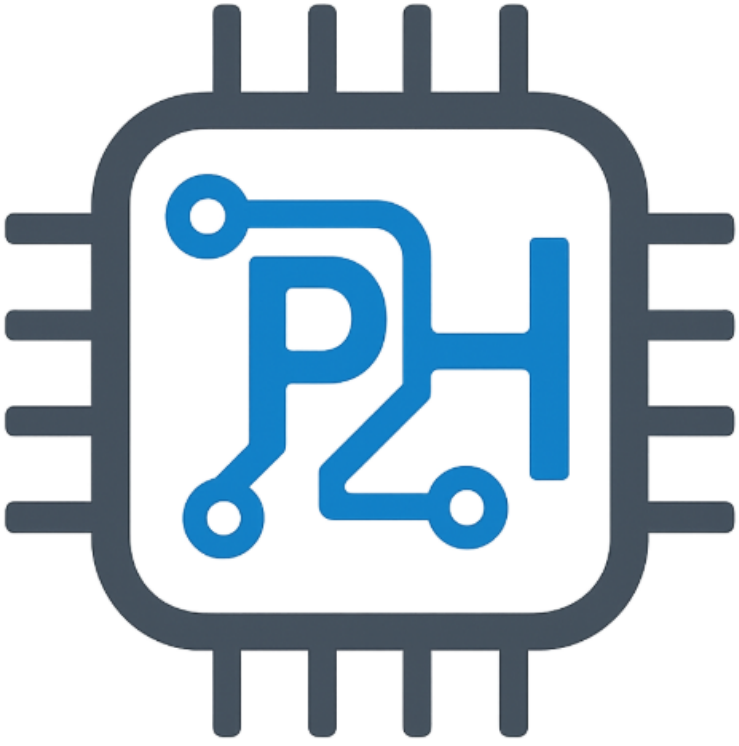
Memoria Final

Proyecto Hardware 2025

Beat Hero

Autores:

Fernando Pastor Peralta (897113)
Guillermo Ledesma Uche (896594)



Universidad de Zaragoza

Escuela de Ingeniería y Arquitectura

Grado en Ingeniería Informática

Proyecto Hardware

Diciembre 2025

Índice

Índice de figuras

Índice de cuadros

Resumen Ejecutivo

Este documento presenta la memoria técnica del proyecto Beat Hero, desarrollado en el marco de la asignatura "Proyecto Hardware" del Grado en Ingeniería Informática. El proyecto consiste en el desarrollo de un sistema empotrado con arquitectura orientada a eventos, implementando un juego de ritmo completo y funcional.

El sistema se ha desarrollado siguiendo un modelo de capas que garantiza la portabilidad entre dos plataformas ARM diferentes: LPC2105 (ARM7TDMI) y nRF52840 DK (ARM Cortex-M4F). El proyecto evolucionó a través de ocho sesiones de prácticas, desde un parpadeo básico de LED hasta un juego interactivo completo con gestión avanzada de eventos, alarmas software, bajo consumo energético y watchdog hardware.

Los objetivos principales alcanzados incluyen: el desarrollo de una arquitectura software modular y portable, la implementación de un modelo de programación orientado a eventos, la gestión eficiente del tiempo mediante temporizadores hardware con extensión a 64 bits, la minimización del consumo energético mediante modos de bajo consumo (WFI, System OFF), y la garantía de robustez mediante watchdog y manejo de condiciones de carrera (sección crítica).

Se ha seguido una metodología de desarrollo incremental, construyendo el sistema en capas: HAL (Hardware Abstraction Layer), Drivers, Runtime, Servicios y Aplicación. El desarrollo se realizó utilizando Keil µVision 5 como entorno de desarrollo, con configuraciones separadas para depuración (O0) y producción (O3). Se implementaron mecanismos de protección mediante secciones críticas para evitar condiciones de carrera, y se utilizaron monitores GPIO para correlacionar el estado del software con mediciones de osciloscopio.

El sistema final implementa exitosamente el juego Beat Hero, un juego de ritmo donde el jugador debe pulsar botones correspondientes a patrones de LEDs al ritmo de un tempo que aumenta progresivamente. El juego incluye: un sistema de puntuación basado en precisión temporal, tres niveles de dificultad, y gestión completa de estados. Se han completado satisfactoriamente todas las sesiones de desarrollo, implementando gestión de eventos con cola FIFO (esta se asegura de evitar condiciones de carreara), sistema de alarmas software (periódicas y esporádicas), modos de bajo consumo con despertar por inactividad, y watchdog hardware. Las mediciones de consumo realizadas con Nordic PPK2 demuestran la eficiencia energética del sistema. El sistema es completamente portable entre las dos plataformas objetivo.

Todos los objetivos planteados han sido cumplidos satisfactoriamente. El sistema implementa una arquitectura robusta de 5 capas que garantiza la portabilidad, un modelo orientado a eventos eficiente, gestión de tiempo de 64 bits, bajo consumo energético, y watchdog para recuperación ante bloqueos. El juego Beat Hero funciona correctamente y demuestra la viabilidad del sistema completo. El proyecto ha permitido adquirir amplios conocimientos en el desarrollo de sistemas empotrados, gestión de concurrencia en los mismos, optimización energética y diseño de arquitecturas software portables.

1. Introducción

1.1. Contexto del Proyecto

El presente proyecto corresponde a la asignatura "Proyecto Hardware" y aborda el desarrollo de un sistema empotrado con arquitectura orientada a eventos, siguiendo un modelo de capas que garantiza la portabilidad entre diferentes plataformas hardware.

El sistema evoluciona a través de ocho sesiones de prácticas, partiendo de un parpadeo básico de LED (blink v1) basado en espera activa, hasta alcanzar un juego interactivo completo (Beat Hero) con gestión de eventos, alarmas software, bajo consumo energético y watchdog hardware.

1.2. Plataformas Objetivo

El sistema se ha desarrollado para dos plataformas ARM:

Característica	**LPC2105**	**nRF52840 DK**
Arquitectura	ARM7TDMI	ARM Cortex-M4F
Frecuencia	15 MHz (PCLK)	16 MHz (Timer)
LEDs disponibles	4	4
Botones	3 (EINT0-2)	4 (GPIOTE)
Temporizadores	Timer0, Timer1	TIMER0, TIMER1
Bajo consumo	Idle, Power-Down	WFI, System OFF

Cuadro 1: Comparación de características entre plataformas LPC2105 y nRF52840 DK

1.3. Alcance Funcional

El sistema final implementa:

- Juego Beat Hero: juego de ritmo con patrones de LEDs, niveles de dificultad progresiva, y sistema de puntuación.
- Juego Bit Counter Strike: juego de reflejos LED-botón con dificultad creciente.
- Gestión completa de eventos con cola FIFO y suscripciones por prioridad.
- Sistema de alarmas software (periódicas y esporádicas).
- Modos de bajo consumo con despertar por inactividad.
- Watchdog hardware para recuperación ante bloqueos.

2. Objetivos

Los objetivos principales de este proyecto son:

- Desarrollar una arquitectura software modular y portable entre diferentes microcontroladores ARM.
- Implementar un modelo de programación orientado a eventos que desacople la lógica de aplicación del hardware.
- Gestionar eficientemente el tiempo mediante temporizadores hardware con extensión a 64 bits.

- Minimizar el consumo energético mediante modos de bajo consumo (WFI, System OFF).
- Garantizar la robustez del sistema mediante watchdog y manejo de condiciones de carrera.
- Implementar un juego completo (Beat Hero) que demuestre la funcionalidad del sistema.

3. Metodología

La metodología seguida en este proyecto se basa en un desarrollo incremental y modular, construyendo el sistema en capas que garantizan la separación entre hardware y software. A continuación se detalla la arquitectura, herramientas y procesos utilizados.

3.1. Arquitectura General del Sistema

3.1.1. Visión por Capas

La arquitectura sigue un modelo de capas estricto que garantiza la portabilidad:

APLICACIÓN / JUEGO juego_beat.c, juego_counter.c	
SERVICIOS (svc_) svc_alarmas.c, svc_GE.c	
RUNTIME (rt_) rt_fifo.c, rt_GE.c, rt_sc.c	
DRIVERS (drv_) drv_leds.c, drv_botones.c, drv_tiempo.c, drv_consumo.c drv_monitor.c, drv_WDT.c	
HAL (hal_) hal_gpio, hal_tiempo, hal_ext_int, hal_consumo, hal_WDT	
LPC2105 hal_*_lpc.c	**nRF52840** hal_*_nrf.c

Cuadro 2: Arquitectura de capas del sistema empotrado

3.1.2. Descripción de Capas

HAL (Hardware Abstraction Layer):

- Define interfaces genéricas en headers compartidos (`hal_*.h`).
- Implementaciones específicas por plataforma en `src_lpc/` y `src_nrf/`.
- Abstrae registros de periféricos, vectores de interrupción y configuración de pines.

Drivers (drv_):

- Ofrecen servicios de alto nivel independientes del hardware.
- Utilizan exclusivamente la API del HAL.
- Ejemplos: `drv_leds_iniciar()`, `drv_tiempo_actual_us()`, `drv_consumo_esperar()`.

Runtime (rt_):

- Infraestructura de ejecución: cola de eventos, secciones críticas.
- `rt_fifo.c`: cola FIFO para eventos con sección crítica incorporada.
- `rt_GE.c`: bucle principal del gestor de eventos.
- `rt_sc.c`: sección crítica por plataforma.

Servicios (svc_):

- Funcionalidad reutilizable construida sobre el runtime.
- `svc_alarmas.c`: alarmas software periódicas/esporádicas.
- `svc_GE.c`: sistema de suscripción a eventos por prioridad.

Aplicación:

- Lógica específica del juego.
- Máquinas de estados que reaccionan a eventos.

3.1.3. Justificación del Modelo Orientado a Eventos

El modelo orientado a eventos ofrece ventajas significativas para sistemas empotrados:

1. **Desacoplamiento:** Los productores de eventos (ISRs, temporizadores) no conocen a los consumidores.
2. **Bajo consumo:** El procesador puede entrar en modo de bajo consumo cuando la cola está vacía.
3. **Concurrencia simplificada:** Un único bucle de despacho elimina la complejidad de múltiples hilos.
4. **Extensibilidad:** Nuevos módulos se suscriben a eventos existentes sin modificar código previo.

3.1.4. Separación Hardware / Software

La separación se logra mediante:

- **Headers genéricos** (`board.h`) que definen `LEDS_NUMBER`, `BUTTONS_LIST`, etc.
- **Compilación condicional** basada en la plataforma objetivo
- **Ficheros board_*.h** específicos (`board_lpc.h`, `board_nrf52840dk.h`)

3.2. Entorno de Desarrollo

3.2.1. Keil µVision

El proyecto utiliza Keil µVision 5 como entorno de desarrollo:

- **Workspace:** `blink.uvmpw` con proyectos separados para LPC y nRF
- **Carpeta `lpc/keil/`:** Proyecto para LPC2105 (simulado)
- **Carpeta `nrf/keil/`:** Proyecto para nRF52840 DK (placa física)

3.2.2. Configuraciones de Compilación

Se mantienen dos configuraciones de compilación:

Configuración	Optimización	Uso
Debug	`-O0`	Desarrollo, depuración paso a paso
Release	`-O3`	Pruebas de rendimiento y consumo

Cuadro 3: Configuraciones de compilación en Keil µVision

La macro `SESION` controla qué versión del programa se compila (1-8).

3.3. Gestión del Tiempo

3.3.1. Reloj Monótono de 64 bits

La precisión de microsegundos requiere contadores de 64 bits. Dado que los contadores hardware son de 32 bits, se implementa extensión por software.

