Final Report

Hardware Project 2025

Beat Hero

Authors:

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

University of Zaragoza

School of Engineering and Architecture

B.Sc. in Computer Engineering

Hardware Project

December 2025

Contents

List of Figures

List of Tables

Executive Summary

This document presents the technical report for the Beat Hero project, developed within the course “Hardware Project” of the B.Sc. in Computer Engineering. The project consists of the development of an event-driven embedded system implementing a complete, fully functional rhythm game.

The system was developed following a layered model that ensures portability across two different ARM platforms: LPC2105 (ARM7TDMI) and nRF52840 DK (ARM Cortex-M4F). The project evolved through eight laboratory sessions, from a basic LED blink to a complete interactive game with advanced event handling, software alarms, low-power operation, and a hardware watchdog.

The main achieved objectives include: the development of a modular and portable software architecture; the implementation of an event-driven programming model; efficient time management using hardware timers extended to 64-bit; energy consumption minimization through low-power modes (WFI, System OFF); and improved robustness via watchdog usage and race-condition handling (critical sections).

An incremental development methodology was applied, building the system in layers: HAL (Hardware Abstraction Layer), Drivers, Runtime, Services, and Application. Development was carried out using Keil µVision 5, with separate configurations for debugging (O0) and production (O3). Protection mechanisms based on critical sections were implemented to prevent race conditions, and GPIO monitors were used to correlate software state with oscilloscope measurements.

The final system successfully implements Beat Hero, a rhythm game in which the player must press buttons corresponding to LED patterns according to an increasingly fast tempo. The game includes: a timing-accuracy scoring system, three difficulty levels, and complete state management. All development sessions were completed satisfactorily, implementing event management via a FIFO queue (designed to prevent race conditions), a software alarm system (periodic and one-shot), low-power modes with inactivity wake-up, and a hardware watchdog. Current consumption measurements performed with Nordic PPK2 demonstrate the energy efficiency of the system. The system is fully portable across the two target platforms.

All stated objectives were met. The system implements a robust five-layer architecture ensuring portability, an efficient event-driven model, 64-bit time management, low-power operation, and a watchdog mechanism for recovery from lockups. The Beat Hero game operates correctly and demonstrates the feasibility of the complete system. This project enabled the acquisition of extensive knowledge in embedded systems development, concurrency management in embedded environments, energy optimization, and portable software architecture design.

1. Introduction

1.1. Project Context

This project belongs to the course "Hardware Project" and addresses the development of an event-driven embedded system following a layered model that guarantees portability across different hardware platforms.

The system evolves through eight laboratory sessions, starting from a basic LED blink (blink v1) based on busy-waiting and culminating in a complete interactive game (Beat Hero) featuring event handling, software alarms, low-power operation, and a hardware watchdog.

1.2. Target Platforms

The system was developed for two ARM-based platforms:

Feature	**LPC2105**	**nRF52840 DK**
Architecture	ARM7TDMI	ARM Cortex-M4F
Clock frequency	15 MHz (PCLK)	16 MHz (Timer)
Available LEDs	4	4
Buttons	3 (EINT0-2)	4 (GPIOTE)
Timers	Timer0, Timer1	TIMER0, TIMER1
Low-power modes	Idle, Power-Down	WFI, System OFF

Table 1: Platform feature comparison: LPC2105 vs. nRF52840 DK

1.3. Functional Scope

The final system implements:

- Beat Hero: a rhythm game with LED patterns, progressive difficulty levels, and a scoring system.
- Bit Counter Strike: a reaction-based LED-button game with increasing difficulty.
- Complete event management with a FIFO queue and priority-based subscriptions.
- Software alarm system (periodic and one-shot).
- Low-power modes with wake-up on inactivity.
- Hardware watchdog for recovery from lockups.

2. Objectives

The primary objectives of this project are:

- Develop a modular and portable software architecture across different ARM microcontrollers.
- Implement an event-driven programming model that decouples application logic from hardware.
- Manage time efficiently using hardware timers extended to 64-bit.

- Minimize energy consumption through low-power modes (WFI, System OFF).
- Ensure system robustness via a watchdog and proper handling of race conditions.
- Implement a complete game (Beat Hero) to demonstrate overall system functionality.

3. Methodology

The methodology followed in this project is based on incremental and modular development, building the system in layers that ensure strict separation between hardware and software. The architecture, tools, and processes are detailed below.

3.1. Overall System Architecture

3.1.1. Layered View

The architecture follows a strict layered model that guarantees portability:

APPLICATION / GAME juego_beat.c, juego_counter.c	
SERVICES (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

Table 2: Layered architecture of the embedded system

3.1.2. Layer Description

HAL (Hardware Abstraction Layer):

- Defines generic interfaces in shared headers (`hal_*.h`).
- Platform-specific implementations in `src_lpc/` and `src_nrf/`.
- Abstracts peripheral registers, interrupt vectors, and pin configuration.

Drivers (drv_):

- Provide high-level services independent of the underlying hardware.
- Use exclusively the HAL API.
- Examples: `drv_leds_iniciar()`, `drv_tiempo_actual_us()`, `drv_consumo_esperar()`.

