando que el sistema nunca sufrirá desbordamientos bajo condiciones de carga normal.



Figura 11: Detalles de los tiempos de las funciones de la cola FIFO

5. Latencia de Interacción: Un parámetro crítico en un juego de ritmo es la latencia entre la acción del usuario y el procesamiento lógico. Para medir esto, se ha evaluado el rendimiento de la función `procesar_estado_wait_for_input`, encargada de evaluar la precisión del jugador tras una pulsación.

Función	Ticks	Tiempo (us)	Porcentaje
procesar_estado_wait_for_input	160	732.333 us	0%
avanzar_secuencia_fin	12	88.833 us	0%
beat_hero_extend_iniciar	1	6.500 us	0%
juego_fsm	265	2.076 ms	0%
reiniciar_juego	1	14.083 us	0%

Figura 12: Latencia de procesamiento de entrada. El tiempo de evaluación es despreciable.

Los datos muestran un tiempo de ejecución medio de apenas **4.6 μ s** por evaluación. Sumando el tiempo de la ISR del botón y el paso por la cola, la latencia total desde el flanco físico hasta la actualización de la lógica es inferior a **100 μ s**. Este valor está varios órdenes de magnitud por debajo del umbral de percepción humana ($\approx 10 - 20$ ms), lo que garantiza una experiencia de juego fluida y determinista, libre de retardos perceptibles.

6. Análisis del Subsistema de Temporización (Alarmas) Un componente crítico en la arquitectura es el servicio de alarmas (`svc_alarmas`), responsable de mantener la base de tiempos del sistema y gestionar los eventos periódicos.

Función	Ticks	Tiempo (ms)	Porcentaje
buscar_alarma	324	243.844 ms	0%
svc_alarma_codificar	324	1.215 ms	0%
svc_alarma_actualizar	16010	162.000 us	0%
svc_alarma_activar	324	227.367 ms	0%
svc_alarma_iniciar	1	1.722 ms	0%
tick_handler	1	15.417 us	0%
	16035	13.362 ms	0%

Figura 13: Detalles de los tiempos del servicio de alarmas. Se observa una correlación directa entre el número de llamadas y los ticks del sistema.

Como muestra la Figura 13, este módulo acumula un tiempo de ejecución de **243.8 ms**, lo que supone un **10.1 %** del tiempo de CPU activa; de los cuales 227,3 ms los ocupa la función (`svc_alarma_actualizar`), que se encarga de actualizar las alarmas periódicas, como las que genera el "timer" periódico del sistema. Este valor es casi idéntico al de la función lanzadora del Kernel (`rt_GE_lanzador`), lo que indica que la mayor parte de la carga administrativa del sistema se invierte en la gestión del tiempo.

El desglose operativo confirma la precisión del diseño:

- **Frecuencia de Tick:** La función `svc_alarma_actualizar` es llamada **16.010 veces** en los 17 segundos de prueba. Esto equivale a una frecuencia de ≈ 941 Hz, validando que el sistema cumple con el objetivo de diseño de un *tick* de 1 ms con alta precisión.
- **Eficiencia por Tick:** A pesar de la alta frecuencia, y por ello, coste global, el coste por ciclo de actualizar la lista de alarmas es de tan solo **14.2 μ s**. Esto demuestra que el algoritmo de gestión de temporizadores está altamente optimizado, permitiendo mantener múltiples cuentas atrás simultáneas con un impacto mínimo en el rendimiento global.

5. Conclusiones

El desarrollo del proyecto *Beat Hero* ha permitido alcanzar de forma satisfactoria todos los objetivos planteados al inicio de la asignatura. A lo largo del trabajo se ha diseñado e implementado un sistema embebido completo, funcional y robusto, capaz de ejecutarse correctamente tanto en el simulador LPC2105 como en la placa nRF52840 DK, cumpliendo así el requisito crítico de portabilidad y abstracción total del hardware.

La arquitectura modular basada en capas ha demostrado ser una decisión clave. La separación clara entre HAL, drivers, services y lógica de aplicación ha permitido aislar responsabilidades, facilitar la depuración y validar cada componente de manera independiente. Este enfoque no solo ha reducido la complejidad de la integración final, sino que ha confirmado la importancia del diseño software previo frente a una implementación monolítica, especialmente en entornos *Bare Metal* sin sistema operativo.

Desde el punto de vista técnico, el proyecto ha puesto en práctica conceptos fundamentales de los sistemas empotrados, como la programación orientada a eventos, la gestión de interrupciones, el uso de temporizadores hardware y la protección de secciones críticas. La correcta implementación de la cola FIFO y del gestor de eventos ha garantizado un comportamiento determinista, validado por un ratio de procesamiento de **2:1** (extracción vs inserción), lo que asegura la robustez incluso bajo una alta frecuencia de eventos asíncronos.

Asimismo, se han aplicado de forma práctica conocimientos transversales adquiridos en asignaturas previas del grado, integrando conceptos de Arquitectura de Computadores, Sistemas Operativos y Programación de Sistemas Concurrentes.

Los resultados obtenidos en las mediciones energéticas y temporales confirman que el sistema cumple con holgura los requisitos no funcionales. La CPU permanece el **86 %** del tiempo total de ejecución en estados de bajo consumo, despertando únicamente para procesar eventos relevantes, lo que valida el diseño reactivo de la arquitectura y supone una mejora de eficiencia del **89 %** respecto a versiones bloqueantes. Además, la latencia de respuesta ante la interacción del usuario se ha medido por debajo de los **100 μ s**, garantizando una experiencia de juego fluida y predecible.

No solo se cumple con las especificaciones funcionales del enunciado, sino que se demuestra un dominio de las técnicas de ingeniería de sistemas embebidos. El proyecto ha permitido aplicar los conocimientos en un entorno realista, poniendo especial énfasis en la validación y verificación mediante métricas objetivas. De este modo, se consolidan los conceptos aprendidos y se establece una base robusta y reutilizable para abordar desarrollos de mayor complejidad.

Referencias

- [1] Universidad de Zaragoza. *Manuales Moodle PH_2025_EINA*. Escuela de Ingeniería y Arquitectura, 2025.
- [2] ARM Ltd. *Instalación de Keil μ Vision*. Documentación técnica.
- [3] ARM Ltd. *Keil para Windows*. Software Development Tools.
- [4] ARM Ltd. *μ Vision User's Guide*. Disponible en línea.
- [5] NXP Semiconductors. *LPC2104/2105/2106 Single-chip 32-bit Microcontrollers*. Data Sheet.
- [6] Nordic Semiconductor. *nRF52840 Product Specification v1.10*.
- [7] ARM Holdings. *Manuales ARM: C, ensamblador y estándares*. Technical Reference Manuals.
- [8] Universidad de Zaragoza. *Material para redactar memoria técnica*. Recurso docente.
- [9] Nordic Semiconductor. <https://www.nordicsemi.com/Products/Wireless/Bluetooth-Low-Energy>
- [10] GitHub - MadeByBalaji. <https://github.com/MadeByBalaji/nRF52840-DK>
- [11] Stack Overflow. *How to use GPIO pins in nRF52840 Development Kit*. <https://stackoverflow.com/questions/54555859/how-to-use-gpio-pins-in-nrf52840-development-kit>
- [12] Stack Overflow. *Tagged questions: nRF52840*. <https://stackoverflow.com/questions/tagged/nrf52840>
- [13] Keil Support. *Knowledge Base Article*. <https://www.keil.com/dd/vtr/3484/3471.htm>