LPC2105 (`hal_tiempo_lpc.c`):

```
static volatile uint32_t s_overflows_t1 = 0;

void T1_ISR(void) __irq {
    T1IR = 1;
    s_overflows_t1++;
    VICVectAddr = 0;
}

uint64_t hal_tiempo_actual_tick64(void) {
    uint32_t hi1, lo1, hi2, lo2;
    lo1 = T1TC;
    hi1 = s_overflows_t1;
    lo2 = T1TC;
    if (lo2 < lo1) {
        hi2 = s_overflows_t1;
        return ((uint64_t)hi2 * 0x100000000ULL) + lo2;
    }
    return ((uint64_t)hi1 * 0x100000000ULL) + lo2;
}
```

nRF52840 (`hal_tiempo_nrf.c`):

```
static volatile uint64_t tiempo_64b = 0;

void TIMER0_IRQHandler(void) {
    if (NRF_TIMER0->EVENTS_COMPARE[0]) {
        NRF_TIMER0->EVENTS_COMPARE[0] = 0;
        tiempo_64b += (1ULL << 32);
    }
}

uint64_t hal_tiempo_actual_tick64(void) {
    uint64_t t1, t2;
    uint32_t ahora;
    do {
        t1 = tiempo_64b;
        NRF_TIMER0->TASKS_CAPTURE[3] = 1;
        ahora = NRF_TIMER0->CC[3];
        t2 = tiempo_64b;
    } while (t1 != t2);
    return t1 + (uint64_t)ahora;
```

```
}
```

3.3.2. Precisión y Rango

Plataforma	Ticks/µs	Período overflow (32b)	Rango 64 bits
LPC2105	15	∼286 segundos	∼39.000 años
nRF52840	16	∼268 segundos	∼36.000 años

Cuadro 4: Precisión y rango del reloj monótono de 64 bits

3.3.3. Conversión Ticks-Tiempo

El driver `drv_tiempo.c` abstrae la conversión:

```
Tiempo_us_t drv_tiempo_actual_us(void) {
    uint64_t ticks = hal_tiempo_actual_tick64();
    return (Tiempo_us_t)(ticks / s_info.ticks_per_us);
}
```

3.3.4. Temporización Periódica

El temporizador periódico genera interrupciones a intervalos regulares para alimentar el sistema de alarmas:

```
void drv_tiempo_periodico_ms(Tiempo_ms_t ms,
                             void(*callback)(uint32_t, uint32_t),
                             uint32_t ID_evento) {
    s_cb_app = callback;
    s_id = ID_evento;
    uint32_t ticks = ms * 1000 * s_info.ticks_per_us;
    hal_tiempo_reloj_periodico_tick(ticks, drv_tiempo_RSI);
}
```

3.4. Gestión de Periféricos

3.4.1. LEDs

El driver `drv_leds.c` proporciona una API idempotente:

- `drv_leds_iniciar()`: configura pines como salida, apaga todos los LEDs
- `drv_led_establecer(id, estado)`: enciende/apaga un LED específico
- `drv_led_conmutar(id)`: invierte el estado actual
- `drv_led_estado(id, *out)`: consulta el estado lógico

Abstracción de polaridad:

- LPC2105: LEDs activos a nivel alto (`LEDS_ACTIVE_STATE = 1`)
- nRF52840: LEDs activos a nivel bajo (`LEDS_ACTIVE_STATE = 0`)

```
static inline uint32_t hw_level_from_status(LED_status_t st) {
    return (LEDS_ACTIVE_STATE ? (st == LED_ON) : (st == LED_OFF));
}
```

3.4.2. Botones e Interrupciones Externas

La gestión de botones se realiza en dos niveles:
HAL (`hal_ext_int_*.c`):

- Configura pines como entrada con pull-up.
- Registra ISR para detección de pulsación.
- Proporciona `hal_ext_int_habilitar/deshabilitar` para control fino.

Driver (`drv_botones.c`):

- Implementa máquina de estados para antirrebotes.
- Detecta pulsaciones cortas y largas ($\geq$3 segundos).
- Encola eventos `ev_BOTON_PULSAR` y `ev_PULSACION_LARGA`.

A continuación, la máquina de estados del driver de botones:

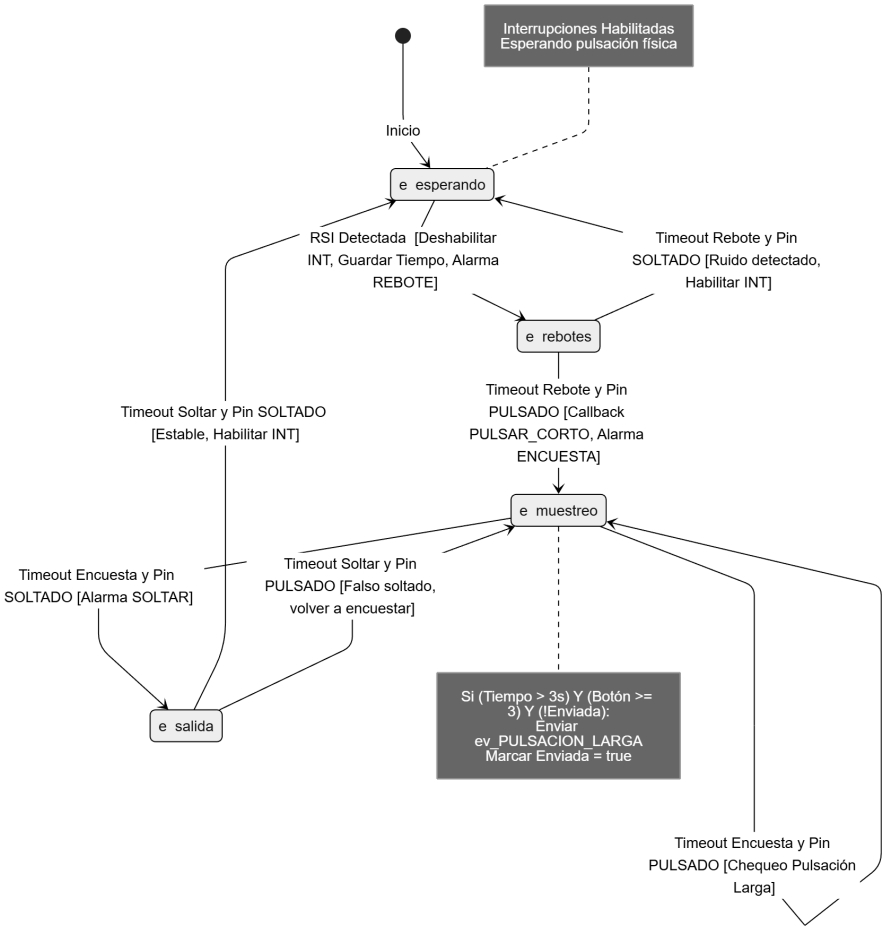


Figura 1: Máquina de estados del driver de botones con antirrebotes

Tiempos configurados:

- `RETARDO_REBOTE_MS = 50`: filtro tras detección de flanco
- `RETARDO_SOLTAR_MS = 50`: estabilización al soltar
- `RETARDO_ENCUESTA_MS = 20`: polling mientras está pulsado
- `PULSACION_LARGA = 3000`: umbral para pulsación larga

3.4.3. Diferencias de Implementación ISR

LPC2105: Interrupciones por nivel requiere reconfigurar PINSEL durante el antirrebote:

```
static void configurar_pin_como_eint(uint8_t pin) { ... }
static void configurar_pin_como_gpio(uint8_t pin) { ... }
```

nRF52840: Uso del mecanismo SENSE/LATCH de GPIOTE:

```
void hal_ext_int_habilitar(uint8_t id_boton) {
    // SENSE_Low -> genera evento PORT cuando pin=0
    cnf |= (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos);
}
```

3.4.4. Monitores GPIO

Los monitores (`drv_monitor.c`) son pines de salida usados para correlacionar el estado del software con mediciones de osciloscopio o analizador de consumo:

```
void drv_monitor_marcar(uint32_t id);    // pin -> HIGH
void drv_monitor_desmarcar(uint32_t id); // pin -> LOW
```

Uso típico (aunque puede variar según SESION a ejecutar y puede cambiarse a gusto):

- Monitor 1: indica ejecución de ISR periódica
- Monitor 2: marca overflow de reloj de 64 bits
- Monitor 3: señaliza modo de bajo consumo
- Monitor 4: marca overflow de cola de eventos, alarmas o suscripciones.

3.5. Watchdog

El watchdog hardware garantiza la recuperación ante bloqueos.

Alimentación desde el bucle principal:

```
void rt_GE_lanzador() {
    while(1) {
        drv_WDT_alimentar();  // previene reset
        // ... procesamiento de eventos
    }
}
```

3.6. Modelo de Concurrencia y Eventos

3.6.1. Definición de Eventos

Los eventos se definen en `rt_evento_t.h`:

```
typedef enum {
    ev_VOID = 0,           // evento nulo
    ev_T_PERIODICO = 1,    // tick del temporizador
    ev_BOTON_PULSAR = 2,   // boton pulsado
    ev_INACTIVIDAD = 3,    // timeout de inactividad
    ev_BOTON_RETARDO = 4,  // gestion rebotes
    ev_JUEGO_TICK = 5,     // temporizacion de juegos
    ev_PULSACION_LARGA = 6 // pulsacion >=3s
} EVENTO_T;

#define EVENT_TYPES 7
```

3.6.2. Cola FIFO de Eventos

La cola FIFO circular (`rt_fifo.c`) almacena eventos pendientes de procesar, tiene 32 posiciones, suficiente espacio para ráfagas de eventos y limitado para detectar overflow y facilitar diagnósticos ante errores.

Operaciones:

- `rt_FIFO_encolar(ID_evento, auxData)`: añade evento con timestamp
- `rt_FIFO_extraer(*ID_evento, *auxData, *TS)`: extrae siguiente evento
- `rt_FIFO_estadisticas(ID_evento)`: consulta contador por tipo

3.6.3. Gestión de Overflow

Si la cola se llena, el sistema entra en un bucle infinito diagnóstico tras marcar un monitor:

```
if (cola.contador == RT_FIFO_CAPACITY) {
    while(true) { /* overflow -> diagnostico */ }
}
```

Esta estrategia permite identificar el problema mediante el analizador conectado al monitor.

3.6.4. Secciones Críticas

Las secciones críticas protegen estructuras compartidas entre el bucle principal (modo Thread) y las ISRs (modo Handler).

Interface (`rt_sc.h`):

```
rt_sc_estado_t rt_sc_entrar(void);  // deshabilita IRQ, guarda estado
void rt_sc_salir(rt_sc_estado_t);   // restaura estado previo
```

3.7. Gestión de Energía

3.7.1. Modos de Consumo

El sistema implementa tres niveles de consumo:

Modo	Función	Comportamiento
Activo	Normal	Procesador ejecutando código
Espera	`drv_consumo_esperar()`	WFI, despierta por IRQ
Dormir	`drv_consumo_dormir()`	System OFF, requiere reset

Cuadro 5: Modos de consumo energético del sistema

3.7.2. Espera Activa vs Bajo Consumo vs Dormir Profundo

Espera activa:

```
delay = 1000000;

while (--delay) { /* busy loop */ }
```

- Consumo máximo constante

- No recomendado para dispositivos a batería

Modo esperar:

- Procesador detenido hasta próxima IRQ
- Reducción significativa de consumo durante idle

Modo Dormir Profundo:
nRF52840:

```
void hal_consumo_dormir(void) {
    NRF_POWER->SYSTEMOFF = POWER_SYSTEMOFF_SYSTEMOFF_Enter;
    __WFE();
    while (1) { }  // no retorna
}
```

LPC2105:

```
void hal_consumo_dormir(void) {
    EXTWAKE = 7;       // EINT0,1,2 como wake-up
    PCON |= 0x02;      // Power-down
    Reset_Handler();  // reinicio tras despertar
}
```

3.7.3. Despertar por Botón

Antes de entrar en modo dormir, se configuran los botones para despertar al sistema:

```
static void preparar_despertar_por_boton(void) {
    hal_ext_int_iniciar();
    for (uint8_t i = 1; i <= BUTTONS_NUMBER; ++i) {
        hal_ext_int_habilitar(i);  // SENSE=LOW
    }
}
```

3.7.4. Timeout por Inactividad

El sistema programa una alarma de inactividad de 20 segundos:

```
void rt_GE_lanzador() {
    svc_alarma_activar(
        svc_alarma_codificar(0, 20*1000, 0),
        ev_INACTIVIDAD, 0
    );
    // ... bucle de eventos
}
```

Al recibir `ev_BOTON_PULSAR`, la alarma se reprograma.