Runtime (rt_):

- Execution infrastructure: event queue, critical sections.
- `rt_fifo.c`: FIFO event queue with an integrated critical section.
- `rt_GE.c`: main loop of the event manager.
- `rt_sc.c`: platform-specific critical section implementation.

Services (svc_):

- Reusable functionality built on top of the runtime.
- `svc_alarmas.c`: periodic/one-shot software alarms.
- `svc_GE.c`: event subscription system with priority.

Application:

- Game-specific logic.
- State machines reacting to events.

3.1.3. Rationale for the Event-Driven Model

The event-driven model offers significant advantages for embedded systems:

1. **Decoupling:** Event producers (ISRs, timers) do not know the consumers.
2. **Low power:** The CPU can enter a low-power state when the queue is empty.
3. **Simplified concurrency:** A single dispatch loop avoids the complexity of multiple threads.
4. **Extensibility:** New modules can subscribe to existing events without modifying prior code.

3.1.4. Hardware/Software Separation

Separation is achieved through:

- **Generic headers** (`board.h`) defining `LEDS_NUMBER`, `BUTTONS_LIST`, etc.
- **Conditional compilation** based on the target platform.
- **Platform-specific board headers** (`board_lpc.h`, `board_nrf52840dk.h`).

3.2. Development Environment

3.2.1. Keil µVision

The project uses Keil µVision 5 as the development environment:

- **Workspace:** `blink.uvmpw` with separate projects for LPC and nRF.
- `lpc/keil/` **folder:** LPC2105 project (simulated).
- `nrf/keil/` **folder:** nRF52840 DK project (physical board).

3.2.2. Build Configurations

Two build configurations are maintained:

Configuration	Optimization	Use
Debug	`-O0`	Development, step-by-step debugging
Release	`-O3`	Performance and power measurements

Table 3: Build configurations in Keil µVision

The macro `SESION` controls which program version is built (1–8).

3.3. Time Management

3.3.1. 64-bit Monotonic Clock

Microsecond-level resolution requires 64-bit counters. Since hardware counters are 32-bit, a software extension is implemented.

LPC2105 (`hal_tiempo_lpc.c`):

```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`):

```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. Resolution and Range

Platform	Ticks/µs	32-bit overflow period	64-bit range
LPC2105	15	∼286 s	∼39,000 years
nRF52840	16	∼268 s	∼36,000 years

Table 4: Resolution and range of the 64-bit monotonic clock

3.3.3. Tick-to-Time Conversion

The `drv_tiempo.c` driver abstracts the conversion:

```c
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. Periodic Timing

The periodic timer generates interrupts at regular intervals to feed the alarm subsystem:

```c
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. Peripheral Management

3.4.1. LEDs

The `drv_leds.c` driver provides an idempotent API:

- `drv_leds_iniciar()`: configures pins as outputs and turns off all LEDs.
- `drv_led_establecer(id, estado)`: turns a specific LED on/off.
- `drv_led_conmutar(id)`: toggles the current state.
- `drv_led_estado(id, *out)`: queries the logical state.

Polarity abstraction:

- LPC2105: active-high LEDs (`LEDS_ACTIVE_STATE = 1`)
- nRF52840: active-low LEDs (`LEDS_ACTIVE_STATE = 0`)

```c
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. Buttons and External Interrupts

Button management is implemented in two layers:
HAL (`hal_ext_int_*.c`):

- Configures pins as inputs with pull-up.
- Registers the ISR for press detection.
- Provides `hal_ext_int_habilitar/deshabilitar` for fine-grained control.

Driver (`drv_botones.c`):

- Implements a state machine for debouncing.
- Detects short and long presses (≥3 s).
- Enqueues `ev_BOTON_PULSAR` and `ev_PULSACION_LARGA` events.

The button driver state machine is shown below:

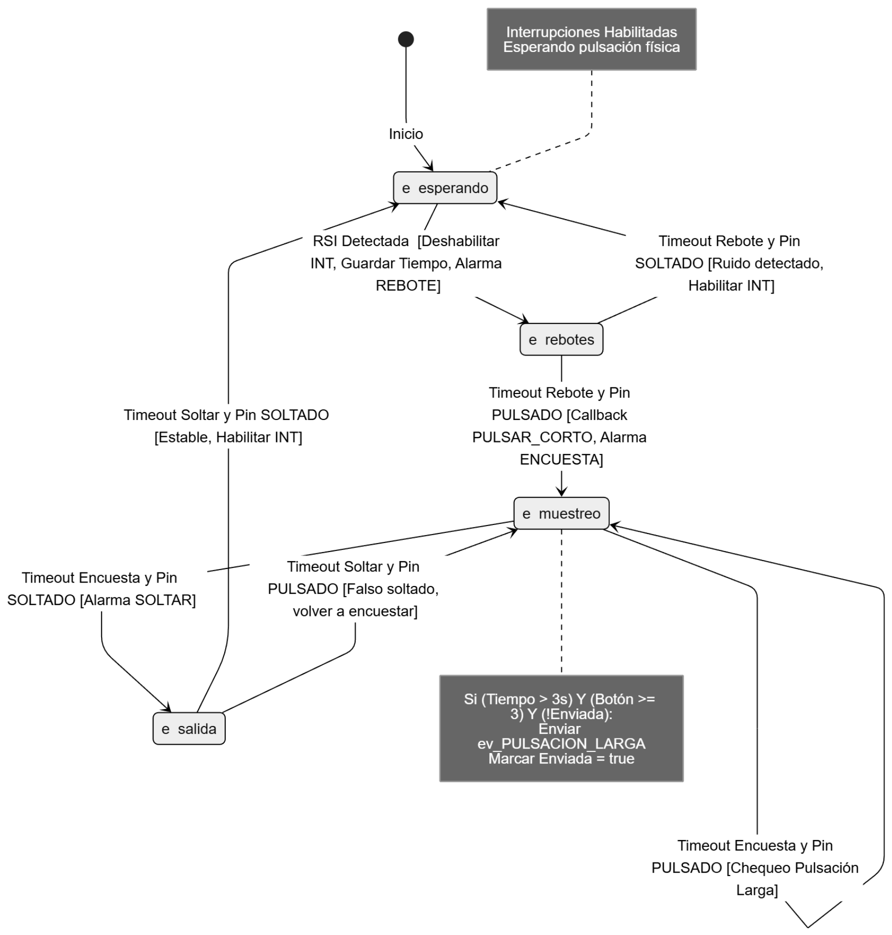

Interrupciones Habilitadas
Esperando pulsación física
Inicio
e esperando
RSI Detectada [Deshabilitar INT, Guardar Tiempo, Alarma REBOTE]
Timeout Rebote y Pin SOLTADO [Ruido detectado, Habilitar INT]
e rebotes
Timeout Soltar y Pin SOLTADO [Estable, Habilitar INT]
Timeout Rebote y Pin PULSADO [Callback PULSAR_CORTO, Alarma ENCUESTA]
e muestreo
Timeout Encuesta y Pin SOLTADO [Alarma SOLTAR]
Timeout Soltar y Pin PULSADO [Falso soltado, volver a encuestar]
e salida
Si (Tiempo > 3s) Y (Botón >= 3) Y (!Enviada):
Enviar
ev_PULSACION_LARGA
Marcar Enviada = true
Timeout Encuesta y Pin PULSADO [Chequeo Pulsación Larga]

Figure 1: Button driver state machine with debouncing

Configured timings:

- `RETARDO_REBOTE_MS = 50`: filter after edge detection
- `RETARDO_SOLTAR_MS = 50`: stabilization after release
- `RETARDO_ENCUESTA_MS = 20`: polling while pressed
- `PULSACION_LARGA = 3000`: long-press threshold

3.4.3. ISR Implementation Differences

LPC2105: Level-triggered external interrupts require reconfiguring PINSEL during the debounce interval:

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

nRF52840: Use of the GPIOTE SENSE/LATCH mechanism:

```c
void hal_ext_int_habilitar(uint8_t id_boton) {
    // SENSE_Low -> generates PORT event when pin=0
    cnf |= (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos);
}
```

3.4.4. GPIO Monitors

GPIO monitors (`drv_monitor.c`) are output pins used to correlate software state with oscilloscope or power analyzer measurements:

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

Typical usage (may vary with the session being executed and can be adapted as desired):

- Monitor 1: indicates periodic ISR execution
- Monitor 2: marks 64-bit clock overflow
- Monitor 3: signals low-power mode entry
- Monitor 4: marks overflow in the event queue, alarms, or subscriptions

3.5. Watchdog

The hardware watchdog provides recovery from lockups.

Feeding from the main loop:

```c
void rt_GE_lanzador() {
    while(1) {
        drv_WDT_alimentar();  // prevents reset
        // ... event processing
    }
}
```

3.6. Concurrency and Event Model

3.6.1. Event Definition

Events are defined in `rt_evento_t.h`:

```c
typedef enum {
    ev_VOID = 0,            // null event
    ev_T_PERIODICO = 1,     // timer tick
    ev_BOTON_PULSAR = 2,    // button pressed
    ev_INACTIVIDAD = 3,     // inactivity timeout
    ev_BOTON_RETARDO = 4,   // debounce handling
    ev_JUEGO_TICK = 5,      // game timing
    ev_PULSACION_LARGA = 6 // press >=3s
} EVENTO_T;

#define EVENT_TYPES 7
```

3.6.2. FIFO Event Queue

The circular FIFO queue (`rt_fifo.c`) stores events pending processing. It has 32 slots, providing sufficient headroom for event bursts while remaining bounded to detect overflow and facilitate diagnostics.

Operations:

- `rt_FIFO_encolar(ID_evento, auxData)`: enqueues an event with timestamp
- `rt_FIFO_extraer(*ID_evento, *auxData, *TS)`: dequeues the next event
- `rt_FIFO_estadisticas(ID_evento)`: queries the per-event-type counter

3.6.3. Overflow Handling

If the queue becomes full, the system enters an infinite diagnostic loop after asserting a monitor:

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

This strategy makes the issue observable using an analyzer connected to the monitor signal.

3.6.4. Critical Sections

Critical sections protect shared data structures accessed both by the main loop (Thread mode) and ISRs (Handler mode).

Interface (`rt_sc.h`):