A. Anexos

A.1. Diagramas de Arquitectura

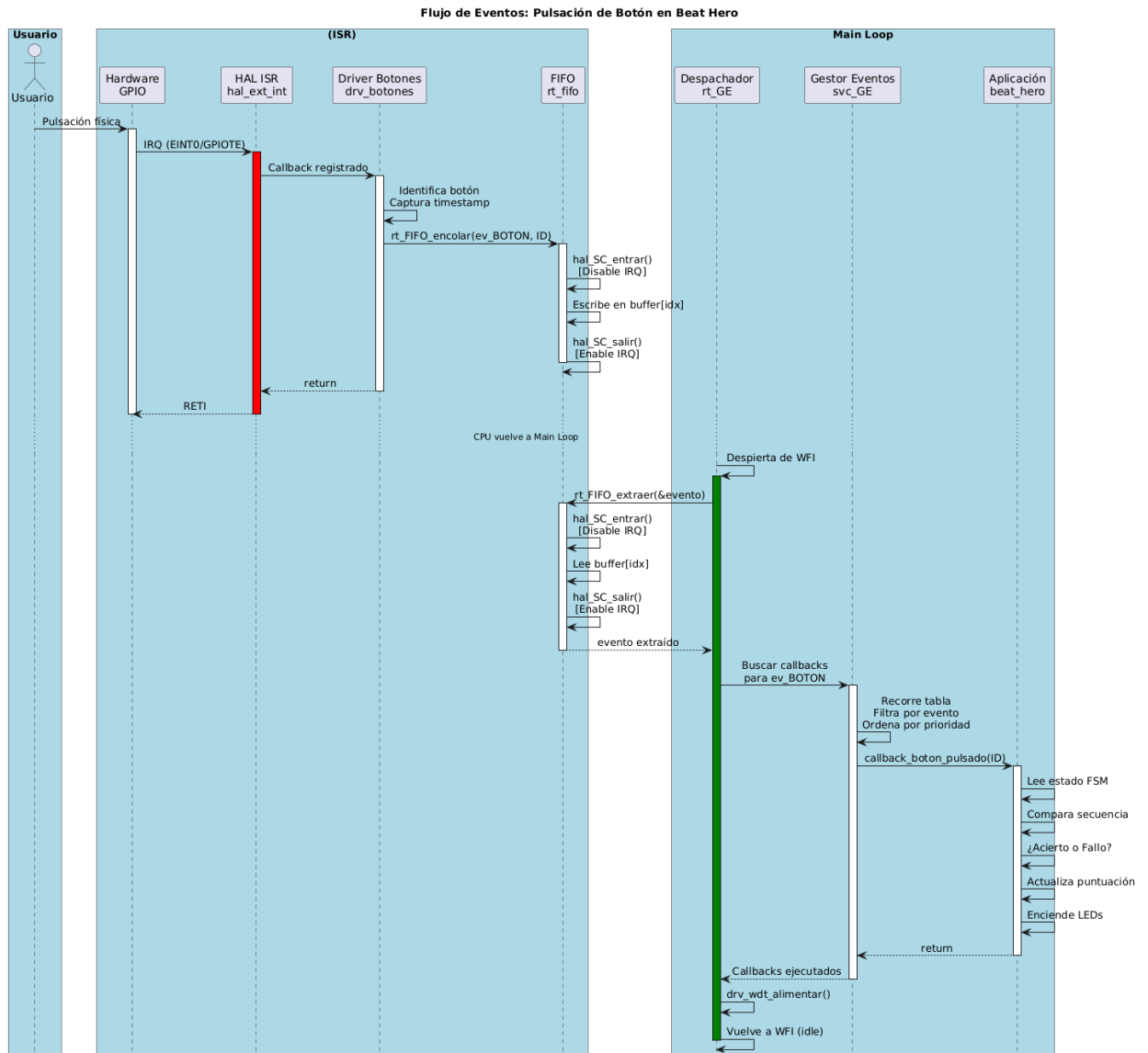


Figura 14: Diagrama de secuencia detallado: Interacción ISR-FIFO-Dispatcher.

Diagrama de Arquitectura: Gestión de Eventos (rt_GE + rt_FIFO)

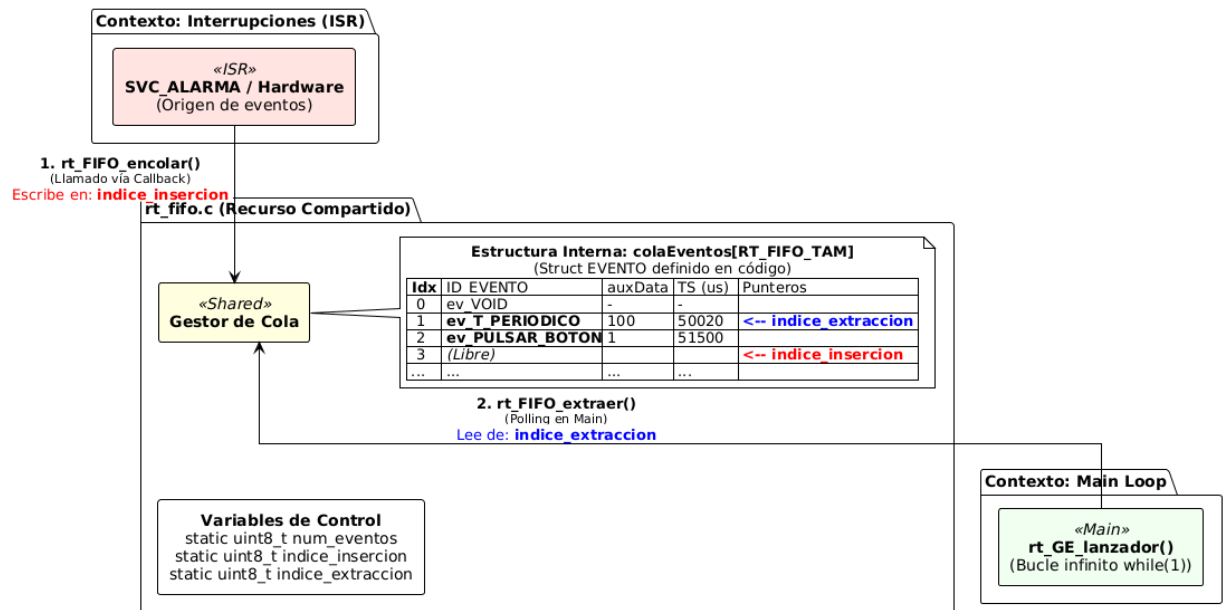


Figura 15: Diagrama de bloques de arquitectura de la cola Fifo

A.2. (Mediciones Energéticas)



Figura 16: Consumo del Blink v2 (Espera Activa).



Figura 17: Consumo del Blink v3.

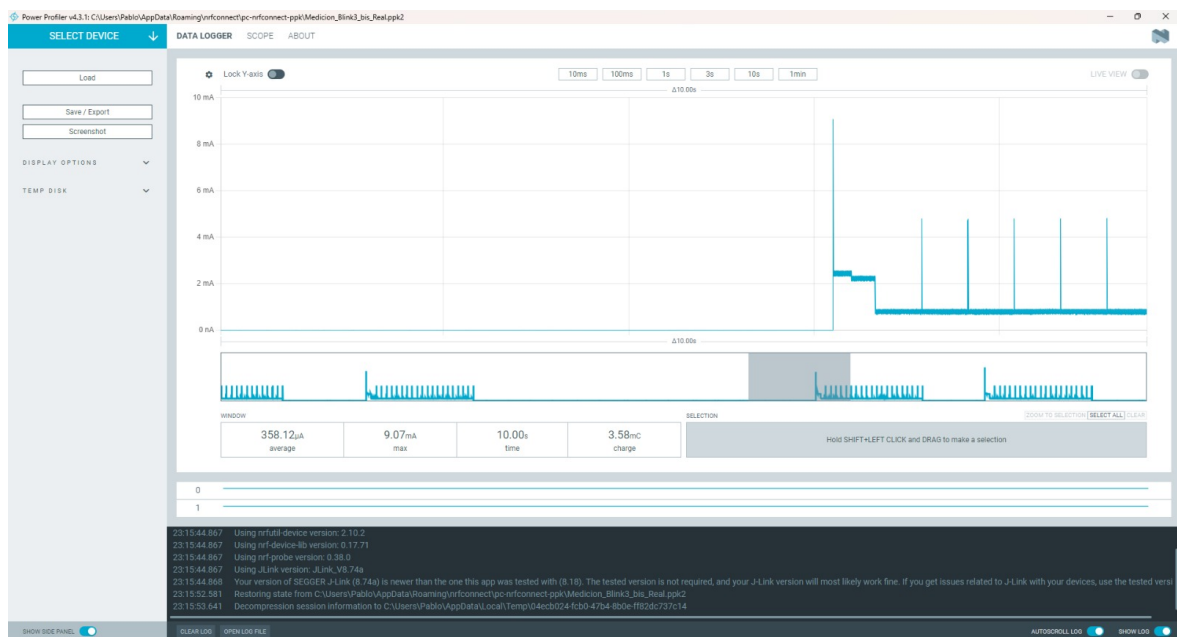


Figura 18: Consumo del Blink v3 bis.