3.8. Servicios del Sistema

3.8.1. Servicio de Alarmas

El servicio de alarmas (`svc_alarmas.c`) gestiona hasta 6 alarmas concurrentes (suficiente para: inactividad + rebotes de hasta 4 botones + tick de juego), se utilizará un timer de 1ms para actualizar todas las alarmas.
Tipos de alarmas:

- **Esporádicas:** disparan una vez y se desactivan

- **Periódicas:** se reprograman automáticamente tras disparar

API:

```
void svc_alarma_iniciar(uint32_t monitor_overflow,
                        void (*cb_encolar)(uint32_t, uint32_t),
                        EVENTO_T ev_tick);

void svc_alarma_activar(svc_alarma_flags_t flags,
                        EVENTO_T id_evento,
                        uint32_t auxData);

void svc_alarma_actualizar(EVENTO_T id_evento, uint32_t auxData);
```

Funcionamiento interno:

1. El timer periódico genera `ev_T_PERIODICO`
2. `svc_alarma_actualizar` decrementa contadores
3. Al vencer, encola el evento asociado
4. Las alarmas periódicas se reprograman automáticamente

3.8.2. Gestor de Eventos (Dispatcher)

El gestor de eventos (`rt_GE.c`) implementa el bucle principal:

```
void rt_GE_lanzador() {
    while(1) {
        drv_WDT_alimentar();

        if (rt_FIFO_extraer(&id_ev, &aux, &ts) != (uint8_t)-1) {
            // Estadisticas de latencia
            Tiempo_us_t delta = drv_tiempo_actual_us() - ts;
            // ...

            // Invocar callbacks suscritos
            for (uint8_t i = 0; i < ge_tabla[id_ev].num_suscritos; i++) {
                ge_tabla[id_ev].lista_callbacks[i](id_ev, aux);
            }
        } else {
            drv_consumo_esperar();  // Cola vacia -> WFI
        }
    }
}
```

3.8.3. Suscripción y Cancelación

El servicio de suscripción (`svc_GE.c`) permite registrar callbacks por evento y prioridad:

```
void svc_GE_suscribir(EVENTO_T id_evento,
                      uint8_t prioridad,
                      rt_GE_callback_t callback);

void svc_GE_cancelar(EVENTO_T id_evento,
                     rt_GE_callback_t callback);
```

Orden de ejecución: Los callbacks se ejecutan en orden de prioridad (menor valor =>mayor prioridad).

4. Resultados

4.1. Bit Counter Strike

4.1.1. Descripción Funcional

Bit Counter Strike es un juego de reflejos donde el jugador debe pulsar el botón correspondiente al LED que se ilumina antes de que expire el tiempo límite. La dificultad aumenta progresivamente reduciendo los tiempos de respuesta y la pausa entre LEDs.

Mecánica:

1. Se enciende un LED secuencialmente (correspondiente a un par LED-botón)
2. El jugador tiene tiempo limitado para pulsar el botón correcto
3. Si acierta: aumenta dificultad (reduce tiempos) y continúa con el siguiente LED
4. Si falla: muestra parpadeo de error pero el juego continúa
5. Si timeout: el LED se apaga y continúa con el siguiente

4.1.2. Diagrama de estados del juego

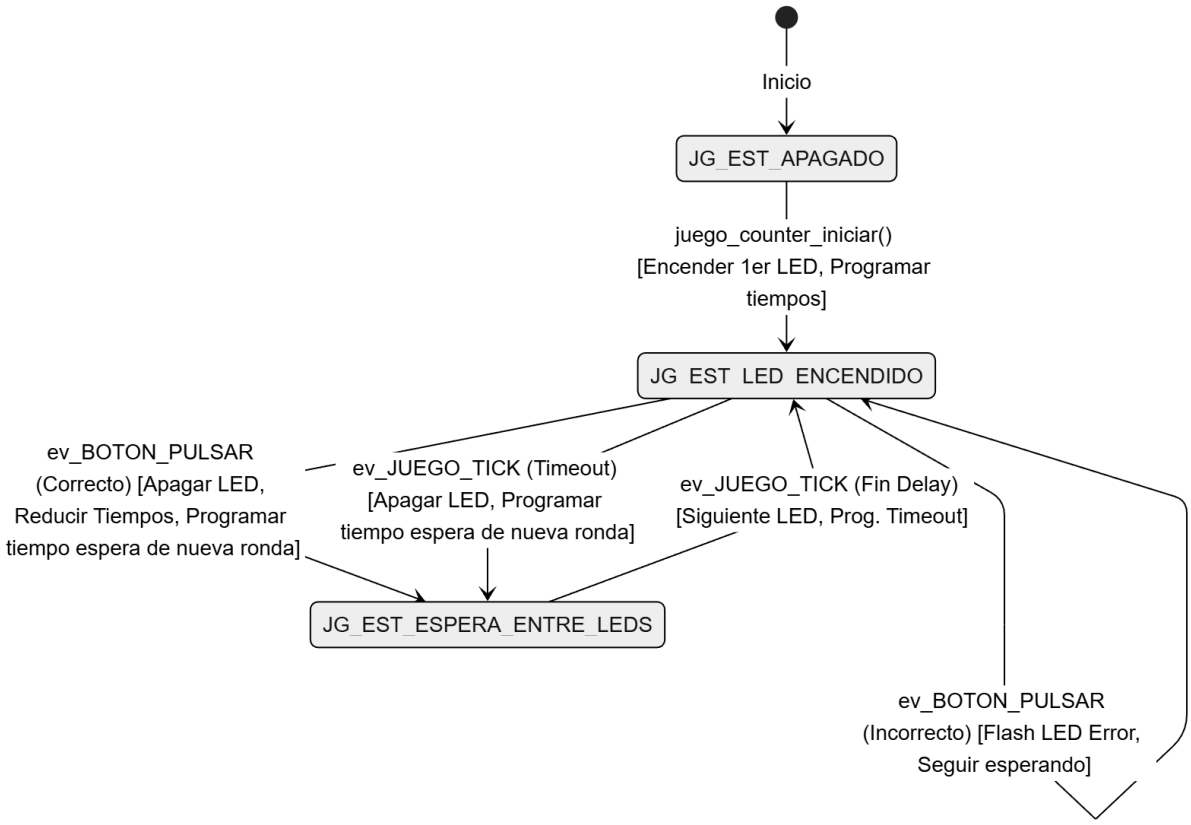


Figura 2: Diagrama de estados del juego Bit Counter Strike

4.1.3. Parámetros de Configuración

Parámetro	Valor Inicial	Valor Mínimo
Tiempo de respuesta	5000 ms	500 ms
Pausa entre LEDs	2000 ms	80 ms
Reducción (respuesta)	500 ms	—
Reducción (pausa)	20 ms	—

Cuadro 6: Parámetros de configuración del juego Bit Counter Strike

4.1.4. Progresión de Dificultad

Cada vez que el jugador pulsa el botón correcto a tiempo:

- Se reduce el tiempo de respuesta en 500 ms (hasta un mínimo de 500 ms).
- Se reduce la pausa entre LEDs en 20 ms (hasta un mínimo de 80 ms).

Los LEDs se encienden de forma secuencial y circular, utilizando el número de pares LED-botón válidos según la configuración de la placa.

4.1.5. Integración con el Sistema

El juego se integra mediante suscripciones al gestor de eventos:

```
svc_GE_suscribir(ev_BOTON_PULSAR, 2, juego_counter_actualizar);
svc_GE_suscribir(ev_JUEGO_TICK,   2, juego_counter_actualizar);
```

4.2. Juego Final: Beat Hero

4.2.1. Descripción Funcional

Beat Hero es un juego de ritmo donde el jugador debe pulsar los botones correspondientes a un patrón de LEDs al ritmo de un tempo (BPM) que aumenta progresivamente.

Mecánica:

1. Se muestran dos patrones de 2 LEDs (bits): el patrón actual y el siguiente
2. El jugador pulsa los botones correspondientes al patrón actual, botón 1 para LED 1 y botón 2 para LED 2
3. La precisión al pulsar determina la puntuación (+2, +1, 0, -1)
4. El tempo aumenta conforme se avanza en los niveles
5. La pulsación del botón 3 o 4 finaliza la partida

4.2.2. Diagrama de estados del juego

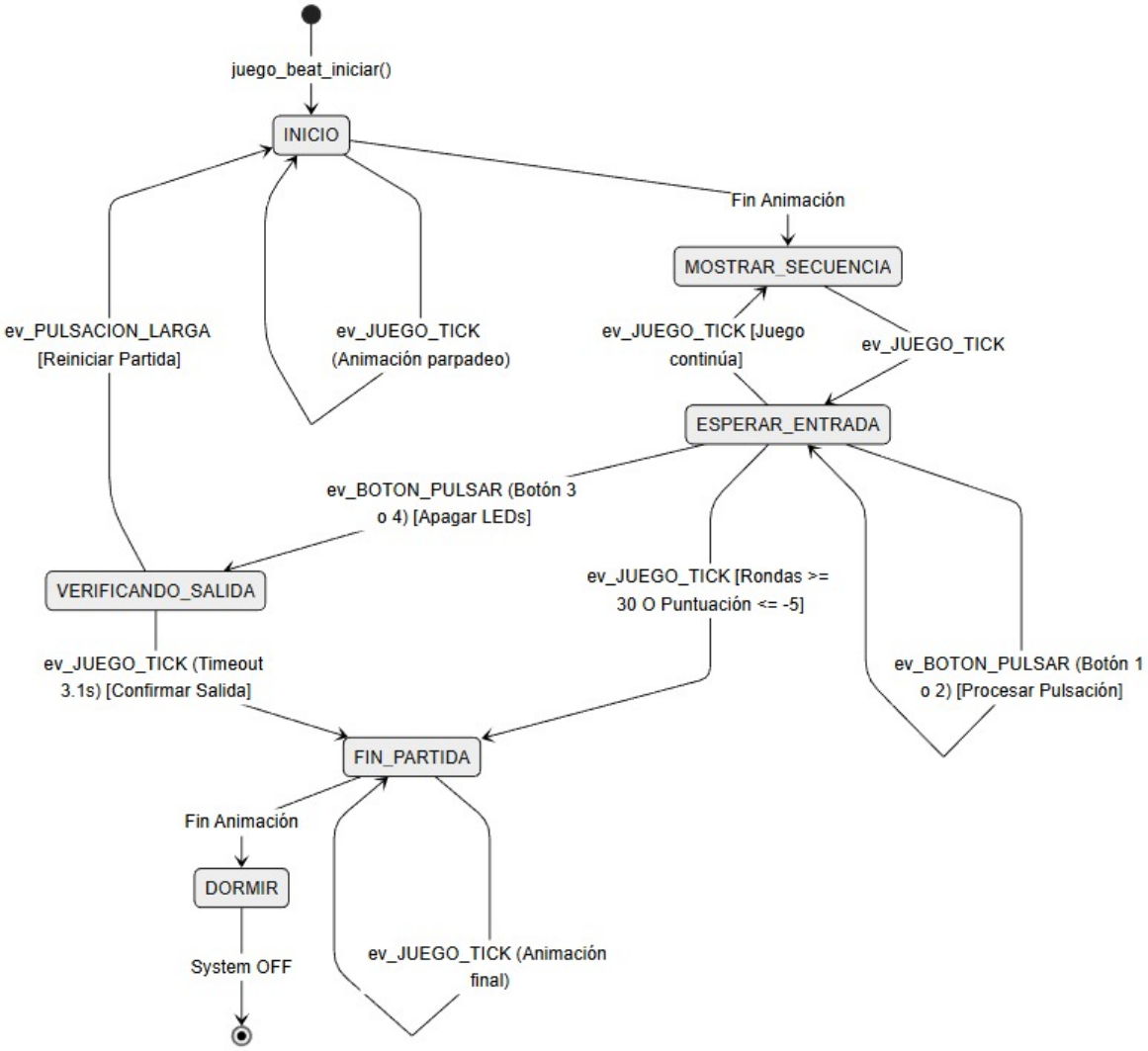


Figura 3: Diagrama de estados del juego Beat Hero

4.2.3. Sistema de Puntuación

La puntuación depende de la precisión al pulsar el botón:

Precisión	Puntos	Condición
Excelente	+2	$\leq$10 % del período del compás
Bien	+1	$\leq$20 % del período
Aceptable	0	$\leq$40 % (ventana)
Mal	-1	$>$40 % o botón incorrecto

Cuadro 7: Sistema de puntuación del juego Beat Hero

4.2.4. Progresión de Dificultad

- **BPM inicial:** 60
- **Compases por nivel:** configurables
- **Niveles máximos:** 3
- **En nivel máximo:** BPM aumenta +2 por compás

```
static void actualizar_nivel_y_bpm(void) {
    if ((j.compases_jugados % JUEGO_COMPASES_POR_NIVEL) == 0
        && j.nivel < JUEGO_NIVEL_MAX)
        j.nivel++;

    if (j.nivel == JUEGO_NIVEL_MAX)
        j.bpm += 2;

    j.t_compas_ms = 60000 / j.bpm;
    j.ventana_ms = (j.t_compas_ms * JUEGO_VENTANA_PORC) / 100;
}
```

4.2.5. Métricas y Estadísticas

El juego mantiene estadísticas detalladas:

- Puntuación total y por nivel
- Tiempo de reacción medio
- Tiempo de reacción más rápido
- Compases jugados

4.2.6. Reinicio por Pulsación Larga

Una pulsación larga ($\geq$3s) en el botón 3 o 4 reinicia la partida:

```
if (evento == ev_PULSACION_LARGA && j.estado != DORMIR) {
    juego_beat_iniciar();
    return;
}
```

4.3. Validación y Pruebas

Para consultar información sobre la ejecución del proyecto, el funcionamiento en detalle del juego Beat Hero, e información sobre cómo realizar diversas pruebas en detalle; puede consultarse README_EjecutarNuestroTestPH.pdf dentro de la carpeta del proyecto para obtener más información.