```
rt_sc_estado_t rt_sc_entrar(void);   // disables IRQs, saves state
void rt_sc_salir(rt_sc_estado_t);    // restores previous state
```

3.7. Power Management

3.7.1. Power Modes

The system implements three power levels:

Mode	Function	Behavior
Active	Normal	CPU executing code
Wait	`drv_consumo_esperar()`	WFI, wakes on IRQ
Sleep	`drv_consumo_dormir()`	System OFF, requires reset

Table 5: System power modes

3.7.2. Busy-Wait vs. Low Power vs. Deep Sleep

Busy-wait:

```
delay = 1000000;

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

- Constant maximum consumption

- Not recommended for battery-powered devices

Wait mode:

- CPU halted until next IRQ
- Significant consumption reduction during idle

Deep sleep:
nRF52840:

```c
void hal_consumo_dormir(void) {
    NRF_POWER->SYSTEMOFF = POWER_SYSTEMOFF_SYSTEMOFF_Enter;
    __WFE();
    while (1) { }  // does not return
}
```

LPC2105:

```c
void hal_consumo_dormir(void) {
    EXTWAKE = 7;       // EINT0,1,2 as wake-up sources
    PCON |= 0x02;      // Power-down
    Reset_Handler();   // restart after wake-up
}
```

3.7.3. Wake-Up by Button

Before entering sleep mode, buttons are configured as wake-up sources:

```c
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. Inactivity Timeout

The system schedules a 20-second inactivity alarm:

```c
void rt_GE_lanzador() {
    svc_alarma_activar(
        svc_alarma_codificar(0, 20*1000, 0),
        ev_INACTIVIDAD, 0
    );
    // ... event loop
}
```

Upon receiving `ev_BOTON_PULSAR`, the alarm is re-armed.

3.8. System Services

3.8.1. Alarm Service

The alarm service (`svc_alarmas.c`) manages up to 6 concurrent alarms (sufficient for: inactivity + debounce handling for up to 4 buttons + game tick). A 1 ms timer is used to update all alarms.
Alarm types:

- **One-shot:** trigger once and then deactivate

- **Periodic:** automatically re-armed after triggering

API:

```c
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);
```

Internal operation:

1. The periodic timer generates `ev_T_PERIODICO`
2. `svc_alarma_actualizar` decrements counters
3. When an alarm expires, it enqueues the associated event
4. Periodic alarms are automatically re-armed

3.8.2. Event Manager (Dispatcher)

The event manager (`rt_GE.c`) implements the main loop:

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

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

            // Invoke subscribed callbacks
            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();  // Empty queue -> WFI
        }
    }
}
```

3.8.3. Subscription and Cancellation

The subscription service (`svc_GE.c`) allows registering callbacks per event and priority:

```c
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);
```

Execution order: Callbacks are executed by priority (lower value $\Rightarrow$ higher priority).

4. Results

4.1. Bit Counter Strike

4.1.1. Functional Description

Bit Counter Strike is a reaction game in which the player must press the button corresponding to the illuminated LED before a response deadline expires. Difficulty increases progressively by reducing both response time and the pause between LEDs.

Mechanics:

1. An LED is turned on sequentially (corresponding to an LED-button pair).
2. The player has limited time to press the correct button.
3. If correct: difficulty increases (times are reduced) and the next LED is selected.
4. If incorrect: an error blink is shown, but the game continues.
5. On timeout: the LED turns off and the game continues with the next LED.