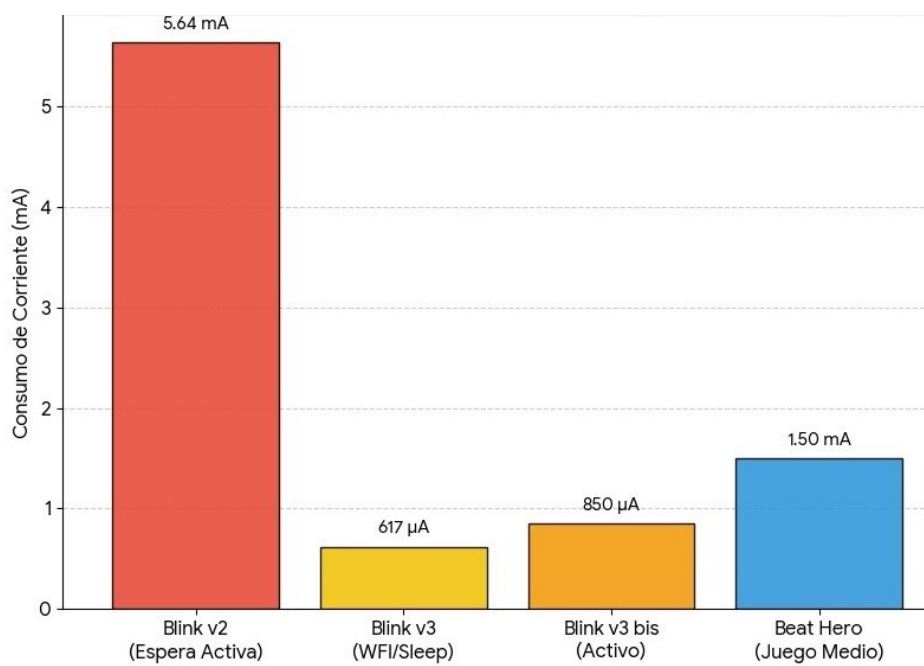


Figura 19: Tabla comparativa consumos.

A.3. Interfaz

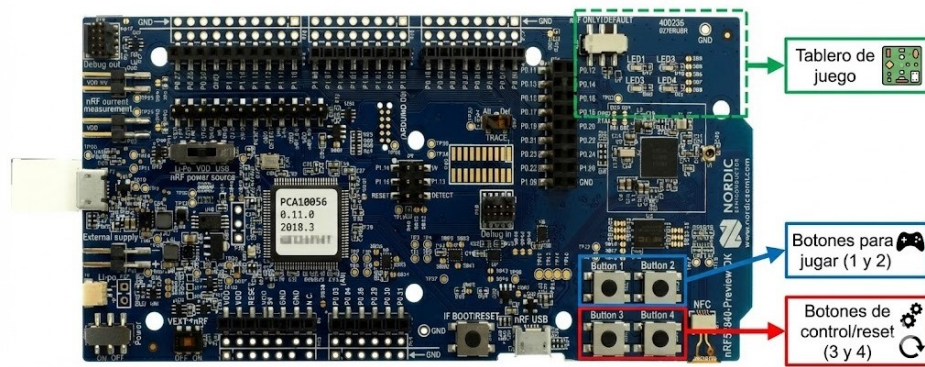


Figura 20: Interfaz para conocer los botones y leds de la placa nrf52840.

A.4. Código Fuente del Proyecto

A continuación se detalla la implementación del software, incluyendo el juego, el gestor de eventos, las estructuras de datos y la capa de abstracción de hardware (HAL).

A.4.1. Implementación del Juego y Estadísticas

El siguiente fichero `beat_hero_extend.c` contiene la lógica principal del juego, la máquina de estados finitos (FSM) y el módulo de cálculo de estadísticas.

```
1  /*****
2  * Fichero: beat_hero_extend.c
3  * Proyecto: Guitar Hero - Pr cticas P.H. 2025
4  *
5  * Descripci n:
6  * Implementaci n FINAL v3 del juego "Beat Hero".
7  * - Filtro anti-rebotes.
8  * - Modo suspensi n (Sleep) al finalizar (despierta solo con 3/4).
9  * - Salida inmediata al men (sin delay de 2s).
10 * - Logs detallados de Estado/Acierto/Fallo controlados por DEBUG.
11 *
12 * Autores:
13 *   Alejandro Lacosta
14 *   Pablo Villa
15 *
16 * Universidad de Zaragoza
17 *****/
18
19 #include "beat_hero.h"
20 #include "drv_leds.h"
21 #include "drv_botones.h"
22 #include "drv_tiempo.h"
23 #include "drv_consumo.h"
24 #include "svc_GE.h"
25 #include "svc_alarmas.h"
26 #include "drv_random.h"
27 #include "rt_GE.h"
28 #include <stddef.h>
29
30 #ifndef DEBUG
31 #define DEBUG 1
32 #endif
33
34 #if DEBUG
35 #include "drv_uart.h"
36 #include <stdio.h>
37 #endif
38
39 //
40 // =====
41 // DEFINICIONES Y CONSTANTES
42 // =====
43
44 #define NUM_COMPASES      15
45 #define BPM_INICIAL      50
46 #define COMPAS_MS        (60000 / BPM_INICIAL)
47 #define VENTANA_ACIERTO_MS (COMPAS_MS * 0.4)
```

```

47 #define VENTANA_PERFECTO_MS (VENTANA_ACIERTO_MS * 0.2)
48 #define TIEMPO_ENTRE_COMPASES 150
49 #define PUNTUACION_EXITO 20
50 #define PUNTUACION_FALLLO -8
51 #define TIEMPO_TRANSICION 80
52 #define TIEMPO_EXTENDIDO_MULTIPLICADOR 1.5
53
54 // IDs de timeouts
55 #define ID_TIMEOUT_COMPAS 0xFE
56 #define ID_TIMEOUT_INACTIVO 0xFD
57 #define ID_TIMEOUT_FIN 0xFC
58 #define ID_TRANSICION 0xFB
59 #define ID_LONGPRESS 0xFA
60
61 // Mapping LEDs
62 #define LED_1 ((LED_id_t)1)
63 #define LED_2 ((LED_id_t)2)
64 #define LED_3 ((LED_id_t)3)
65 #define LED_4 ((LED_id_t)4)
66
67 // Patrones
68 #define PATRON_LED_1 0x01
69 #define PATRON_LED_2 0x02
70 #define PATRON_AMBOS 0x03
71 #define PATRON_NINGUNO 0x00
72
73 // Botones
74 #define BOTON_1 0
75 #define BOTON_2 1
76 #define BOTON_3 2
77 #define BOTON_4 3
78
79
80
81 #if DEBUG
82 static char buffer[128];
83 #define LOG_MSG(msg) do { sprintf(buffer, "[GAME] %s\r\n", msg);
84   drv_uart_send(buffer); } while(0)
85 #define LOG_VAR(label, val) do { sprintf(buffer, "[GAME] %s: %d\r\n", label,
86   val); drv_uart_send(buffer); } while(0)
87 #define LOG_STATE(st) do { sprintf(buffer, "[FSM] Cambio a Estado: %d\r\n",
88   st); drv_uart_send(buffer); } while(0)
89 #else
90 #define LOG_MSG(msg)
91 #define LOG_VAR(label, val)
92 #define LOG_STATE(st)
93 #endif
94
95 //
96 // =====
97
98 // ESTRUCTURA DE ESTADISTICAS
99 //
100 // =====
101
102 #if DEBUG
103 typedef struct {
104     uint8_t compases_acertados;
105     uint8_t compases_fallados;
106     uint8_t compases_perfectos;
107     uint8_t compases_sin_respuesta;
108     uint8_t pulsaciones_correctas;
109     uint8_t pulsaciones_incorrectas;