4.3.1. Pruebas de Módulos

Todas las sesiones de desarrollo se completaron satisfactoriamente:

Sesión	Prueba	Resultado
1	Blink por espera activa	OK
2	Blink con temporizador	OK
3	Blink periódico + callback	OK
4	Cola FIFO de eventos	OK
5	Test de overflow de cola	OK
6	Blink con bajo consumo	OK
7	Bit Counter Strike	OK
8	Beat Hero completo	OK

Cuadro 8: Resultados de pruebas de módulos por sesión

4.3.2. Test de Overflow de Cola

Con SESION en 5 se puede ejecutar un test explícito:

```
static void test_cola_overflow() {
    rt_FIFO_inicializar(4);

    __disable_irq();
    for (int i = 0; i < 32; ++i) rt_FIFO_encolar(ev_VOID, 0);
    __enable_irq();

    int fin = 1; // debe alcanzarse (32 OK, no overflow).

    // Overflow intencionado
    rt_FIFO_encolar(ev_VOID, 1); // debe ir a bucle infinito por overflow

    fin = -1;    // nunca debe llegar aqui
}
```

4.3.3. Verificación de Watchdog

En el estado `VERIFICANDO_SALIDA` se puede descomentar un bucle infinito para probar el watchdog (línea 328 del fichero "juego_beat.c"), después para provocar el error solo hay que pulsar el botón 3 o 4 durante la ejecución del juego:

```
case VERIFICANDO_SALIDA:
    // while(1); // descomentar para comprobar watchdog
```

El watchdog de 10 segundos provoca un reset si no se alimenta.

4.3.4. Mediciones de Consumo

Se realizaron capturas con Nordic PPK2 (Power Profiler Kit). Dentro de la carpeta del proyecto se puede encontrar en la subcarpeta de /contenido_practicas_anteriores los distintos archivos .ppk2 y algunas fotografías relacionadas a estos archivos:

- `blinkv2`: Trazado bastante plano en 6 mA con picos cuando conmuta el LED/temporizador. Indica CPU siempre despierta (busy-wait).

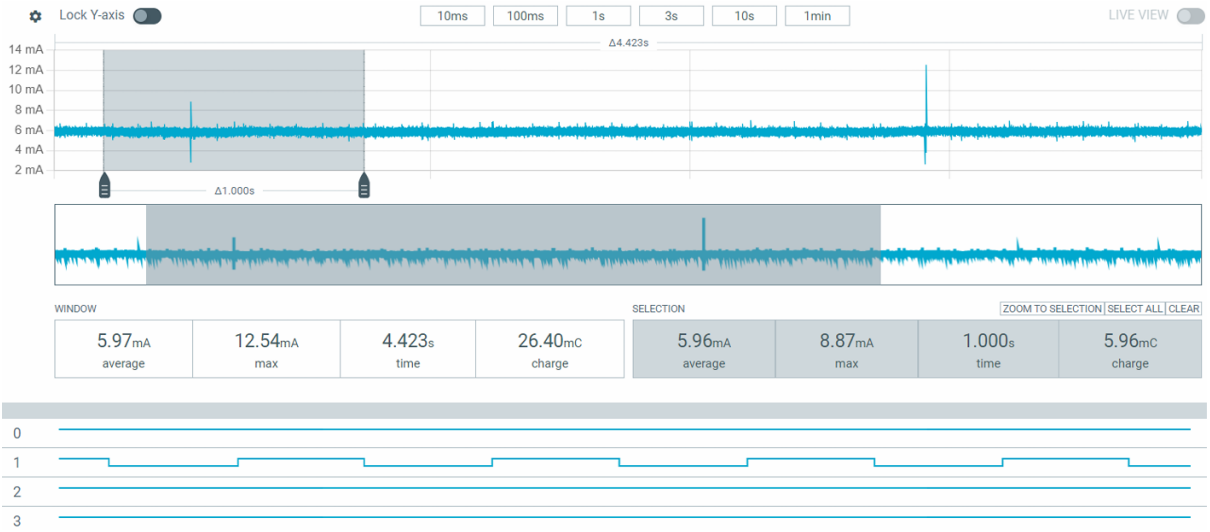


Figura 4: Medición de consumo - Blink v2 (espera activa)

- `blinkv3`: Señal con "pinchazos" estrechos: la CPU duerme y solo despierta para atender IRQ's y conmutar el LED.

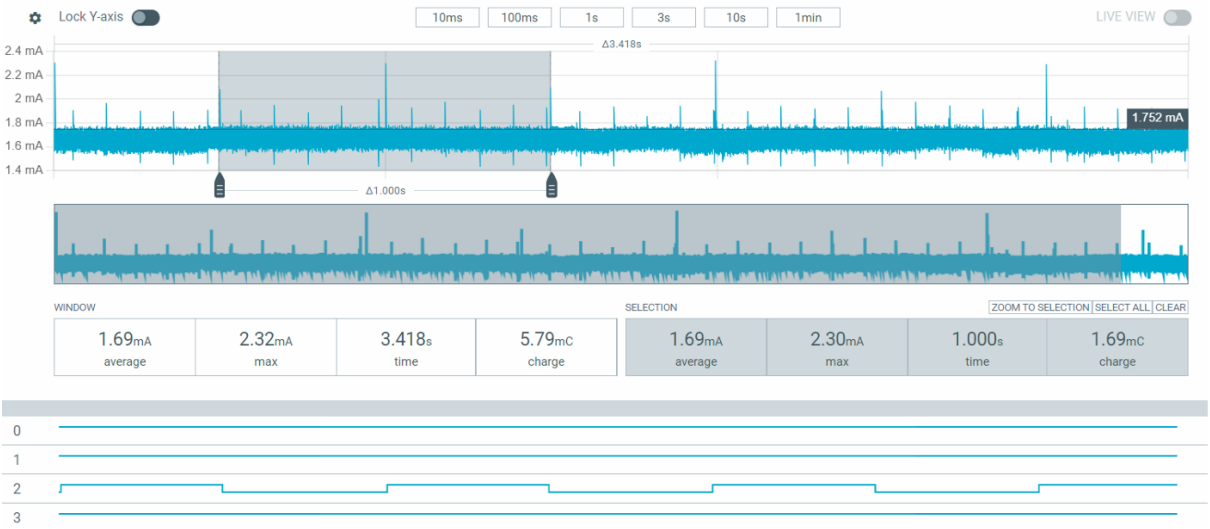


Figura 5: Medición de consumo - Blink v3 (bajo consumo con WFI)

- `blinkv3bis (despierto)`: consumo durante parpadeo activo.

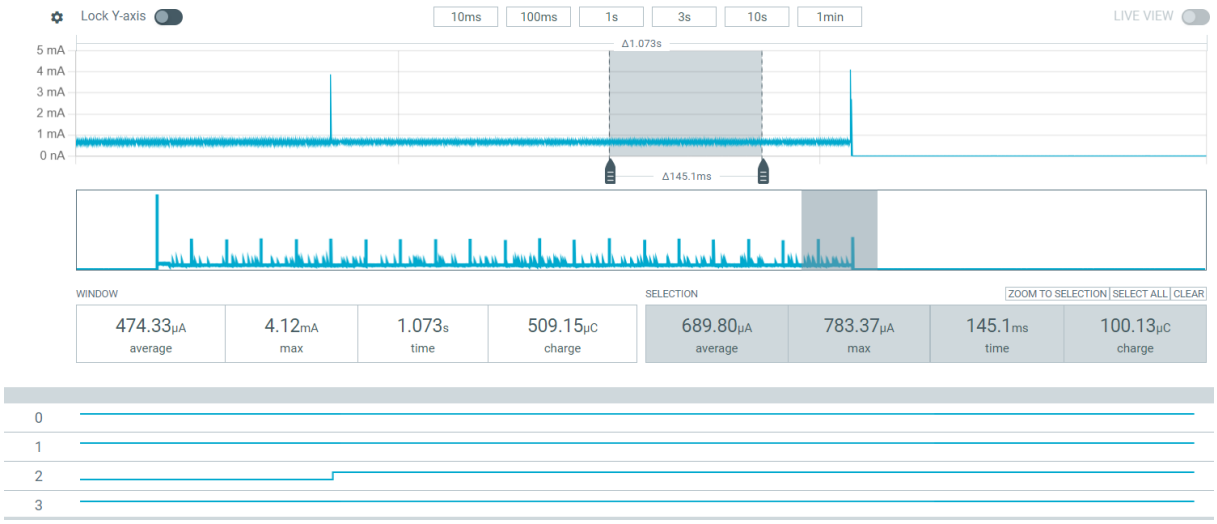


Figura 6: Medición de consumo - Blink v3 bis (modo despierto)

- `blinkv3bis (dormido)`: consumo en System OFF.

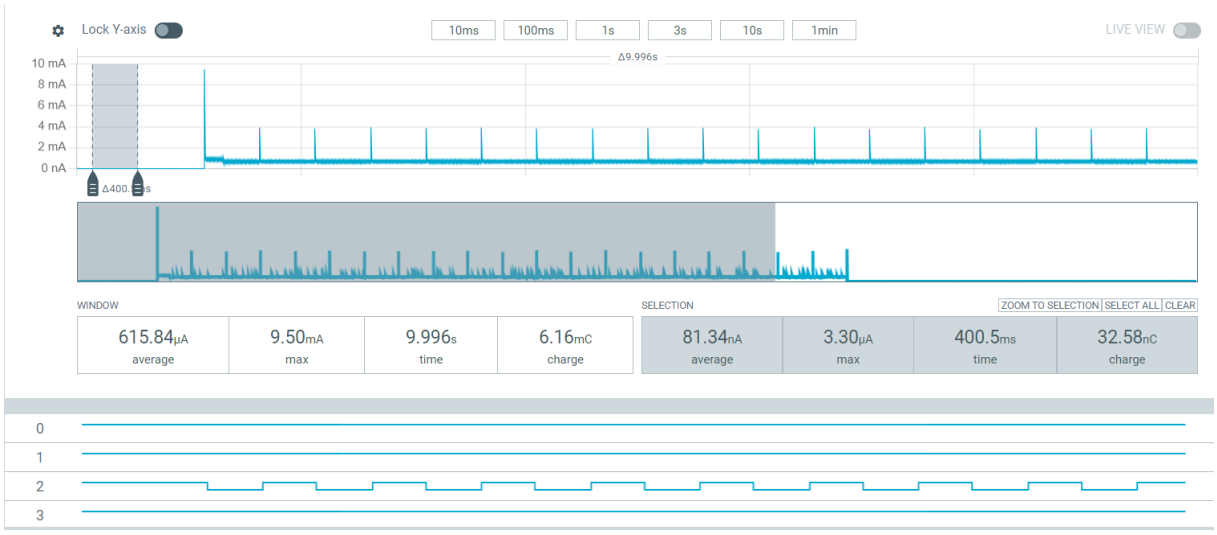


Figura 7: Medición de consumo - Blink v3 bis (modo dormido/System OFF)

- `Beat Hero`:

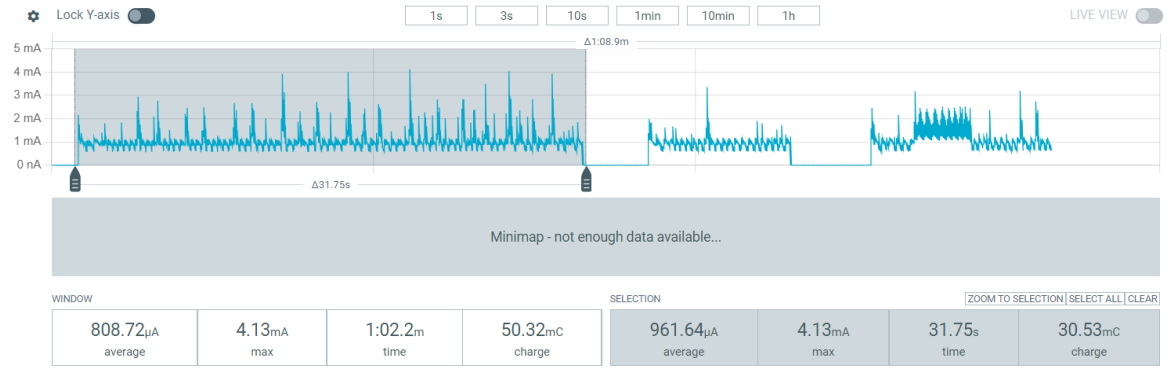


Figura 8: Medición de consumo - Beat Hero durante ejecución del juego

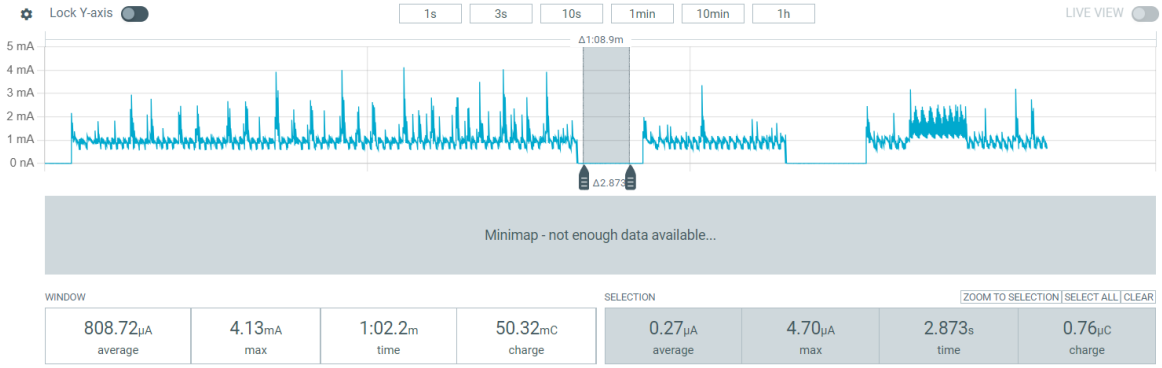


Figura 9: Medición de consumo - Beat Hero con procesador dormido

Para evaluar el comportamiento energético del sistema se han realizado mediciones de consumo sobre la placa nRF52840, diferenciando dos escenarios: ejecución activa del juego y procesador en modo dormido.

Consumo durante la ejecución del juego:

La medición correspondiente a una partida completa (duración: 31,75 s) muestra:

- Consumo medio: 961,64 µA
- Consumo máximo: 4,13 mA
- Carga total: 30,53 mC

El perfil presenta un nivel base cercano a 1 mA con picos periódicos de hasta 4 mA, correspondientes a la atención de interrupciones, ejecución del planificador, gestión de eventos y activación de periféricos. Este comportamiento es coherente con un sistema orientado a eventos donde el procesador se despierta únicamente cuando existe trabajo que realizar.

Consumo en modo dormido (WFI):

Con el procesador en modo de bajo consumo sin eventos pendientes(duración: 2,873 s):

- Consumo medio: 0,27 µA

- Consumo máximo: 4,70 µA
- Carga total: 0,76 µC

El consumo se sitúa en el rango de submicroamperios, una reducción de más de tres órdenes de magnitud respecto a la ejecución activa.
Comparación de resultados:

Estado del sistema	Consumo medio
Juego en ejecución	∼962 µA
Procesador dormido	∼0,27 µA

Cuadro 9: Comparación de consumo energético entre estados

El consumo en reposo es aproximadamente 3500 veces menor que durante la ejecución activa, confirmando que el diseño orientado a eventos cumple los objetivos de eficiencia energética: se elimina la espera activa, el procesador solo se mantiene despierto cuando es necesario, y se maximiza el tiempo en modo de bajo consumo sin perder capacidad de respuesta.

5. Conclusiones

5.1. Grado de Cumplimiento

Objetivo	Estado
Arquitectura por capas	✓ Cumplido
Portabilidad LPC/nRF	✓ Cumplido
Modelo orientado a eventos	✓ Cumplido
Gestión de tiempo 64 bits	✓ Cumplido
Bajo consumo	✓ Cumplido
Watchdog	✓ Cumplido
Juego Beat Hero	✓ Cumplido
Antirrebotes	✓ Cumplido

Cuadro 10: Grado de cumplimiento de objetivos del proyecto

5.2. Conclusiones y qué se ha aprendido

Este proyecto ha permitido alcanzar todos los objetivos planteados y adquirir competencias fundamentales en desarrollo de sistemas empotrados. A lo largo de las ocho sesiones de trabajo, hemos aprendido a diseñar e implementar una arquitectura software modular de cinco capas (HAL, drivers, runtime, servicios y aplicación) que garantiza la portabilidad entre diferentes plataformas ARM, comprendiendo la importancia de la abstracción de hardware y la separación de responsabilidades.

Hemos adquirido experiencia práctica en programación orientada a eventos, implementando un sistema de gestión de eventos con cola FIFO circular y suscripciones por prioridad, lo que nos ha enseñado a manejar la concurrencia mediante secciones críticas y a evitar condiciones de carrera. El desarrollo del sistema de alarmas software nos ha permitido comprender la gestión temporal en sistemas empotrados, incluyendo la extensión de contadores de 32 bits a 64 bits mediante técnicas de software. Además, hemos aprendido técnicas de optimización energética mediante modos de bajo consumo (WFI, System OFF) y hemos implementado mecanismos de robustez

como el watchdog hardware para recuperación ante bloqueos.

La implementación exitosa de los juegos Beat Hero y Bit Counter Strike demuestra la viabilidad del sistema completo y nos ha permitido aplicar todos estos conceptos en un proyecto práctico e integrador, desarrollando competencias en depuración de sistemas empotrados, uso de herramientas como Keil µVision, medición de consumo con Nordic PPK2, y comprensión de las diferencias entre plataformas de hardware.

5.3. Notas de Autoevaluación

Nombre	**NIP**	**Nota**	**Comentarios**
Fernando	897113	8	He aprendido mucho sobre sistemas emportrados y ha sido un placer trabajar por primera vez con mi compañero Guillermo.
Guillermo	896594	8	El llevar a cabo un proyecto como este donde la carga de trabajo ha sido considerablemente superior a lo que estamos acostumbrados ha supuesto un gran reto. Sin embargo, considero que he aprendido mucho en cuanto a organización y coordinación, ya que aunque era la primera vez que trabaja con mi compañero Fernando, hemos logrado cumplir con el objetivo entre los dos.

Cuadro 11: Autoevaluación de los autores del proyecto

Referencias

- NXP Semiconductors. *UM10275: LPC2104/05/06 User Manual* (Rev. 02). NXP B.V., 8 de abril de 2009. Disponible en: https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc2104_2105_2106.pdf
- Nordic Semiconductor. *nRF52840 Product Specification v1.10* (4413 417). Nordic Semiconductor ASA, 4 de julio de 2024. [En línea]. Disponible en: https://docs.nordicsemi.com/bundle/ps_nrf52840/page/keyfeatures_html5.html
- Nordic Semiconductor. *nRF52840 DK User Guide v1.0.0* (4440 050). Nordic Semiconductor ASA, 3 de diciembre de 2020.
- Nordic Semiconductor. *Power Profiler Kit II User Guide*. Nordic Semiconductor ASA, 2020.
- Arm Limited. *DUI 0553B: Cortex-M4 Devices Generic User Guide*. Arm Limited, 3 de agosto de 2011.
- Arm Limited. *100166_0001_00_en: ARM Cortex-M4 Processor Technical Reference Manual* (Revision r0p1). Arm Limited, 23 de febrero de 2015.
- Arm Limited. *DDI 0403E.e: Arm v7-M Architecture Reference Manual*. Arm Limited, 15 de febrero de 2021.
- ARM Limited. *ARM7TDMI Technical Reference Manual*. ARM Limited, 1995.
- Keil. *µVision User's Guide*. ARM Keil, 2024.
- Profesorado de la asignatura "Proyecto Hardware". *Material docente y guías de laboratorio*. Universidad de Zaragoza, 2025.

Anexos

Anexo A: Estadísticas

Estadísticas de la Cola FIFO

El módulo `rt_fifo.c` mantiene estadísticas de eventos que son accesibles públicamente mediante la función `rt_FIFO_estadisticas()`:

- `estadisticas[EVENT_TYPES]`: contador de eventos encolados por tipo
- `total`: total histórico de eventos encolados

Uso:

```
uint32_t total_eventos = rt_FIFO_estadisticas(ev_VOID);
uint32_t eventos_boton = rt_FIFO_estadisticas(ev_BOTON_PULSAR);
```

Métricas de Latencia del Gestor de Eventos

El gestor de eventos `rt_GE.c` calcula métricas de latencia internamente para análisis de rendimiento:

- `tiempo_max_sin_tratar`: máxima latencia entre encolado y procesamiento
- `media_tiempo_sin_tratar`: latencia media acumulada
- `eventos_tratados`: contador total de eventos procesados
- `tiempo_sin_tratar`: acumulador de latencia total

Nota: Estas variables son internas al módulo y no están expuestas mediante una API pública. Solo son accesibles mediante debugger para diagnóstico.

Estadísticas del Juego Beat Hero

El juego Beat Hero (`juego_beat.c`) mantiene estadísticas propias de rendimiento del jugador:

- `puntuacion`: puntuación total acumulada
- `puntuacion_por_compas`: puntuación media por compás
- `puntuacion_por_nivel[4]`: puntuación acumulada por cada nivel
- `tiempo_reaccion_total`: suma de todos los tiempos de reacción
- `tiempo_reaccion_medio_ms`: tiempo de reacción medio
- `tiempo_reaccion_mas_rapido_ms`: mejor tiempo de reacción registrado

Estas estadísticas se actualizan en tiempo real durante la partida y se utilizan para evaluar el rendimiento del jugador.