4.1.2. Game state diagram

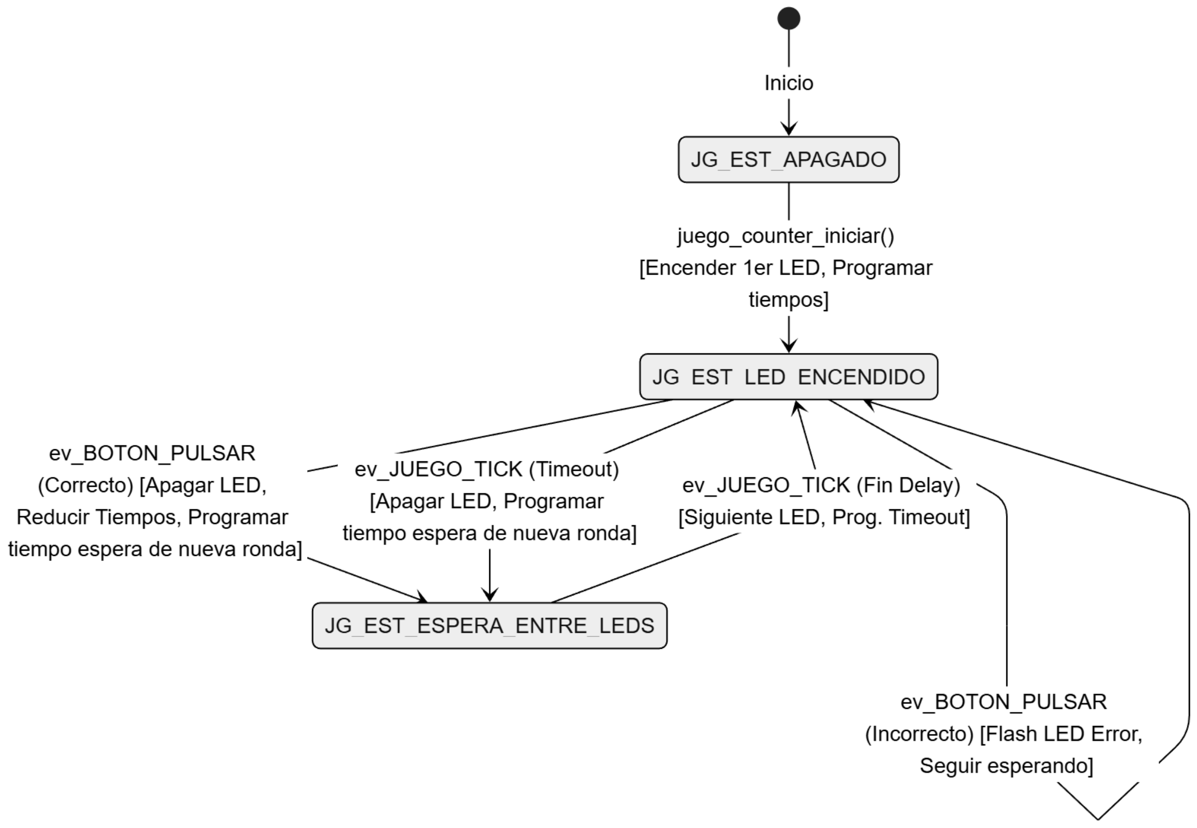

Inicio
JG_EST_APAGADO
juego_counter_iniciar() [Encender 1er LED, Programar tiempos]
JG_EST_LED_ENCENDIDO
ev_BOTON_PULSAR (Correcto) [Apagar LED, Reducir Tiempos, Programar tiempo espera de nueva ronda]
ev_JUEGO_TICK (Timeout) [Apagar LED, Programar tiempo espera de nueva ronda]
ev_JUEGO_TICK (Fin Delay) [Siguiente LED, Prog. Timeout]
JG_EST_ESPERA_ENTRE_LEDS
ev_BOTON_PULSAR (Incorrecto) [Flash LED Error, Seguir esperando]

Figure 2: State diagram of the Bit Counter Strike game

4.1.3. Configuration Parameters

Parameter	Initial value	Minimum value
Response time	5000 ms	500 ms
Pause between LEDs	2000 ms	80 ms
Reduction (response)	500 ms	—
Reduction (pause)	20 ms	—

Table 6: Configuration parameters of the Bit Counter Strike game

4.1.4. Difficulty Progression

Each time the player presses the correct button within the deadline:

- The response time is reduced by 500 ms (down to a minimum of 500 ms).
- The pause between LEDs is reduced by 20 ms (down to a minimum of 80 ms).

LEDs are enabled sequentially and cyclically, using the number of valid LED-button pairs according to the board configuration.

4.1.5. System Integration

The game integrates through subscriptions to the event manager:

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

4.2. Final Game: Beat Hero

4.2.1. Functional Description

Beat Hero is a rhythm game in which the player must press buttons corresponding to an LED pattern in sync with a tempo (BPM) that increases progressively.

Mechanics:

1. Two 2-LED (2-bit) patterns are displayed: the current pattern and the next one.
2. The player presses the buttons corresponding to the current pattern: button 1 for LED 1 and button 2 for LED 2.
3. The timing accuracy determines the score (+2, +1, 0, -1).
4. The tempo increases as levels progress.
5. Pressing button 3 or 4 ends the session.