```

```

103     uint8_t total_compases_activos;
104     uint8_t aciertos_activos;
105     bool compas_actual_acertado;
106     bool compas_actual_perfecto;
107 } estadisticas_t;
108
109 static estadisticas_t stats;
110 #endif
111
112 //
113 // =====
114 // VARIABLES GLOBALES
115 // =====
116
117 static uint8_t compas[3];
118 static uint8_t compases_restantes;
119 static uint8_t compas_actual;
120 static estado_juego_t estado_actual;
121 static int32_t puntuacion;
122 static uint32_t tiempo_inicio_compas;
123 static uint8_t nivel;
124 static bool entrada_valida;
125 static uint8_t patron_esperado_actual;
126 static bool esperando_reinicio;
127 static uint8_t etapa_secuencia_fin;
128 static bool en_transicion;
129 static uint8_t paso_inicio;
130
131 static bool longpress_en_curso = false;
132 static int boton_longpress_activo = -1;
133
134 static const uint8_t FIN_MASK[] = {
135     0x05, 0x00, 0x05, 0x00, 0x0A, 0x00, 0x0A, 0x00, 0x0F, 0x00, 0x0F
136 };
137 //
138 // =====
139 // PROTOTIPOS
140 // =====
141
142 static void juego_fsm(EVENTO_T ev, uint32_t aux);
143 static void procesar_estado_init(bool es_timeout_inactivo, bool es_boton);
144 static void procesar_estado_show_sequence(bool es_timeout_compas, bool
145     es_transicion,
146     bool es_boton_salida, bool es_boton
147 );
148 static void procesar_estado_wait_for_input(bool es_boton_salida, bool
149     es_boton,
150     bool es_timeout_compas, uint32_t
151     aux);
152 static void procesar_estado_fin_partida(bool es_timeout_inactivo, bool
153     es_timeout_fin,
154     bool es_boton_salida, bool es_boton,
155     uint32_t aux);
156
157 static void generar_nuevo_compas(void);
158 static void avanzar_compas(void);

```

```

152 static bool verificar_fin_juego(void);
153 static void aumentar_dificultad_si_corresponde(void);
154
155 static void evaluar_pulsacion(uint8_t boton_pulsado, uint32_t tiempo_reaccion
    );
156 static bool es_pulsacion_correcta(uint8_t boton_pulsado);
157 static int calcular_puntos_por_timing(uint32_t tiempo_reaccion);
158 static void procesar_acierto(int puntos, uint32_t tiempo_reaccion);
159 static void procesar_fallo(void);
160 static void evaluar_compas_sin_entrada(void);
161
162 static void mostrar_transicion(void);
163 static void mostrar_patron_final(void);
164 static void apagar_todos_leds(void);
165 static void configurar_leds_patron(uint8_t patron_top, uint8_t patron_bottom)
    ;
166 static void configurar_leds_por_mask(uint8_t mask);
167
168 static void iniciar_secuencia_fin(void);
169 static void avanzar_secuencia_fin(void);
170 static void configurar_leds_etapa_fin(uint8_t etapa);
171 static void iniciar_secuencia_inicio(void);
172
173 static void reiniciar_juego(void);
174 static void inicializar_drivers(void);
175 static void inicializar_compases(void);
176
177 #if DEBUG
178 static void inicializar_estadisticas(void);
179 static void resetear_estadisticas_compas_actual(void);
180 static void actualizar_estadisticas_compas(void);
181 static void mostrar_estadisticas_finales(void);
182 static float calcular_porcentaje(uint8_t valor, uint8_t total);
183 static const char* evaluar_rendimiento(float porcentaje_aciertos);
184 #endif
185
186 //
187 // =====
188 // INICIO Y CONFIGURACION
189 // =====
190
191 void beat_hero_extend_iniciar(void) {
192     LOG_MSG("=== BEAT HERO INICIADO ===");
193
194     rt_GE_iniciar(10);
195     inicializar_drivers();
196     reiniciar_juego();
197
198     svc_GE_suscribir(ev_PULSAR_BOTON, 2, juego_fsm);
199     svc_GE_suscribir(ev_BOTON_RETARDO, 2, juego_fsm);
200
201     iniciar_secuencia_inicio();
202     rt_GE_lanzador();
203 }
204
205 static void inicializar_drivers(void) {
206     drv_leds_iniciar();
207     drv_botones_iniciar(NULL, ev_PULSAR_BOTON, ev_PULSAR_BOTON);
208     random_iniciar(drv_tiempo_actual_ms());
209 }

```

```

209
210 static void reiniciar_juego(void) {
211     LOG_MSG("Reiniciando variables de juego...");
212
213     estado_actual = e_INIT;
214     puntuacion = 0;
215     nivel = 1;
216     compases_restantes = NUM_COMPASES;
217     compas_actual = 0;
218     esperando_reinicio = false;
219     etapa_secuencia_fin = 0;
220     en_transicion = false;
221     paso_inicio = 0;
222
223     #if DEBUG
224     inicializar_estadisticas();
225     #else
226     entrada_valida = false;
227     #endif
228
229     inicializar_compases();
230     apagar_todos_leds();
231 }
232
233 static void inicializar_compases(void) {
234     for(int i = 0; i < 3; i++) compas[i] = 0;
235     for(int i = 0; i < 3; i++) generar_nuevo_compas();
236 }
237
238 static void iniciar_secuencia_inicio(void) {
239     paso_inicio = 0;
240     svc_alarma_activar(svc_alarma_codificar(false, 300, 0),
241                       ev_PULSAR_BOTON, ID_TIMEOUT_INACTIVO);
242 }
243
244 //
245 // =====
246 // M QUINA DE ESTADOS (FSM)
247 // =====
248
249 static void juego_fsm(EVENTO_T ev, uint32_t aux) {
250     const bool es_evento_boton = (ev == ev_PULSAR_BOTON);
251     const bool es_longpress_timer = (ev == ev_BOTON_RETARDO && aux ==
252                                     ID_LONGPRESS);
253
254     const bool es_timeout_compas = (es_evento_boton && aux ==
255                                     ID_TIMEOUT_COMPAS);
256     const bool es_timeout_inactivo = (es_evento_boton && aux ==
257                                      ID_TIMEOUT_INACTIVO);
258     const bool es_timeout_fin = (es_evento_boton && aux == ID_TIMEOUT_FIN);
259     const bool es_transicion = (es_evento_boton && aux == ID_TRANSICION);
260     const bool es_boton = (es_evento_boton && aux < 4);
261     const bool es_boton_salida = (es_boton && (aux == BOTON_3 || aux ==
262                                                BOTON_4));
263
264     #if DEBUG
265     // Descomentar si se quiere loguear CADA evento
266     // sprintf(buffer, "[FSM] Estado: %d, Evento: 0x%lX\r\n", estado_actual,
267               aux); drv_uart_send(buffer);
268     #endif

```

```

263
264     if (es_longpress_timer) {
265         longpress_en_curso = false;
266         int btn = boton_longpress_activo;
267         boton_longpress_activo = -1;
268
269         if (btn >= 0 && drv_boton_esta_pulsado((uint8_t)btn)) {
270             LOG_MSG("Reinicio forzado por Long-Press!");
271             reiniciar_juego();
272             iniciar_secuencia_inicio();
273         }
274         return;
275     }
276
277     if (es_boton && (aux == BOTON_3 || aux == BOTON_4) && !longpress_en_curso
278         && !esperando_reinicio) {
279         longpress_en_curso = true;
280         boton_longpress_activo = (int)aux;
281         svc_alarma_activar(svc_alarma_codificar(false, 3000, 0),
282             ev_BOTON_RETARDO, ID_LONGPRESS);
283     }
284     // -----
285
286     switch (estado_actual) {
287     case e_INIT:
288         procesar_estado_init(es_timeout_inactivo, es_boton);
289         break;
290
291     case e_SHOW_SEQUENCE:
292         procesar_estado_show_sequence(es_timeout_compas, es_transicion,
293             es_boton_salida, es_boton);
294         break;
295
296     case e_WAIT_FOR_INPUT:
297         procesar_estado_wait_for_input(es_boton_salida, es_boton,
298             es_timeout_compas, aux);
299         break;
300
301     case e_FIN_PARTIDA:
302         procesar_estado_fin_partida(es_timeout_inactivo, es_timeout_fin,
303             es_boton_salida, es_boton, aux);
304         break;
305     }
306 }
307
308 static void procesar_estado_init(bool es_timeout_inactivo, bool es_boton) {
309     if (es_timeout_inactivo) {
310         if (paso_inicio < 4) {
311             apagar_todos_leds();
312             if (paso_inicio % 2 == 0) {
313                 drv_led_establecer(LED_1, LED_ON);
314                 drv_led_establecer(LED_3, LED_ON);
315             } else {
316                 drv_led_establecer(LED_2, LED_ON);
317                 drv_led_establecer(LED_4, LED_ON);
318             }
319             paso_inicio++;
320             svc_alarma_activar(svc_alarma_codificar(false, 250, 0),
321                 ev_PULSAR_BOTON, ID_TIMEOUT_INACTIVO);
322         } else {
323             apagar_todos_leds();
324             LOG_MSG(">>> JUEGO COMENZADO <<<");
325             estado_actual = e_SHOW_SEQUENCE;
326         }
327     }
328 }