Anexo B: Juego Beat Hero (juego_beat.c)

```c
#include "juego_beat.h"
#include "drv_leds.h"
#include "svc_alarmas.h"
#include "drv_tiempo.h"
#include "board.h"
#include "drv_consumo.h"
#include "drv_random.h"
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>

// ============================
//   CONSTANTES
// ============================
#define JUEGO_BPM_INICIAL             60
#define JUEGO_COMPASES_MAX            30
#define JUEGO_PUNTOS_MIN              -5
#define JUEGO_VENTANA_PORC            40
#define JUEGO_PRECISION_10            10
#define JUEGO_PRECISION_20            20
#define JUEGO_COMPASES_POR_NIVEL      7
#define JUEGO_TIMEOUT_INACTIVIDAD_MS  10000
#define JUEGO_NIVEL_MAX               4

#define ANIM_PARPADEO_MS      500
#define ANIM_FIN_MS           300

// ------------------- Estado interno -------------------------
typedef enum {
    INICIO = 0,
    MOSTRAR_SECUENCIA = 1,
    ESPERAR_ENTRADA = 2,
        VERIFICANDO_SALIDA = 3,
    FIN_PARTIDA = 4,
    DORMIR = 5
} juego_estado_t;

typedef struct {
    juego_estado_t estado;
    uint8_t compas[3];
    uint8_t nivel;
    uint8_t bpm;
    uint32_t t_compas_ms;
    uint32_t ventana_ms;

    int8_t compases_jugados;

    uint32_t timestamp_inicio;
    uint8_t patron_recibido;
    bool primera_pulsacion;
    uint32_t timestamp_primera_pulsacion;

    uint8_t paso_animacion;
    bool anim_exito;

        //-----Estadísticas-------
        int32_t puntuacion;
        uint8_t puntuacion_por_compas;
        uint8_t puntuacion_por_nivel[4];
        uint32_t tiempo_reaccion_total;
        uint32_t tiempo_reaccion_medio_ms;
```

```
62          uint32_t tiempo_reaccion_mas_rapido_ms;
63  } juego_t;
64
65  static juego_t j;
66
67  // --------------- Helpers -------------------
68
69  static void juego_apagar_todos(void){
70      for (uint8_t i = 1; i <= LEDS_NUMBER; ++i) drv_led_establecer((LED_id_t)i,
            LED_OFF);
71  }
72
73  static void juego_encender_todos(void){
74      for (uint8_t i = 1; i <= LEDS_NUMBER; ++i) drv_led_establecer((LED_id_t)i,
            LED_ON);
75  }
76
77  static void juego_reprogramar(uint32_t retardo_ms){
78      svc_alarma_activar(svc_alarma_codificar(0, retardo_ms, 0), ev_JUEGO_TICK, 0)
            ;
79  }
80
81  static void juego_resetear_inactividad(void) {
82      svc_alarma_activar(svc_alarma_codificar(0, JUEGO_TIMEOUT_INACTIVIDAD_MS, 0),
             ev_INACTIVIDAD, 0);
83  }
84
85  // --------------- Lógica  -------------------
86
87  static uint8_t generar_compas(void){
88          static uint8_t ultimo_generado = 0;
89      uint32_t r;
90      uint8_t resultado = 0;
91
92      do {
93          r = drv_random_generar_rango(4); // Genera 0, 1, 2, 3
94
95          switch (j.nivel) {
96              case 1:
97                  //Nivel 1: SOLO un LED encendido (1 o 2).
98                  //No se permiten vacíos (0) ni dobles (3).
99                  if (r == 0) resultado = 1;
100                 else if (r == 3) resultado = 2;
101                 else resultado = (uint8_t)r;
102                 break;
103
104             case 2:
105                 //Nivel 2: Se permiten compases sin leds (0).
106                 //NO se permiten dobles (3).
107                 if (r == 3) resultado = 0;
108                 else resultado = (uint8_t)r;
109                 break;
110
111             default:
112                 //Nivel 3 y 4: Permitir dos LEDs (3).
113                 resultado = (uint8_t)r;
114                 break;
115         }
116
117         //En nivel 1 forzamos alternancia para que sea visualmente claro.
118         if (j.nivel == 1 && ultimo_generado != 0 && resultado == ultimo_generado
                ) {
119             //Si sale repetido en nivel 1, forzamos el contrario.
```

```
120             resultado = (resultado == 1) ? 2 : 1;
121         }
122
123     } while (resultado == ultimo_generado && j.nivel > 1); //Evitar repetición
        exacta en niveles altos.
124
125     ultimo_generado = resultado;
126     return resultado;
127 }
128
129 static void avanzar_compas(void){
130     j.compas[0] = j.compas[1];
131     j.compas[1] = j.compas[2];
132     j.compas[2] = generar_compas();
133 }
134
135 static void mostrar_compases(void){
136     juego_apagar_todos();
137     //ARRIBA (Leds 1 y 2) .
138     if (j.compas[2] & 0x01) drv_led_establecer(1, LED_ON);
139     if (j.compas[2] & 0x02) drv_led_establecer(2, LED_ON);
140
141     //ABAJO (Leds 3 y 4).
142     if (j.compas[1] & 0x01) drv_led_establecer(3, LED_ON);
143     if (j.compas[1] & 0x02) drv_led_establecer(4, LED_ON);
144 }
145
146 static void procesar_pulsacion(uint32_t boton_id) {
147     uint32_t ahora = drv_tiempo_actual_ms();
148     if (!j.primera_pulsacion) {
149         j.primera_pulsacion = true;
150         j.timestamp_primera_pulsacion = ahora;
151         j.patron_recibido = 0;
152     }
153     if (boton_id == 1) j.patron_recibido |= 0x01;
154     if (boton_id == 2) j.patron_recibido |= 0x02;
155 }
156
157 static void evaluar_ronda(void){
158     if (!j.primera_pulsacion) {// Si no pulsó nada...
159         if (j.compas[0] != 0) j.puntuacion--;
160         return;
161     }
162
163         uint32_t tiempo = j.timestamp_primera_pulsacion - j.timestamp_inicio;
164
165     //Fuera de ventana (>40%): -1 punto.
166     if (tiempo > j.ventana_ms) {
167         j.puntuacion -= 1;
168         return;
169     }
170
171     //Patrón incorrecto: -1 punto.
172     uint8_t esperado = j.compas[0];
173     if (j.patron_recibido != esperado) {
174         j.puntuacion -= 1;
175         return;
176     }
177
178     //Calcular precisión.
179     uint32_t porcentaje = (tiempo * 100) / j.t_compas_ms;
180
181     //Asignación de puntos:
```

```
182     if (porcentaje <= JUEGO_PRECISION_10) {
183         j.puntuacion += 2; //+2 si 10%
184     }
185     else if (porcentaje <= JUEGO_PRECISION_20) {
186         j.puntuacion += 1; //+1 si 20%
187     }
188     //0 si del 20 al 40% (no hacemos nada).
189 
190         // Actualizar estadísticas
191         j.puntuacion_por_nivel[j.nivel-1] += j.puntuacion;
192         if (j.compases_jugados > 0) j.puntuacion_por_compas = j.puntuacion / j.
                compases_jugados;
193         j.tiempo_reaccion_total += tiempo;
194         if (j.compases_jugados > 0) j.tiempo_reaccion_medio_ms = j.
                tiempo_reaccion_total / j.compases_jugados;
195         if (tiempo < j.tiempo_reaccion_mas_rapido_ms) j.
                tiempo_reaccion_mas_rapido_ms = tiempo;
196 
197 }
198 
199 static void actualizar_nivel_y_bpm(void){
200         //Si se han superado todos los compases del nivel se pasa al siguiente
                si aún quedan.
201     if ((j.compases_jugados % JUEGO_COMPASES_POR_NIVEL) == 0 && j.nivel <
            JUEGO_NIVEL_MAX)
202         j.nivel++;
203 
204         if (j.nivel == JUEGO_NIVEL_MAX) j.bpm += 2;   //Aumenta dificultad.
205 
206     j.t_compas_ms = 60000 / j.bpm;
207     j.ventana_ms = (j.t_compas_ms * JUEGO_VENTANA_PORC) / 100;
208 }
209 
210 // ------------------- Interfaz pública -----------------------
211 
212 void juego_beat_iniciar(void){
213     juego_apagar_todos();
214     j.nivel = 1;
215     j.bpm = JUEGO_BPM_INICIAL;
216     j.puntuacion = 0;
217         j.puntuacion_por_compas = 0;
218         for (uint8_t i=0; i<JUEGO_NIVEL_MAX; i++) j.puntuacion_por_nivel[i] = 0;
219       j.tiempo_reaccion_total = 0;
220         j.tiempo_reaccion_medio_ms = 0;
221         j.tiempo_reaccion_mas_rapido_ms = j.ventana_ms; //no puede reaccionar má
                s lento que la ventana porque perdería un punto.
222     j.compases_jugados = -1; //comienza en -1 porque tiene que bajar el primer
            pixel hasta abajo para empezar.
223     j.t_compas_ms = 60000/ j.bpm;
224         j.patron_recibido = 0;
225     j.primera_pulsacion = false;
226     j.timestamp_primera_pulsacion = 0;
227 
228     j.compas[0] = 0;
229     j.compas[1] = 0;
230     j.compas[2] = generar_compas();
231 
232     juego_resetear_inactividad();
233 
234     j.paso_animacion = 0;
235     j.estado = INICIO;
236     juego_encender_todos();
237     juego_reprogramar(ANIM_PARPADEO_MS);
```

```
238 }
239
240 void juego_beat_actualizar(EVENTO_T evento, uint32_t auxData){
241
242       //Si vence el periodo de inactividad se duerme.
243     if (evento == ev_INACTIVIDAD && j.estado != DORMIR) {
244                 juego_apagar_todos();
245                 j.estado = DORMIR;
246                 juego_reprogramar(1);
247         return;
248     }
249         //Si se detecta una pulsación larga se inicia nueva partida.
250     if (evento == ev_PULSACION_LARGA && j.estado != DORMIR) {
251         juego_beat_iniciar();
252         return;
253     }
254
255     switch (j.estado) {
256         case INICIO:
257             if (evento == ev_JUEGO_TICK) {
258                                 //Realizar animación comienzo partida.
259                 j.paso_animacion++;
260                 switch(j.paso_animacion) {
261                     case 1: juego_apagar_todos();   juego_reprogramar(
                            ANIM_PARPADEO_MS); break;
262                     case 2: juego_encender_todos(); juego_reprogramar(
                            ANIM_PARPADEO_MS); break;
263                     case 3: juego_apagar_todos();   juego_reprogramar(200);
                            break;
264                     default:
265                                                 //Cuando acaba la animación se
                                                        pasa a MOSTRAR_SECUENCIA.
266                         j.estado = MOSTRAR_SECUENCIA;
267                         juego_reprogramar(1);
268                         break;
269                 }
270             }
271             break;
272
273         case MOSTRAR_SECUENCIA:
274             if (evento == ev_JUEGO_TICK) {
275                                 //Se muestra la secuencia de leds al jugador y se
                                      pasa a esperar a entrada.
276                 mostrar_compases();
277                 j.timestamp_inicio = drv_tiempo_actual_ms();
278                 j.patron_recibido = 0;
279                 j.primera_pulsacion = false;
280                 juego_reprogramar(j.t_compas_ms);
281                 j.estado = ESPERAR_ENTRADA;
282             }
283             break;
284
285         case ESPERAR_ENTRADA:
286             if (evento == ev_BOTON_PULSAR) {
287                                 //Si llega una pulsación se resetea la
                                        inactividad.
288                 juego_resetear_inactividad();
289                                 //Si se pulsa el botón 3 o 4 significa que se
                                        acaba la partida o se reinicia
290                                 //así que se pasa a VERIFICANDO_SALIDA
291                 if (auxData >= 3) { // Salir
292                                             juego_apagar_todos();
293                     j.estado = VERIFICANDO_SALIDA;
```

```
294                     juego_reprogramar(3100); //más de 3 segundos para que de
                            tiempo a pulsación reinicio
295                     return;
296                 }
297                                 // Si no se procesa la pulsación del botón para
                                        luego ver si ha acertado el jugador o no
298                 if (auxData == 1 || auxData == 2) procesar_pulsacion(auxData);
299
300             } else if (evento == ev_JUEGO_TICK) {
301                                 //Cuando vence la reprogramación con el compás
                                        se calcula la puntuacion y estadísticas.
302                                 j.compases_jugados++;
303                 evaluar_ronda();
304                 avanzar_compas();
305                                 //Actualizar nivel, si procede.
306                             actualizar_nivel_y_bpm();
307
308                             //Fin de partida si llega al máximo de compases o
                                    si hace una mala puntuación.
309                 if (j.compases_jugados >= JUEGO_COMPASES_MAX || j.puntuacion <=
                        JUEGO_PUNTOS_MIN) { //comentar "|| j.puntuacion <=
                        JUEGO_PUNTOS_MIN)" para no perder por puntos y ver apagado
                        por inactividad.
310                     j.estado = FIN_PARTIDA;
311
312                                         if (j.compases_jugados >=
                                                JUEGO_COMPASES_MAX) {
313                         j.anim_exito = true; //Ganar.
314                     } else {
315                         j.anim_exito = false; //Perder.
316                     }
317                                     //Se prepara la animación de fin de
                                            partida.
318                     j.paso_animacion = 0;
319                     juego_reprogramar(1);
320                 } else {
321                                         //Si no ha acabado se pasa a la
                                                siguiente ronda/compás.
322                     j.estado = MOSTRAR_SECUENCIA;
323                     juego_reprogramar(1);
324                 }
325             }
326             break;
327                 case VERIFICANDO_SALIDA:
328                         //while(1); //descomentar para comprobar el whatchdog
329
330             //Si llega el evento de pulsación larga: REINICIAR
331             if (evento == ev_PULSACION_LARGA) {
332                 juego_beat_iniciar();
333             }
334             //Si se acaba el tiempo (Tick de 3100ms) y no llegó pulsación larga:
                     fin de partida.
335             else if (evento == ev_JUEGO_TICK) {
336                 j.estado = FIN_PARTIDA;
337                 j.paso_animacion = 0;
338                 juego_reprogramar(1);
339             }
340             break;
341
342         case FIN_PARTIDA:
343             if (evento == ev_JUEGO_TICK) {
344                                 // Mostrar animación
345                 j.paso_animacion++;
```

```
346                 if (j.paso_animacion < 8) {
347                     if (j.paso_animacion % 2 != 0) {
348                         //Distinguir animaciones.
349                         if (j.anim_exito) {
350                             juego_encender_todos(); //VICTORIA, todo luz.
351                         } else {
352                             //DERROTA:Cruz Leds 1 y 4
353                             juego_apagar_todos();
354                             drv_led_establecer(1, LED_ON);
355                             drv_led_establecer(4, LED_ON);
356                         }
357                     }
358                     else juego_apagar_todos();
359 
360                     juego_reprogramar(ANIM_FIN_MS);
361                 } else {
362                                         //Cuando se acaba la animación pasa a
                                            dormirse.
363                     j.estado=DORMIR;
364                     juego_reprogramar(1);
365                 }
366             }
367             break;
368 
369         case DORMIR:
370                         juego_apagar_todos();
371             drv_consumo_dormir();
372             break;
373     }
374 }
```