4.2.2. Game state diagram

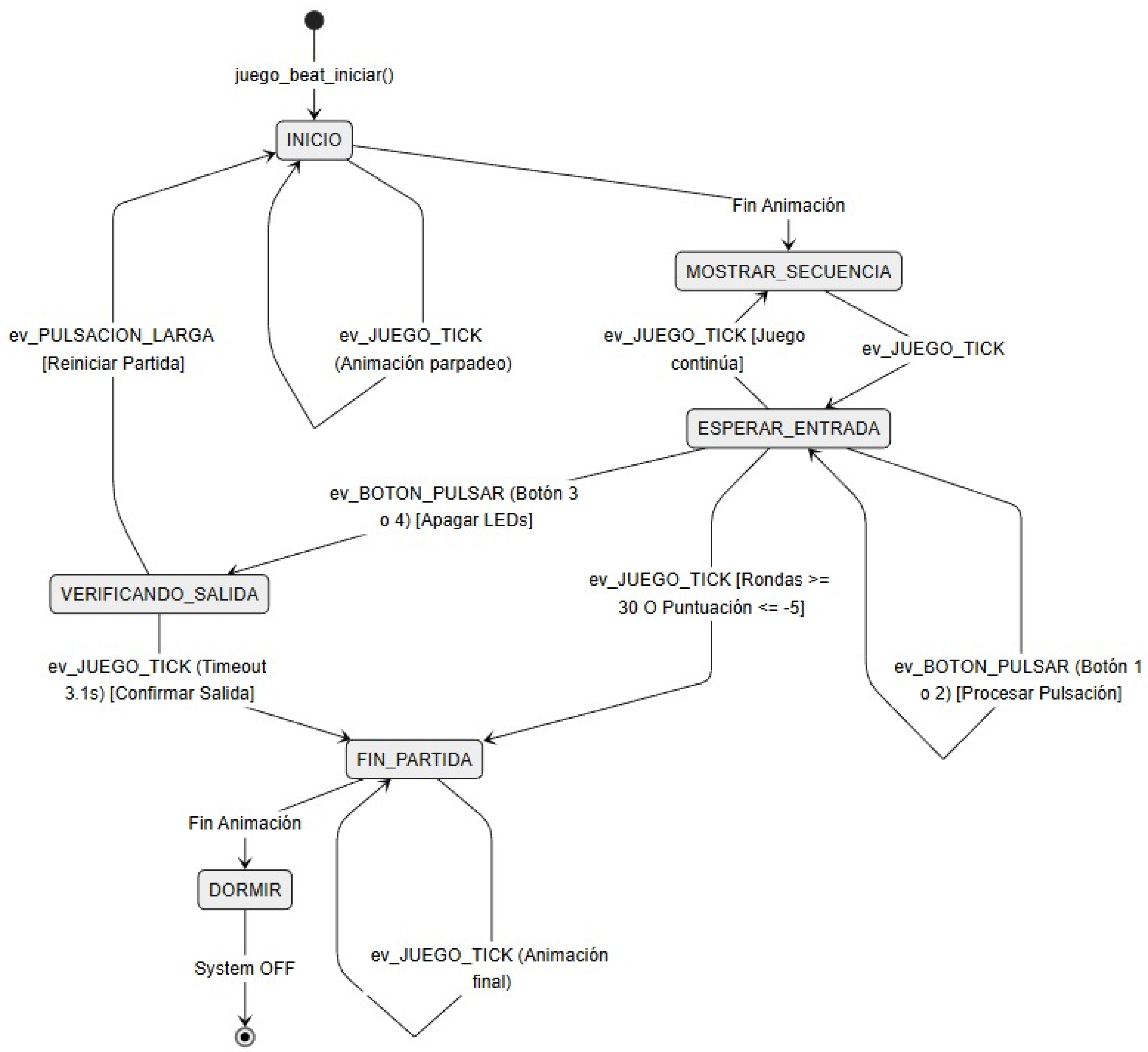

juego_beat_iniciar()
INICIO
Fin Animación
MOSTRAR_SECUENCIA
ev_PULSACION_LARGA [Reiniciar Partida]
ev_JUEGO_TICK (Animación parpadeo)
ev_JUEGO_TICK [Juego continúa]
ev_JUEGO_TICK
ESPERAR_ENTRADA
ev_BOTON_PULSAR (Botón 3 o 4) [Apagar LEDs]
VERIFICANDO_SALIDA
ev_JUEGO_TICK [Rondas >= 30 O Puntuación <= -5]
ev_JUEGO_TICK (Timeout 3.1s) [Confirmar Salida]
ev_BOTON_PULSAR (Botón 1 o 2) [Procesar Pulsación]
FIN_PARTIDA
Fin Animación
DORMIR
System OFF
ev_JUEGO_TICK (Animación final)

Figure 3: State diagram of the Beat Hero game

4.2.3. Scoring System

The score depends on button press timing accuracy:

Accuracy	Points	Condition
Excellent	+2	≤10% of the bar period
Good	+1	≤20% of the period
Acceptable	0	≤40% (window)
Poor	-1	>40% or incorrect button

Table 7: Beat Hero scoring system

4.2.4. Difficulty Progression

- **Initial BPM:** 60
- **Bars per level:** configurable
- **Maximum levels:** 3
- **At maximum level:** BPM increases by +2 per bar

```c
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. Metrics and Statistics

The game maintains detailed statistics:

- Total score and per-level score
- Average reaction time
- Fastest reaction time
- Bars played

4.2.6. Reset via Long Press

A long press ($\geq$3 s) on button 3 or 4 resets the session:

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

4.3. Validation and Testing

For execution information, detailed Beat Hero behavior, and detailed instructions on how to run various tests, refer to `README_EjecutarNuestroTestPH.pdf` in the project directory.

4.3.1. Module Tests

All development sessions were completed successfully:

Session	Test	Result
1	Busy-wait LED blink	OK
2	Timer-based LED blink	OK
3	Periodic blink + callback	OK
4	FIFO event queue	OK
5	Queue overflow test	OK
6	Low-power blink	OK
7	Bit Counter Strike	OK
8	Complete Beat Hero	OK

Table 8: Per-session module test results

4.3.2. Queue Overflow Test

With `SESION` set to 5, an explicit test can be executed:

```c
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; // must be reached (32 OK, no overflow).