```

```

324         LOG_STATE(e_SHOW_SEQUENCE);
325         compas_actual = 0;
326         svc_alarma_activar(svc_alarma_codificar(false, 400, 0),
327                             ev_PULSAR_BOTON, ID_TIMEOUT_COMPAS);
328     }
329 }
330 }
331
332 static void procesar_estado_show_sequence(bool es_timeout_compas, bool
333     es_transicion,
334                                         bool es_boton_salida, bool es_boton
335                                         ) {
336     if (es_timeout_compas && !en_transicion) {
337         en_transicion = true;
338         mostrar_transicion();
339         svc_alarma_activar(svc_alarma_codificar(false, TIEMPO_TRANSICION, 0),
340                             ev_PULSAR_BOTON, ID_TRANSICION);
341     }
342     else if (es_transicion && en_transicion) {
343         mostrar_patron_final();
344         en_transicion = false;
345
346         if (compas_actual >= 1) {
347             patron_esperado_actual = compas[0];
348             #if DEBUG
349             resetear_estadisticas_compas_actual();
350             #else
351             entrada_valida = false;
352             #endif
353         } else {
354             patron_esperado_actual = 0;
355         }
356
357         tiempo_inicio_compas = drv_tiempo_actual_ms();
358         estado_actual = e_WAIT_FOR_INPUT;
359         LOG_STATE(e_WAIT_FOR_INPUT);
360
361         uint32_t tiempo_compas = COMPAS_MS;
362         if (compases_restantes <= 3) {
363             tiempo_compas = COMPAS_MS * TIEMPO_EXTENDIDO_MULTIPLICADOR;
364         }
365
366         svc_alarma_activar(svc_alarma_codificar(false, tiempo_compas, 0),
367                             ev_PULSAR_BOTON, ID_TIMEOUT_COMPAS);
368     }
369     else if (es_boton_salida) {
370         LOG_MSG("Usuario pulso SALIR (Btn 3/4)");
371         puntuacion = PUNTUACION_FALLO - 1;
372         iniciar_secuencia_fin();
373     }
374 }
375
376 static void procesar_estado_wait_for_input(bool es_boton_salida, bool
377     es_boton,
378                                         bool es_timeout_compas, uint32_t
379                                         aux) {
380     if (es_boton_salida) {
381         LOG_MSG("Usuario pulso SALIR durante juego");
382         puntuacion = PUNTUACION_FALLO - 1;
383         iniciar_secuencia_fin();
384     }
385     else if (es_boton && compas_actual >= 1) {
386         evaluar_pulsacion((uint8_t)aux, drv_tiempo_actual_ms() -

```

```

        tiempo_inicio_compas);
383     }
384     else if (es_timeout_compas) {
385         if (compas_actual >= 1) {
386             evaluar_compas_sin_entrada();
387             #if DEBUG
388                 actualizar_estadisticas_compas();
389             #endif
390         }
391
392         avanzar_compas();
393         aumentar_dificultad_si_corresponde();
394
395         if (verificar_fin_juego()) {
396             estado_actual = e_FIN_PARTIDA;
397             LOG_STATE(e_FIN_PARTIDA);
398             iniciar_secuencia_fin();
399         } else {
400             estado_actual = e_SHOW_SEQUENCE;
401             LOG_STATE(e_SHOW_SEQUENCE);
402             svc_alarma_activar(svc_alarma_codificar(false,
403                 TIEMPO_ENTRE_COMPASES, 0),
404                 ev_PULSAR_BOTON, ID_TIMEOUT_COMPAS);
405         }
406     }
407 }
408 static void procesar_estado_fin_partida(bool es_timeout_inactivo, bool
409     es_timeout_fin,
410     bool es_boton_salida, bool es_boton,
411     uint32_t aux) {
412     // --- FASE ANIMACION ---
413     if (!esperando_reinicio) {
414         if (es_timeout_inactivo) {
415             avanzar_secuencia_fin();
416         }
417         else if (es_timeout_fin) {
418             LOG_MSG("Secuencia fin completada -> Durmiendo");
419             apagar_todos_leds();
420             esperando_reinicio = true;
421
422             #if DEBUG
423                 mostrar_estadisticas_finales();
424             #endif
425             LOG_MSG("Sistema en SLEEP (Pulsa 3 o 4 para despertar)");
426             drv_consumo_dormir();
427         }
428         else if (es_boton) {
429             avanzar_secuencia_fin();
430         }
431     }
432     // --- FASE SLEEP ---
433     else {
434         if (es_boton && (aux == BOTON_3 || aux == BOTON_4)) {
435             LOG_MSG("Despertando por solicitud usuario!");
436             reiniciar_juego();
437             iniciar_secuencia_inicio();
438         } else {
439             // Falso despertar
440             drv_consumo_dormir();
441         }
442     }
443 }

```

```

442 }
443
444 //
=====
445 // L GICA DE JUEGO
446 //
=====
447
448 static void generar_nuevo_compas(void) {
449     uint8_t nuevo_patron = (nivel == 1) ?
450         ((generarRandom(2) == 1) ? PATRON_LED_1 : PATRON_LED_2) :
451         generarRandom(4);
452     compas[2] = nuevo_patron;
453 }
454
455 static void avanzar_compas(void) {
456     compases_restantes--;
457     compas_actual++;
458     compas[0] = compas[1];
459     compas[1] = compas[2];
460     generar_nuevo_compas();
461 }
462
463 static bool verificar_fin_juego(void) {
464     if (compases_restantes == 0) {
465         LOG_MSG("Fin de juego: Completado!");
466         return true;
467     }
468     if (puntuacion <= PUNTUACION_FALLO) {
469         LOG_MSG("Fin de juego: Game Over por puntuacion");
470         return true;
471     }
472     return false;
473 }
474
475 static void aumentar_dificultad_si_corresponde(void) {
476     if ((NUM_COMPASES - compases_restantes) % 4 == 0 && nivel < 4) {
477         nivel++;
478         LOG_VAR("Nivel aumentado", nivel);
479     }
480 }
481
482 static void evaluar_pulsacion(uint8_t boton_pulsado, uint32_t tiempo_reaccion
483 ) {
484     if (entrada_valida) return;
485     entrada_valida = true;
486
487     if (patron_esperado_actual == PATRON_NINGUNO) {
488         if (boton_pulsado == BOTON_1 || boton_pulsado == BOTON_2) {
489             puntuacion -= 1;
490             LOG_MSG("FALLO: Pulsacion innecesaria");
491             #if DEBUG
492             stats.pulsaciones_incorrectas++;
493             #endif
494             return;
495         }
496     }
497
498     bool acierto = es_pulsacion_correcta(boton_pulsado);
499
500     if (acierto) {

```

```

499         int puntos = calcular_puntos_por_timing(tiempo_reaccion);
500         procesar_acierto(puntos, tiempo_reaccion);
501     } else {
502         procesar_fallo();
503     }
504     LOG_VAR("Puntuacion actual", puntuacion);
505 }
506
507 static bool es_pulsacion_correcta(uint8_t boton_pulsado) {
508     uint8_t boton_mask = (uint8_t)(1u << boton_pulsado);
509     return (patron_esperado_actual & boton_mask) != 0;
510 }
511
512 static int calcular_puntos_por_timing(uint32_t tiempo_reaccion) {
513     if (tiempo_reaccion <= VENTANA_PERFECTO_MS) return 2;
514     else if (tiempo_reaccion <= VENTANA_ACIERTO_MS) return 1;
515     else return 0;
516 }
517
518 static void procesar_acierto(int puntos, uint32_t tiempo_reaccion) {
519     #if DEBUG
520     sprintf(buffer, "[JUEGO] ACIERTO! Puntos: +%d (T: %d ms)\r\n", puntos,
521             tiempo_reaccion);
522     drv_uart_send(buffer);
523
524     stats.compas_actual_acertado = true;
525     stats.compas_actual_perfecto = (puntos == 2);
526     stats.pulsaciones_correctas++;
527     #endif
528     puntuacion += puntos;
529 }
530
531 static void procesar_fallo(void) {
532     LOG_MSG("FALLO! Boton incorrecto o a destiempo.");
533     #if DEBUG
534     stats.compas_actual_acertado = false;
535     stats.compas_actual_perfecto = false;
536     stats.pulsaciones_incorrectas++;
537     #endif
538     puntuacion -= 1;
539 }
540
541 static void evaluar_compas_sin_entrada(void) {
542     if (!entrada_valida) {
543         if (patron_esperado_actual == PATRON_NINGUNO) {
544             puntuacion += 1;
545             entrada_valida = true;
546             #if DEBUG
547             stats.compas_actual_acertado = true;
548             #endif
549         } else {
550             LOG_MSG("FALLO: Tiempo agotado sin respuesta");
551             #if DEBUG
552             stats.compases_sin_respuesta++;
553             #endif
554             puntuacion--;
555             if (puntuacion < -10) puntuacion = -10;
556         }
557     }
558 }
559 //