Anexo C: Gestor de Eventos (rt_GE.c)

```c
#include "rt_fifo.h"
#include "drv_tiempo.h"
#include "drv_monitor.h"
#include "drv_consumo.h"
#include "rt_GE.h"
#include "svc_GE.h"
#include "rt_evento_t.h"
#include "svc_alarmas.h"
#include "board.h"
#include "hal_ext_int.h"
#include "drv_WDT.h"


// Estructura para almacenar las suscripciones
uint32_t rt_monitor_overflow;
GE_suscripcion_t ge_tabla[EVENT_TYPES];
Tiempo_us_t tiempo_max_sin_tratar;
uint32_t eventos_tratados;
Tiempo_us_t tiempo_sin_tratar;
Tiempo_us_t media_tiempo_sin_tratar;



void rt_GE_iniciar(uint32_t monitor_overflow){
  tiempo_max_sin_tratar = 0;
    eventos_tratados = 0;
    media_tiempo_sin_tratar = 0;
    tiempo_sin_tratar = 0;
    rt_monitor_overflow = monitor_overflow;
    for (uint8_t i = 0; i < EVENT_TYPES; i++) {
    ge_tabla[i].num_suscritos = 0;
    for (uint8_t j = 0; j < rt_GE_MAX_SUSCRITOS; j++) {
            ge_tabla[i].lista_callbacks[j] = NULL;
    }
  }

}


void rt_GE_lanzador(){

    svc_alarma_activar(svc_alarma_codificar(0,20*1000,0), ev_INACTIVIDAD, 0); // alarma por inactividad a los 20 segundos sin hacer nada.
    EVENTO_T id_ev;
  uint32_t aux;
  Tiempo_us_t ts;
    while(1){

        drv_WDT_alimentar(); //vamos alimentando al watchdog

        //si hay eventos en FIFO
        if(rt_FIFO_extraer(&id_ev, &aux, &ts) != (uint8_t)-1){
                //Actualizar  e s t a d sticas
                Tiempo_us_t ahora = drv_tiempo_actual_us();
                Tiempo_us_t delta = ahora - ts;
                eventos_tratados++;
                tiempo_sin_tratar += delta;
                media_tiempo_sin_tratar = tiempo_sin_tratar / eventos_tratados;
                if (delta > tiempo_max_sin_tratar) tiempo_max_sin_tratar = delta;
                //recorer fila del evento llamando funciones
```

```
60                  if ((uint8_t)id_ev < EVENT_TYPES) {
61                 for (uint8_t i = 0; i < ge_tabla[(uint8_t)id_ev].num_suscritos;
                       i++) {
62                                 if (ge_tabla[(uint8_t)id_ev].lista_callbacks[i]
                                       != NULL) {
63                                     ge_tabla[(uint8_t)id_ev].lista_callbacks[i](
                                           id_ev, aux);
64                                 }
65                             }
66                  }
67         } else{
68                 //sino esperar
69                 drv_consumo_esperar();
70         }
71     }
72
73 }
74
75 //helper
76 static void preparar_despertar_inactividad(void){
77 #if BUTTONS_NUMBER > 0
78     for (uint8_t i = 1; i <= BUTTONS_NUMBER; ++i) {
79         hal_ext_int_habilitar(i);   // SENSE=LOW en TODOS los botones
80     }
81 #endif
82 }
83
84
85 static uint32_t contador_alarma = 0;
86
87 //Reprograma
88 void rt_GE_actualizar(EVENTO_T ID_EVENTO, uint32_t auxData){
89
90     switch((uint8_t)ID_EVENTO){
91         case ev_INACTIVIDAD:
92             preparar_despertar_inactividad();
93             drv_consumo_dormir();
94             break;
95
96         case ev_BOTON_PULSAR:
97             contador_alarma++;
98             svc_alarma_activar(
99                 svc_alarma_codificar(0, 20000, 0),
100                ev_INACTIVIDAD,
101                contador_alarma
102            );
103            break;
104
105        default:
106            break;
107    }
108 }
```

Anexo D: Cola FIFO (rt_fifo.c)

```c
#include "rt_fifo.h"
#include "drv_tiempo.h"
#include "drv_monitor.h"
#include "rt_sc.h"

// P a r metros  de la cola.
#ifndef RT_FIFO_CAPACITY
#define RT_FIFO_CAPACITY 32
#endif


//Estado interno.
typedef struct {
    EVENTO eventos_encolados[RT_FIFO_CAPACITY];
    volatile uint32_t head;                                    // p r xima
        p o s i c i n  libre para escribir
    volatile uint32_t tail;                                    // p r xima
        p o s i c i n  con dato para leer (main)
    uint32_t estadisticas[EVENT_TYPES];         //contadores por tipo
    uint32_t total;                                                 //total
        encolados  h i s t rico
    uint32_t mon_overflow;                                     //monitor a marcar
        en overflow (0 => ninguno)
    volatile uint32_t contador;                              //contador actual de
        elementos encolados del 1 al RT_FIFO_CAPACITY, 0 si cola  v a c a
} fifo_t;


static fifo_t cola;


void rt_FIFO_inicializar(uint32_t monitor_overflow) {
    cola.head = 0;
    cola.tail = 0;
      cola.contador = 0;
    for (uint32_t i = 0; i < EVENT_TYPES; ++i) cola.estadisticas[i] = 0;
    cola.total = 0;
    cola.mon_overflow = monitor_overflow;
}


void rt_FIFO_encolar(uint32_t ID_evento, uint32_t auxData) {

    //entrar en  s e c c i n  c r tica
        rt_sc_estado_t sc = rt_sc_entrar();

        if (cola.contador == RT_FIFO_CAPACITY) {
        if (cola.mon_overflow) drv_monitor_marcar(cola.mon_overflow);
        while(true){ /* overflow -> bucle infinito para  d i a g n stico  */ }
    }

    uint32_t pos = cola.head;
    cola.eventos_encolados[pos].ID_EVENTO = (EVENTO_T)ID_evento;
    cola.eventos_encolados[pos].auxData = auxData;
    cola.eventos_encolados[pos].TS = drv_tiempo_actual_us();


    cola.head = (pos + 1) % RT_FIFO_CAPACITY;
        cola.contador++;

    //  E s t a d sticas
```

```
57      if (ID_evento < EVENT_TYPES) cola.estadisticas[ID_evento]++;
58      cola.total++;
59
60          //salir de la seccin crtica
61      rt_sc_salir(sc);
62  }
63
64
65  uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t *auxData, Tiempo_us_t *TS)
        {
66
67          //entrar en seccin crtica
68          rt_sc_estado_t sc = rt_sc_entrar();
69
70          uint32_t t = cola.tail;
71      if (cola.contador == 0){
72              rt_sc_salir(sc);
73              return (uint8_t)-1;
74          }
75
76      if (ID_evento) *ID_evento = cola.eventos_encolados[t].ID_EVENTO;
77      if (auxData) *auxData = cola.eventos_encolados[t].auxData;
78      if (TS) *TS = cola.eventos_encolados[t].TS;
79
80      cola.tail = (t + 1) % RT_FIFO_CAPACITY;
81          cola.contador--;
82
83      //devolver cantidad de eventos an sin procesar tras la extraccin.
84      uint32_t pendientes = cola.contador;
85      if (pendientes > 254) pendientes = 254;
86
87          //salir de la seccin crtica
88          rt_sc_salir(sc);
89
90          return (uint8_t)pendientes;
91  }
92
93
94  uint32_t rt_FIFO_estadisticas(EVENTO_T ID_evento) {
95      if (ID_evento == ev_VOID) return cola.total;
96      if ((uint32_t)ID_evento < EVENT_TYPES) return cola.estadisticas[(uint32_t)
            ID_evento];
97      return 0;
98  }
```

Anexo E: Random LPC (hal_random_lpc.c)

```c
#include "hal_random.h"
#include <LPC210x.H>

/*
 * Necesitamos acceso a algn contador de tiempo para "simular" entropa
 * Se asume que hal_tiempo ha iniciado el Timer 1.
 * Si no, devolveremos un valor fijo y dependeremos de la semilla manual del main.
 */

void hal_random_iniciar(void) {
    //En el LPC2105 no hay hardware especfico de generador aleatorio a iniciar.
}

uint8_t hal_random_generar_u8(void) {

        //Usa el valor actual del Timer 1 (T1TC) como fuente de "ruido".
    uint32_t entropia_simulada = T1TC;

    //Cogemos el byte bajo.
    return (uint8_t)(entropia_simulada & 0xFF);
}
```

Anexo F: Random nRF (hal_random_nrf.c)

```c
#include "hal_random.h"
#include <nrf.h>

void hal_random_iniciar(void) {
    /*
         * Configurar el generador de nmeros aleatorios.
     * Activar correccin de sesgo.
     */
        NRF_RNG->CONFIG = RNG_CONFIG_DERCEN_Enabled;

    //Arrancar la tarea.
    NRF_RNG->TASKS_START = 1;
}

uint8_t hal_random_generar_u8(void) {
    //Limpiar evento previo por seguridad.
    NRF_RNG->EVENTS_VALRDY = 0;

    // Esperar a que haya un valor listo. El tiempo es indeterminado pero r pido para 1 byte.
    while (NRF_RNG->EVENTS_VALRDY == 0);

    //Limpiar evento para la prxima lectura
    NRF_RNG->EVENTS_VALRDY = 0;

    //Leer el valor del registro VALUE.
    return (uint8_t)(NRF_RNG->VALUE);
}
```

Anexo G: Secciones Criticas LPC (rt_sc_lpc.c)

```c
#include <LPC210x.H>
#include <stdint.h>
#include "rt_sc.h"


rt_sc_estado_t rt_sc_entrar(void){
    __disable_irq();

    //Para esta práctica no usamos el valor previo, asi que devolvemos 0.
    return 0;
}

void rt_sc_salir(rt_sc_estado_t estado_prev){
    (void)estado_prev;   //No lo usamos en esta versión simple.
    __enable_irq();
}
```

Anexo H: Secciones Criticas nRF (rt_sc_nrf.c)

```c
#include <stdint.h>
#include <nrf.h>
#include "rt_sc.h"

/*
 * - PRIMASK = 1  => IRQs deshabilitadas
 * - PRIMASK = 0  => IRQs habilitadas
 */

rt_sc_estado_t rt_sc_entrar(void)
{
    //Guardamos el estado previo.
    uint32_t primask = __get_PRIMASK();
    //Deshabilitamos interrupciones.
    __disable_irq();
    return primask;
}

void rt_sc_salir(rt_sc_estado_t estado_prev)
{
    //Restauramos exactamente el estado previo.
    __set_PRIMASK(estado_prev);
}
```

Anexo I: Driver de Botones (drv_botones.c)

```c
#include "drv_botones.h"
#include "svc_GE.h"
#include "board.h"
#include "hal_ext_int.h"
#include "svc_alarmas.h"
#include "drv_tiempo.h"

#define RETARDO_REBOTE_MS 50    // Tiempo rebote presión
#define RETARDO_SOLTAR_MS 50    // Tiempo rebote al soltar
#define RETARDO_ENCUESTA_MS 20  // Tiempo encuesta periódica (revisar si sigue pulsado)
#define PULSACION_LARGA 3000    // 3 segundos como pulsación larga

typedef enum {
    e_esperando,
    e_salida,
    e_rebotes,
    e_muestreo,
} boton_estado_t;

typedef struct {
    boton_estado_t estado;
    uint8_t pin;
      uint32_t inicio_pulsacion; // Timestamp de cuando se ha pulsado el botón
        bool larga_enviada;
} boton_t;

static boton_t botones[BUTTONS_NUMBER] = {
    {e_salida, BUTTON_1},
    {e_salida, BUTTON_2},
    {e_salida, BUTTON_3}
        #if BUTTON_4
        ,
    {e_salida, BUTTON_4}
        #endif
};

//Callback a nivel superior.
static drv_botones_callback_t drv_botones_callback;

void drv_botones_iniciar(drv_botones_callback_t botones_callback, EVENTO_T ev1, EVENTO_T ev2){
    drv_botones_callback = botones_callback;
    hal_ext_int_registrar_callback(drv_botones_pulsar_RSI);
    hal_ext_int_iniciar();

    //Estado inicial: esperando pulsaciones.
    for (uint8_t i = 0; i < BUTTONS_NUMBER; ++i) {
        botones[i].estado = e_esperando;
    }

    //Habilitar SENSE (LOW) en todos los botones.
    for (uint8_t i = 1; i <= BUTTONS_NUMBER; ++i) {
        hal_ext_int_habilitar(i);
    }

    svc_GE_suscribir(ev1, 1, drv_botones_actualizar);
    if (ev2 != ev_VOID) {
        svc_GE_suscribir(ev2, 1, drv_botones_actualizar);
    }
}
```

```
60
61
62  //Llamada desde la RSI del HAL al detectar una pulsación.
63  void drv_botones_pulsar_RSI(uint8_t id_boton){
64
65      if (id_boton == 0 || id_boton > BUTTONS_NUMBER) return;
66
67      //Solo procesar si estamos esperando (evita crear múltiples alarmas)
68      if (botones[id_boton - 1].estado != e_esperando) {
69          return;  //Ya estamos procesando este botón, ignorar
70      }
71
72      // Se pulsa el botón y se guarda su timestamp
73      botones[id_boton - 1].inicio_pulsacion = drv_tiempo_actual_ms();
74
75      //Reseteamos la "flag" de pulsación larga.
76      botones[id_boton - 1].larga_enviada = false;
77
78
79      //NOTA: Para LPC con interrupciones por nivel, la ISR ya deshabilita la
            interrupción.
80      //Para NRF con interrupciones por flanco, necesitamos deshabilitarla aquí.
81      //Como esta función se llama desde la ISR, la interrupción ya está
            deshabilitada en LPC.
82    hal_ext_int_deshabilitar(id_boton);
83
84      svc_alarma_activar(svc_alarma_codificar(0, RETARDO_REBOTE_MS, 0),
            ev_BOTON_RETARDO, id_boton);
85
86    botones[id_boton - 1].estado = e_rebotes;
87
88  }
89
90  void drv_botones_actualizar(EVENTO_T evento, uint32_t auxiliar){
91          uint32_t ahora; // variable para medir el tiempo pulsado del botón
92      uint8_t id_boton = (uint8_t)auxiliar;
93
94      //Botón fuera de rango? return...
95      if (id_boton == 0 || id_boton > BUTTONS_NUMBER) {
96          return;
97      }
98
99      switch (evento) {
100
101         //Evento interno: timeout de las alarmas de rebote/soltar.
102         case ev_BOTON_RETARDO:
103             switch (botones[id_boton - 1].estado){
104
105                 //Acabamos de detectar RSI y estamos en antirrebotes.
106                 case e_rebotes:
107                     if (hal_ext_int_obtener_estado(id_boton)) {
108                                     //Sigue pulsado después del retardo: pulsaci
                                        ón válida.
109                         botones[id_boton - 1].estado = e_muestreo;
110
111                         if (drv_botones_callback) {
112                             drv_botones_callback(ev_BOTON_PULSAR, id_boton);
113                         }
114
115                         //Programar alarma para comprobar periódicamente.
116                         svc_alarma_activar(svc_alarma_codificar(0,
                                RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton)
                                ;
```

```
117                     } else {
118                                                 //Era ruido/rebote: volver a
                                                        esperar y re-habilitar.
119                         botones[id_boton - 1].estado = e_esperando;
120                         hal_ext_int_habilitar(id_boton);
121                     }
122                     break;
123
124                 //Estamos esperando a que se suelte el botón.
125                 case e_muestreo:
126                     ahora = drv_tiempo_actual_ms();
127
128                     //Comprobamos si se sigue pulsando físicamente.
129                     if(hal_ext_int_obtener_estado(id_boton)) {
130
131                         //Calculamos cuánto tiempo lleva pulsado.
132                         uint32_t duracion = ahora - botones[id_boton - 1].
                                inicio_pulsacion;
133
134                         //Si lleva más de 3 segundos Y todavía no hemos avisado
                                Y que venga del botón 3 o 4 (auxData >= 3).
135                         if (duracion >= PULSACION_LARGA && !botones[id_boton -
                                1].larga_enviada && id_boton >= 3) {
136
137                             //Enviamos el evento al juego.
138                             if (drv_botones_callback) {
139                                 drv_botones_callback(ev_PULSACION_LARGA,
                                        id_boton);
140                             }
141
142                             //Marcamos que ya lo enviamos para no spamear.
143                             botones[id_boton - 1].larga_enviada = true;
144                         }
145
146                         //Seguimos muestreando periódicamente mientras siga
                                pulsado.
147                         svc_alarma_activar(
148                             svc_alarma_codificar(0, RETARDO_ENCUESTA_MS, 0),
149                             ev_BOTON_RETARDO,
150                             id_boton
151                         );
152
153                     } else {
154                         //Si ya no está pulsado: pasar a salida.
155                         botones[id_boton - 1].estado = e_salida;
156
157                         //Retardo de rebote al soltars.
158                         svc_alarma_activar(svc_alarma_codificar(0,
                                RETARDO_SOLTAR_MS, 0), ev_BOTON_RETARDO, id_boton);
159                     }
160                     break;
161
162                 //Tras soltar, esperamos a que se estabilice en reposo antes de
                        rearmar SENSE.
163                 case e_salida:
164                     if (!hal_ext_int_obtener_estado(id_boton)) {
165                                                 //listo para nueva pulsación.
166                         botones[id_boton - 1].estado = e_esperando;
167                         hal_ext_int_habilitar(id_boton);
168                     } else {
169                         //Si sigue pulsado, volvemos a muestreo.
170                         botones[id_boton - 1].estado = e_muestreo;
```

```
171                         svc_alarma_activar(svc_alarma_codificar(0,
                                RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton)
                                ;
172                     }
173                     break;
174
175                 case e_esperando:
176                 default:
177
178                     break;
179             }
180             break;
181
182         case ev_BOTON_PULSAR:
183             break;
184
185         default:
186             break;
187     }
188 }
```

Anexo J: Servicio de Alarmas (svc_alarmas.c)

```c
#include "svc_alarmas.h"
#include "drv_monitor.h"
#include "rt_sc.h"

//Estructura  interna
typedef struct{
    bool activa;
    svc_alarma_flags_t flags;       //Codificación de flags de activación.
    uint32_t restante_ms;           //Cuenta atrás en ms.
    EVENTO_T id_evento;             //Evento a encolar al vencer.
    uint32_t aux;                   //auxData que viajará con el evento.
} alarma_t;

typedef struct{
    alarma_t alarmas[svc_ALARMAS_MAX];                      //Todas las alarmas del sistema.
    MONITOR_id_t monitor_overflow;                      //Monitor que se marca cuando no quedan alarmas.
    void (*cb_encolar)(uint32_t id_ev, uint32_t aux);       //Función de encolar
    EVENTO_T ev_tick;                                   //Evento de tick que dispara la actualización.
} sistema;

static sistema s;

//Funciones auxiliares (internas).
static inline bool es_periodica(svc_alarma_flags_t f) {
    return (f & SVC_ALARMA_FLAG_PERIODICA) != 0;
}

static inline uint32_t periodo_ms(svc_alarma_flags_t f) {
    return (f & SVC_ALARMA_RETARDO_MASK_MS);
}

static int buscar_por_evento(EVENTO_T ev, uint32_t aux)
{
    for (int i = 0; i < (int)svc_ALARMAS_MAX; ++i) {
        if (!s.alarmas[i].activa) continue;
        if (s.alarmas[i].id_evento != ev) continue;

        // Para rebotes de botones, distinguimos también por aux (id de botón)
        if (ev == ev_BOTON_RETARDO) {
            if (s.alarmas[i].aux == aux) return i;
        } else {
            // para el resto de eventos mantenemos la semántica original
            return i;
        }
    }
    return -1;
}


static int buscar_libre(void) {
    for (int i = 0; i < (int)svc_ALARMAS_MAX; ++i) {
        if (!s.alarmas[i].activa) return i;
    }
    return -1;
}

static void disparar(const alarma_t* a) {
    if (s.cb_encolar) {
```

```
59          s.cb_encolar((uint32_t)a->id_evento, a->aux);
60      }
61  }
62
63  //Operaciones públicas
64  void svc_alarma_iniciar(uint32_t monitor_overflow, void (*cb_encolar)(uint32_t
        id_ev, uint32_t aux), EVENTO_T ev_a_notificar){
65
66      for (int i = 0; i < (int)svc_ALARMAS_MAX; ++i) {
67          s.alarmas[i].activa = false;
68                  s.alarmas[i].flags = 0;
69                  s.alarmas[i].restante_ms = 0;
70                  s.alarmas[i].id_evento = ev_VOID;
71                  s.alarmas[i].aux = 0;
72      }
73
74      s.monitor_overflow = monitor_overflow;
75      s.cb_encolar = cb_encolar;
76      s.ev_tick = ev_a_notificar;
77  }
78
79  void svc_alarma_activar(svc_alarma_flags_t flags, EVENTO_T id_evento, uint32_t
        auxData){
80      rt_sc_estado_t sc = rt_sc_entrar();
81      //Buscamos si la alarma del evento existe y está actuiva
82          int id_alarma = buscar_por_evento(id_evento, auxData);
83
84
85
86      //Si retardo_ms == 0 => CANCELA la alarma (si existe).
87      uint32_t retardo = periodo_ms(flags);
88      if (retardo == 0) {
89          if (id_alarma >= 0) {
90              s.alarmas[id_alarma].activa = false;
91          }
92          return;
93      }
94
95      //Si no vamos a reprogramra, buscamos el siguiente espacio de alarma libre.
96      if (id_alarma < 0) id_alarma = buscar_libre();
97
98      //Si no hay libres levantamos el monitor.
99      if (id_alarma < 0) {
100         if (s.monitor_overflow) drv_monitor_marcar(s.monitor_overflow);
101         while(true){ /* Todas las alarmas ocupadas */ }
102     }
103
104     //Si repogramamos o creamos la alarma, rellenamos con los nuevos datos.
105     s.alarmas[id_alarma].activa = true;
106     s.alarmas[id_alarma].flags = flags;
107     s.alarmas[id_alarma].restante_ms = retardo;
108     s.alarmas[id_alarma].id_evento = id_evento;
109     s.alarmas[id_alarma].aux = auxData;
110         rt_sc_salir(sc);
111 }
112
113 void svc_alarma_actualizar(EVENTO_T id_evento, uint32_t auxData){
114
115     if (id_evento != s.ev_tick) return;
116
117     //tomamos auxData como ms transcurridos desde el último tick (si viniera a
            0, usamos al menos 1).
118     uint32_t transcurrido = (auxData == 0) ? 1 : auxData;
```

```
119
120         rt_sc_estado_t sc = rt_sc_entrar();
121
122     for(int i = 0; i < (int)svc_ALARMAS_MAX; ++i){
123
124         if(!s.alarmas[i].activa) continue;
125
126         if(transcurrido >= s.alarmas[i].restante_ms){
127             // VENCE
128             disparar(&s.alarmas[i]);
129
130             if (es_periodica(s.alarmas[i].flags)) {
131                 //Reprogramación periódica.
132                 uint32_t periodo = periodo_ms(s.alarmas[i].flags);
133                 uint32_t overshoot = transcurrido - s.alarmas[i].restante_ms;
134
135                 if (periodo == 0) periodo = 1;
136                 // Si el overshoot es múltiplo del periodo, disparamos una vez y
                        dejamos periodo entero
137                 s.alarmas[i].restante_ms = periodo - (overshoot % periodo);
138                 if (s.alarmas[i].restante_ms == 0) s.alarmas[i].restante_ms =
                        periodo;
139             } else {
140                 //Esporódica: se desactiva.
141                 s.alarmas[i].activa = false;
142             }
143         }else{
144             // Aún no vence: decrementa
145             s.alarmas[i].restante_ms -= transcurrido;
146         }
147     }
148
149         rt_sc_salir(sc);
150 }
```