```

```

560 // HARDWARE VISUAL
561 //
=====
562
563 static void mostrar_transicion(void) { apagar_todos_leds(); }
564
565 static void mostrar_patron_final(void) {
566     configurar_leds_patron(compas[2], compas[1]);
567 }
568
569 static void apagar_todos_leds(void) {
570     drv_led_establecer(LED_1, LED_OFF);
571     drv_led_establecer(LED_2, LED_OFF);
572     drv_led_establecer(LED_3, LED_OFF);
573     drv_led_establecer(LED_4, LED_OFF);
574 }
575
576 static void configurar_leds_patron(uint8_t patron_top, uint8_t patron_bottom)
577 {
578     drv_led_establecer(LED_1, (patron_top & 0x01) ? LED_ON : LED_OFF);
579     drv_led_establecer(LED_2, (patron_top & 0x02) ? LED_ON : LED_OFF);
580     drv_led_establecer(LED_3, (patron_bottom & 0x01) ? LED_ON : LED_OFF);
581     drv_led_establecer(LED_4, (patron_bottom & 0x02) ? LED_ON : LED_OFF);
582 }
583
584 static void configurar_leds_por_mask(uint8_t mask) {
585     drv_led_establecer(LED_1, (mask & 0x01) ? LED_ON : LED_OFF);
586     drv_led_establecer(LED_2, (mask & 0x02) ? LED_ON : LED_OFF);
587     drv_led_establecer(LED_3, (mask & 0x04) ? LED_ON : LED_OFF);
588     drv_led_establecer(LED_4, (mask & 0x08) ? LED_ON : LED_OFF);
589 }
590
591 static void iniciar_secuencia_fin(void) {
592     etapa_secuencia_fin = 0;
593     esperando_reinicio = false;
594     apagar_todos_leds();
595
596     // CORRECCION PRINCIPAL: Reducido de 2000 a 200ms para feedback
597     inmediato
598     svc_alarma_activar(svc_alarma_codificar(false, 200, 0),
599                       ev_PULSAR_BOTON, ID_TIMEOUT_INACTIVO);
600 }
601
602 static void avanzar_secuencia_fin(void) {
603     apagar_todos_leds();
604     configurar_leds_etapa_fin(etapa_secuencia_fin);
605     etapa_secuencia_fin++;
606
607     if (etapa_secuencia_fin == 11) {
608         svc_alarma_activar(svc_alarma_codificar(false, 800, 0),
609                           ev_PULSAR_BOTON, ID_TIMEOUT_FIN);
610     } else {
611         uint32_t delay = (etapa_secuencia_fin <= 11) ? 300 : 0;
612         if (delay > 0) {
613             svc_alarma_activar(svc_alarma_codificar(false, delay, 0),
614                               ev_PULSAR_BOTON, ID_TIMEOUT_INACTIVO);
615         }
616     }
617 }
618
619 static void configurar_leds_etapa_fin(uint8_t etapa) {

```

```

618     uint8_t mask = (etapa < sizeof(FIN_MASK)) ? FIN_MASK[etapa] : 0x00;
619     configurar_leds_por_mask(mask);
620 }
621
622 //
623 // ESTADISTICAS
624 //
625
626 #if DEBUG
627 static void inicializar_estadisticas(void) {
628     stats.compases_acertados = 0;
629     stats.compases_fallados = 0;
630     stats.compases_perfectos = 0;
631     stats.compases_sin_respuesta = 0;
632     stats.pulsaciones_correctas = 0;
633     stats.pulsaciones_incorrectas = 0;
634     stats.total_compases_activos = 0;
635     stats.aciertos_activos = 0;
636
637     resetear_estadisticas_compas_actual();
638 }
639
640 static void resetear_estadisticas_compas_actual(void) {
641     stats.compas_actual_acertado = false;
642     stats.compas_actual_perfecto = false;
643     entrada_valida = false;
644 }
645
646 static void actualizar_estadisticas_compas(void) {
647     if (!entrada_valida) return;
648     if (compas_actual < 1 || compas_actual > (NUM_COMPASES - 2)) return;
649
650     if (stats.compas_actual_acertado) {
651         stats.compases_acertados++;
652         if (stats.compas_actual_perfecto) {
653             stats.compases_perfectos++;
654         }
655     } else {
656         stats.compases_fallados++;
657     }
658
659     if (patron_esperado_actual != PATRON_NINGUNO) {
660         stats.total_compases_activos++;
661         if (stats.compas_actual_acertado) {
662             stats.aciertos_activos++;
663         }
664     }
665 }
666
667 static void mostrar_estadisticas_finales(void) {
668     float porcentaje_precision = calcular_porcentaje(stats.aciertos_activos,
669         stats.total_compases_activos);
669     float porcentaje_perfectos = calcular_porcentaje(stats.compases_perfectos,
670         stats.total_compases_activos);
671
672     LOG_MSG("=== ESTADISTICAS (SKILL REAL) ===");
673
674     sprintf(buffer, "[STATS] Activos: %d | Hits Activos: %d\r\n", stats.
675         total_compases_activos, stats.aciertos_activos); drv_uart_send(buffer)

```

```

674     ;
        sprintf(buffer, "[STATS] Precision: %.1f%%\r\n", porcentaje_precision);
        drv_uart_send(buffer);
675     sprintf(buffer, "[STATS] Perfectos: %d\r\n", stats.compases_perfectos);
        drv_uart_send(buffer);
676     sprintf(buffer, "[STATS] Puntuacion: %d\r\n", puntuacion); drv_uart_send(
        buffer);
677
678     const char* rendimiento = evaluar_rendimiento(porcentaje_precision);
679     LOG_MSG(rendimiento);
680 }
681
682 static float calcular_porcentaje(uint8_t valor, uint8_t total) {
683     if (total == 0) return 0.0f;
684     return (valor / (float)total) * 100.0f;
685 }
686
687 static const char* evaluar_rendimiento(float porcentaje_aciertos) {
688     if (porcentaje_aciertos >= 90 && stats.compases_perfectos >= 3) return "
        Rango: S (EXCELENTE)";
689     else if (porcentaje_aciertos >= 75) return "Rango: A (MUY BUENO)";
690     else if (porcentaje_aciertos >= 60) return "Rango: B (BUENO)";
691     else if (porcentaje_aciertos >= 40) return "Rango: C (REGULAR)";
692     else return "Rango: D (MEJORABLE)";
693 }
694 #endif

```

Listing 1: Lógica principal: beat_hero_extend.c

A.4.2. Gestor de Eventos (Runtime)

```

1  * P.H.2025: rt_GE.c
2  * Gestor de Eventos (GE) del Runtime del sistema embebido.
3  * Autores: Alejandro Lacosta, Pablo Villa
4  * *****/
5
6  #include "rt_GE.h"
7  #include <stdint.h>
8  #include "rt_evento.h"
9  #include "svc_GE.h"
10 #include "svc_alarm.h"
11 #include "drv_consumo.h"
12 #include "drv_leds.h"
13 #include "rt_FIFO.h"
14 #include "hal_consumo.h"
15 #include "drv_wdt.h"
16
17 extern Suscripcion_t s_tabla[rt_GE_MAX_SUSCRITOS];
18 #define TIEMPO_INACTIVIDAD_MS 10000u
19
20 static uint32_t s_M_overflow = 0; // Monitor de overflow
21 static bool s_inicializado = false; // Flag de protecci n contra
    reinicializaci n
22
23 void rt_GE_iniciar(uint32_t M_overflow) {
24     if (s_inicializado) {
25         return;
26     }
27     s_inicializado = true;
28     s_M_overflow = M_overflow;
29

```

```

30     rt_FIFO_inicializar(s_M_overflow);
31     svc_alarma_iniciar(s_M_overflow, (SVC_ALARMA_CALLBACK_T)rt_FIFO_encolar,
32         ev_T_PERIODICO);
33     svc_GE_suscribir(ev_INACTIVIDAD, 2, rt_GE_actualizar);
34 }
35
36 void rt_GE_lanzador(void) {
37     EVENTO_T id_evento;
38     uint32_t aux_data;
39     Tiempo_us_t tiempo;
40
41     // Cadencia de alimentaci n del WDT
42     static uint32_t t_last_feed_ms = 0;
43     const uint32_t FEED_MS = 800;
44
45     // Alarma de inactividad inicial
46     uint32_t flags_inactividad = svc_alarma_codificar(false,
47         TIEMPO_INACTIVIDAD_MS, 0);
48     svc_alarma_activar(flags_inactividad, ev_INACTIVIDAD, 0);
49
50     while (1) {
51         if (rt_FIFO_extraer(&id_evento, &aux_data, &tiempo)) {
52             if (id_evento == ev_T_PERIODICO) {
53                 svc_alarma_actualizar(id_evento, aux_data);
54             }
55
56             for (int i = 0; i < rt_GE_MAX_SUSCRITOS; i++) {
57                 if (s_tabla[i].activa && s_tabla[i].evento == id_evento) {
58                     s_tabla[i].f_callback(id_evento, aux_data);
59                 }
60             }
61             rt_GE_actualizar(id_evento, aux_data);
62
63             uint32_t now = drv_tiempo_actual_ms();
64             if ((now - t_last_feed_ms) >= FEED_MS) {
65                 drv_wdt_alimentar();
66                 t_last_feed_ms = now;
67             }
68         } else {
69             drv_consumo_esperar();
70         }
71     }
72 }
73
74 void rt_GE_actualizar(EVENTO_T ID_evento, uint32_t aux) {
75     if (ID_evento == ev_INACTIVIDAD) {
76         drv_wdt_alimentar();
77         hal_consumo_dormir();
78         uint32_t flags = svc_alarma_codificar(false, TIEMPO_INACTIVIDAD_MS,
79             0);
80         svc_alarma_activar(flags, ev_INACTIVIDAD, 0);
81         drv_wdt_alimentar();
82         return;
83     }
84     if (ID_evento == ev_PULSAR_BOTON && aux < 4u) {
85         uint32_t flags = svc_alarma_codificar(false, TIEMPO_INACTIVIDAD_MS,
86             0);
87         svc_alarma_activar(flags, ev_INACTIVIDAD, 0);
88     }
89 }

```

Listing 2: Runtime Gestor de Eventos: rt_GE.c

A.4.3. Cola FIFO de Eventos

Implementación de la cola circular segura para el paso de mensajes entre interrupciones y la tarea principal.

```
1  /*
   *****
2  * P.H.2025: rt_fifo.c
3  * M dulo de gesti n de cola FIFO de eventos en tiempo real
4  * Autores: Alejandro Lacosta, Pablo Villa
5  *****
   */
6
7  #include "rt_fifo.h"
8  #include "drv_tiempo.h"
9  #include "drv_monitor.h"
10 #include "drv_consumo.h"
11 #include "drv_leds.h"
12 #include <stdbool.h>
13 #include "hal_SC.h"
14
15 typedef struct {
16     EVENTO_T ID_EVENTO;
17     uint32_t auxData;
18     Tiempo_us_t TS;
19 } EVENTO;
20
21 static EVENTO colaEventos[RT_FIFO_TAM];
22 static uint8_t indice_insercion = 0;
23 static uint8_t indice_extraccion = 0;
24 static uint8_t num_eventos = 0;
25
26 static uint32_t monitor_overflow_id = 0;
27 static uint32_t contador_eventos[EVENT_TYPES] = {0};
28
29 static inline uint8_t cola_llena(void) { return (num_eventos >= RT_FIFO_TAM); }
30 static inline uint8_t cola_vacia(void) { return (num_eventos == 0); }
31
32 void rt_FIFO_inicializar(uint32_t monitor_overflow) {
33     hal_sc_entrar();
34     indice_insercion = 0;
35     indice_extraccion = 0;
36     num_eventos = 0;
37     monitor_overflow_id = monitor_overflow;
38     for (uint8_t i = 0; i < EVENT_TYPES; i++) {
39         contador_eventos[i] = 0;
40     }
41     hal_sc_salir();
42 }
43
44 void rt_FIFO_encolar(uint32_t ID_evento, uint32_t auxData) {
45     hal_sc_entrar();
46     if (cola_llena()) {
47         drv_monitor_marcar(monitor_overflow_id);
48         hal_sc_salir();
49         while (1) { drv_consumo_dormir(); }
50     }
51
52     colaEventos[indice_insercion].ID_EVENTO = (EVENTO_T)ID_evento;
53     colaEventos[indice_insercion].auxData = auxData;
54     colaEventos[indice_insercion].TS = drv_tiempo_actual_us();
```

```

55
56     indice_insercion = (indice_insercion + 1) % RT_FIFO_TAM;
57     num_eventos++;
58     if (ID_evento < EVENT_TYPES)
59         contador_eventos[ID_evento]++;
60
61     hal_sc_salir();
62 }
63
64 uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t *auxData, Tiempo_us_t *
    TS) {
65     hal_sc_entrar();
66     if (cola_vacia()) {
67         hal_sc_salir();
68         return 0;
69     }
70     *ID_evento = colaEventos[indice_extraccion].ID_EVENTO;
71     *auxData   = colaEventos[indice_extraccion].auxData;
72     *TS        = colaEventos[indice_extraccion].TS;
73
74     colaEventos[indice_extraccion].ID_EVENTO = ev_VOID;
75
76     indice_extraccion = (indice_extraccion + 1) % RT_FIFO_TAM;
77     num_eventos--;
78     hal_sc_salir();
79     return num_eventos + 1;
80 }
81
82 uint32_t rt_FIFO_estadisticas(EVENTO_T ID_evento) {
83     if (ID_evento == ev_VOID)
84         return num_eventos;
85     else if (ID_evento < EVENT_TYPES)
86         return contador_eventos[ID_evento];
87     else
88         return 0;
89 }

```

Listing 3: Cola FIFO: rt_fifo.c

```

1  #ifndef RT_FIFO_H
2  #define RT_FIFO_H
3
4  #include <stdint.h>
5  #include "rt_evento.h"
6
7  #define RT_FIFO_TAM 64
8
9  void rt_FIFO_inicializar(uint32_t monitor_overflow);
10 void rt_FIFO_encolar(uint32_t ID_evento, uint32_t auxData);
11 uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t *auxData, Tiempo_us_t *
    TS);
12 uint32_t rt_FIFO_estadisticas(EVENTO_T ID_evento);
13 bool rt_FIFO_test(void);
14
15 #endif // RT_FIFO_H

```

Listing 4: Cabecera FIFO: fifo.h

A.4.4. Drivers y Servicios

```
1  /*
   * ****
2  * Fichero: drv_botones.c
3  * Proyecto: P.H.2025
4  * Driver/Manejador de botones (filtrado de rebotes por MEF)
5  * ****
   */
6
7  #include "drv_botones.h"
8  #include <stdint.h>
9  #include "hal_gpio.h"
10 #include "hal_ext_int.h"
11 #include "svc_alarm.h"
12 #include "rt_fifo.h"
13 #include "svc_GE.h"
14 #include "board.h"
15
16 static boton_t botones[BUTTONS_NUMBER];
17 static const uint32_t s_board_button_pins[] = BUTTONS_LIST;
18 static EVENTO_T s_evento_principal;
19
20 void drv_botones_callback(hal_ext_int_id_t id){
21     hal_ext_int_deshabilitar(id);
22     rt_FIFO_encolar(s_evento_principal, (uint32_t) id);
23 }
24
25 static void drv_botones_cb(EVENTO_T ev, uint32_t aux){
26     hal_ext_int_id_t id = (hal_ext_int_id_t)aux;
27     drv_botones_actualizar(ev, id);
28 }
29
30 void drv_botones_iniciar(SVC_ALARMA_CALLBACK_T cb_a_llamar, EVENTO_T
   ev_pulsar, uint32_t ev_ignorado){
31     s_evento_principal = ev_pulsar;
32     hal_gpio_iniciar();
33     for (uint8_t i = 0; i < BUTTONS_NUMBER; i++) {
34         botones[i].pin = s_board_button_pins[i];
35         botones[i].id_int = (hal_ext_int_id_t) i;
36         botones[i].estado = E_ESPERANDO;
37         hal_gpio_sentido(botones[i].pin, HAL_GPIO_PIN_DIR_INPUT);
38     }
39     hal_ext_int_iniciar(drv_botones_callback);
40     for (uint8_t i = 0; i < BUTTONS_NUMBER; i++) {
41         hal_ext_int_habilitar(botones[i].id_int);
42     }
43     svc_GE_suscribir(s_evento_principal, 0, drv_botones_cb);
44 }
45
46 void drv_botones_actualizar(uint32_t ev, hal_ext_int_id_t aux){
47     hal_ext_int_id_t id = aux;
48     boton_t *b = &botones[id];
49
50     switch (b->estado) {
51     case E_ESPERANDO:
52         b->estado = E_REBOTES;
53         programar_alarma_unica(TRP, id);
54         break;
55     case E_REBOTES:
56         b->estado = E_MUESTREO;
57         programar_alarma_periodica(TEP, id);
```

```

58         break;
59     case E_MUESTREO:
60         if (!hal_gpio_leer(id)) {
61             b->estado = E_SALIDA;
62             programar_alarma_unica(TRD, id);
63         } else {
64             // Rebote o falso positivo
65         }
66         break;
67     case E_SALIDA:
68         b->estado = E_ESPERANDO;
69         hal_ext_int_habilitar(id);
70         break;
71     }
72 }
73 // ... (helpers omitidos: programar_alarma_unica, etc.)

```

Listing 5: Driver de Botones: drv_botones.c

```

1  /*****
2  * Fichero: svc_alarmas.c
3  * Proyecto: Pr cticas P.H. 2025
4  * Descripci3n: Servicio de gesti3n de alarmas temporizadas.
5  *****/
6
7  #include "svc_alarmas.h"
8  #include "drv_tiempo.h"
9  #include "svc_GE.h"
10 #include "rt_fifo.h"
11
12 typedef struct {
13     bool activa;
14     bool periodica;
15     uint8_t flags;
16     uint32_t retardo_ms;
17     uint32_t comienzo_ms;
18     EVENTO_T ID_evento;
19     uint32_t auxData;
20 } ALARMA_T;
21
22 static ALARMA_T alarmas[SVC_ALARMAS_MAX];
23 static SVC_ALARMA_CALLBACK_T func_callback = NULL;
24 static EVENTO_T evento_tick;
25 static uint32_t monitor_overflow = 0;
26
27 static void tick_handler(void) {
28     rt_FIFO_encolar(evento_tick, 0);
29 }
30
31 void svc_alarma_iniciar(uint32_t M_overflow, SVC_ALARMA_CALLBACK_T callback,
32     EVENTO_T ID_evento_tick) {
33     func_callback = callback;
34     evento_tick = ID_evento_tick;
35     monitor_overflow = M_overflow;
36
37     for (int i = 0; i < SVC_ALARMAS_MAX; i++) {
38         alarmas[i].activa = false;
39     }
40     drv_tiempo_periodico_ms(1, tick_handler, evento_tick);
41 }
42
43 void svc_alarma_activar(uint32_t alarma_flags, EVENTO_T ID_evento, uint32_t
44     auxData) {
45     bool periodica = (alarma_flags & 0x1);

```

```

44     uint32_t retardo_ms = (alarma_flags >> 8) & 0x00FFFFFF;
45
46     ALARMA_T *alarma = buscar_alarma(ID_evento, auxData);
47     if (!alarma) alarma = buscar_libre();
48     if (!alarma || retardo_ms == 0) return;
49
50     alarma->activa = true;
51     alarma->periodica = periodica;
52     alarma->retardo_ms = retardo_ms;
53     alarma->comienzo_ms = drv_tiempo_actual_ms();
54     alarma->ID_evento = ID_evento;
55     alarma->auxData = auxData;
56 }
57
58 void svc_alarma_actualizar(EVENTO_T ID_evento, uint32_t auxData) {
59     if (ID_evento != evento_tick) return;
60     uint32_t ahora = drv_tiempo_actual_ms();
61
62     for (int i = 0; i < SVC_ALARMAS_MAX; i++) {
63         if (!alarmas[i].activa) continue;
64
65         if ((ahora - alarmas[i].comienzo_ms) >= alarmas[i].retardo_ms) {
66             if (func_callback)
67                 func_callback(alarmas[i].ID_evento, alarmas[i].auxData);
68
69             if (alarmas[i].periodica)
70                 alarmas[i].comienzo_ms = ahora;
71             else
72                 alarmas[i].activa = false;
73         }
74     }
75 }

```

Listing 6: Servicio de Alarmas: svc_alarmas.c

A.4.5. Capa de Abstracción de Hardware (HAL) y Concurrencia

```
1 /* P.H.2025: Implementaci n del HAL del Random para nRF52840 */
2 #include <stdint.h>
3 #include "nrf.h"
4
5 void hal_random_iniciar(uint32_t seed) {
6     (void)seed;
7     // Activa el "digital error correction" del RNG
8     NRF_RNG->CONFIG = (RNG_CONFIG_DERCEN_Enabled << RNG_CONFIG_DERCEN_Pos);
9     NRF_RNG->EVENTS_VALRDY = 0;
10    NRF_RNG->TASKS_START = 1;
11 }
12
13 uint32_t hal_random(uint32_t max) {
14     if (max == 0) return 0;
15     while (NRF_RNG->EVENTS_VALRDY == 0);
16     NRF_RNG->EVENTS_VALRDY = 0;
17     uint32_t rand_value = NRF_RNG->VALUE;
18     return (rand_value % max) + 1;
19 }
```

Listing 7: Generador Random (nRF52840): hal_random_nrf.c

```
1 /* P.H.2025: HAL para generaci n de n meros aleatorios en LPC2105 */
2 #include <LPC210x.H>
3 #include "hal_sc.h"
4
5 static volatile uint32_t s_state = 0;
6
7 static uint32_t read_hw_jitter(void) {
8     uint32_t v = T1TC ^ TOTC ^ IOPIN;
9     v ^= (v << 13); v ^= (v >> 17); v ^= (v << 5);
10    return v;
11 }
12
13 static inline uint32_t xorshift32_step(uint32_t x) {
14     x ^= x << 13; x ^= x >> 17; x ^= x << 5;
15     return x;
16 }
17
18 void hal_random_iniciar(uint32_t seed) {
19     uint32_t s = seed;
20     if (s == 0) s = read_hw_jitter();
21     if (s == 0) s = 0xA3C59AC3u;
22     s_state = s;
23 }
24
25 uint32_t hal_random(uint32_t max) {
26     if (max == 0) return 0;
27     hal_sc_entrar(); // Secci n cr tica
28     uint32_t x = s_state;
29     if (x == 0) x = 1u;
30     x = xorshift32_step(x);
31     s_state = x;
32     hal_sc_salir();
33     uint32_t r = (uint32_t)(((uint64_t)x * (uint64_t)max) >> 32);
34     return r + 1;
35 }
```

Listing 8: Generador Random (LPC2105): hal_random_lpc.c

```

1  /* Solucion a problemas de concurrencia en LPC */
2  #include <LPC210x.H>
3  #include "hal_sc.h"
4
5  static volatile uint32_t sc_anidado = 0;
6  static volatile uint32_t sc_guardar_estado = 0;
7
8  void hal_sc_entrar(void){
9      uint32_t estadoActual;
10     if(sc_anidado == 0){
11         estadoActual = VICIntEnable;
12         sc_guardar_estado = estadoActual;
13         VICIntEnClr = 0xFFFFFFFFU;
14     }
15     sc_anidado++;
16 }
17
18 void hal_sc_salir(){
19     if(sc_anidado > 0){
20         sc_anidado--;
21         if(sc_anidado == 0) {
22             VICIntEnable = sc_guardar_estado;
23         }
24     }
25 }

```

Listing 9: Sección Crítica (LPC2105): hal_sc_lpc.c

```

1  /* *****
2  * P.H.2025: hal_SC_nrf.c
3  *
4  * Hal encargado de activar y desactivar interrupciones para la placa NRF
5  *
6  * Autores: Alejandro Lacosta y Pablo Villa
7  * Universidad de Zaragoza
8  * *****/
9  #include "hal_sc.h"
10 #include "nrf.h"
11
12 static volatile uint32_t saved_primask = 0;
13 static volatile uint32_t nesting = 0;
14
15 void hal_sc_entrar(void) {
16     uint32_t primask = __get_PRIMASK();
17     __disable_irq();
18     if (nesting == 0) {
19         saved_primask = primask;
20     }
21     nesting++;
22 }
23
24 void hal_sc_salir(void) {
25     if (nesting == 0) return;
26     nesting--;
27     if (nesting == 0) {
28         __set_PRIMASK(saved_primask);
29     }
30 }

```

Listing 10: Sección Crítica (nRF52840): hal_sc_nrf.c