    // Intentional overflow
    rt_FIFO_encolar(ev_VOID, 1); // must enter infinite loop due to overflow

    fin = -1;      // must never reach here
}
```

4.3.3. Watchdog Verification

In the `VERIFICANDO_SALIDA` state, an infinite loop can be uncommented to test the watchdog (line 328 of `juego_beat.c`). To trigger the issue, press button 3 or 4 during gameplay:

```c
case VERIFICANDO_SALIDA:
    // while(1); // uncomment to verify watchdog
```

The 10-second watchdog triggers a reset if it is not fed.

4.3.4. Power Consumption Measurements

Captures were performed using Nordic PPK2 (Power Profiler Kit). The project folder contains, under `/contenido_practicas_anteriores`, multiple `.ppk2` files and related photographs:

- `blinkv2`: fairly flat trace at ~6 mA with peaks when toggling the LED/timer. This indicates the CPU is always awake (busy-wait).

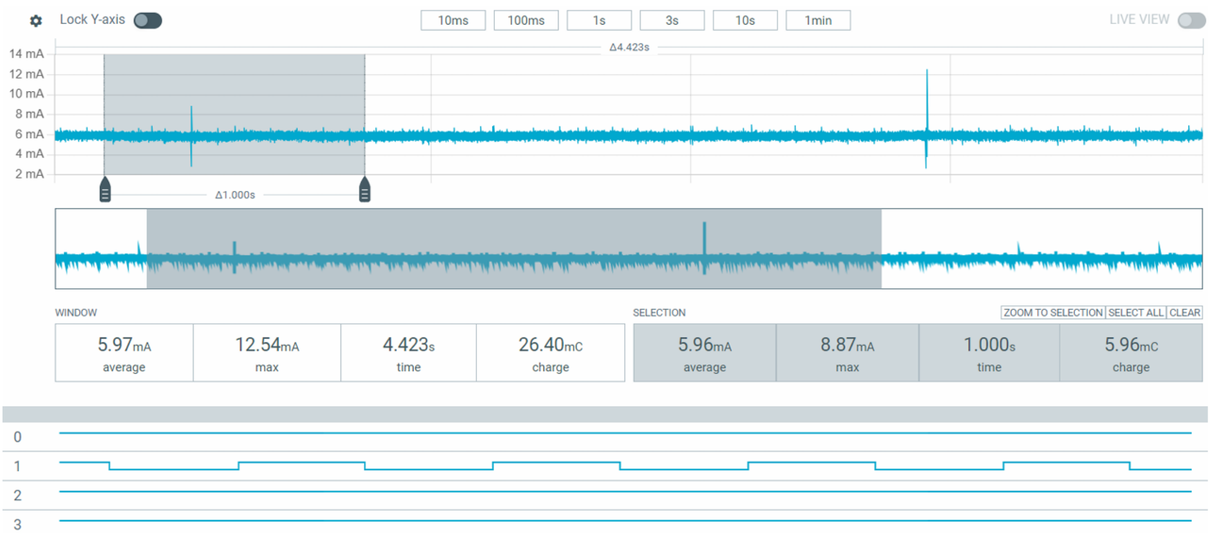


Figure 4: Power measurement — Blink v2 (busy-wait)

- `blinkv3`: waveform with narrow spikes: the CPU sleeps and only wakes up to handle IRQs and toggle the LED.

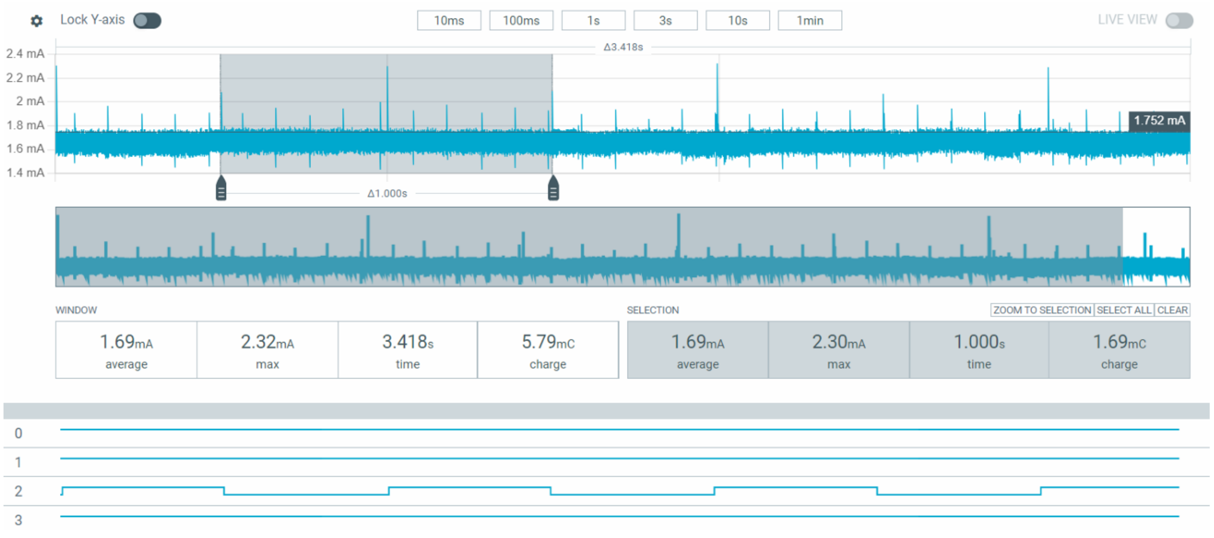
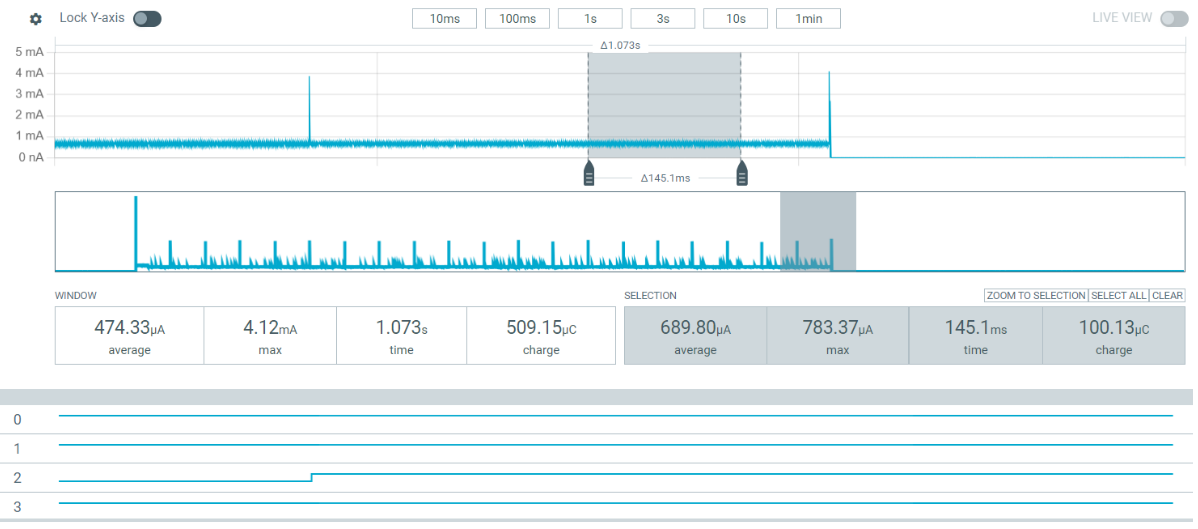


Figure 5: Power measurement — Blink v3 (low power with WFI)

- `blinkv3bis (awake)`: consumption during active blinking.



WINDOW

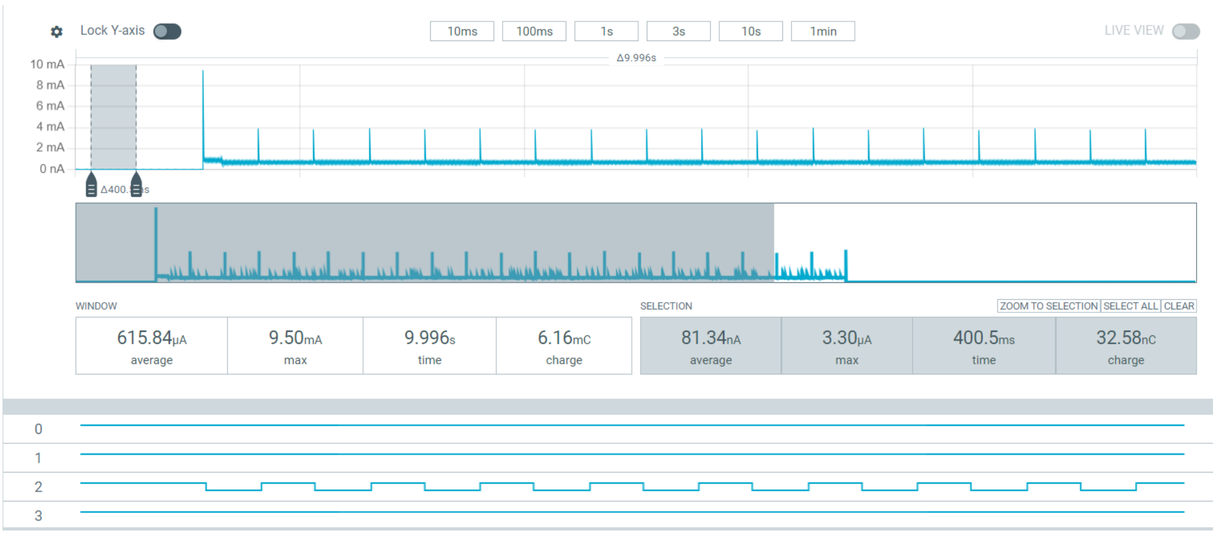
474.33µA	4.12mA	1.073s	509.15µC
average	max	time	charge

SELECTION

689.80µA	783.37µA	145.1ms	100.13µC
average	max	time	charge

Figure 6: Power measurement — Blink v3 bis (awake mode)

- `blinkv3bis (asleep)`: consumption in System OFF.



WINDOW

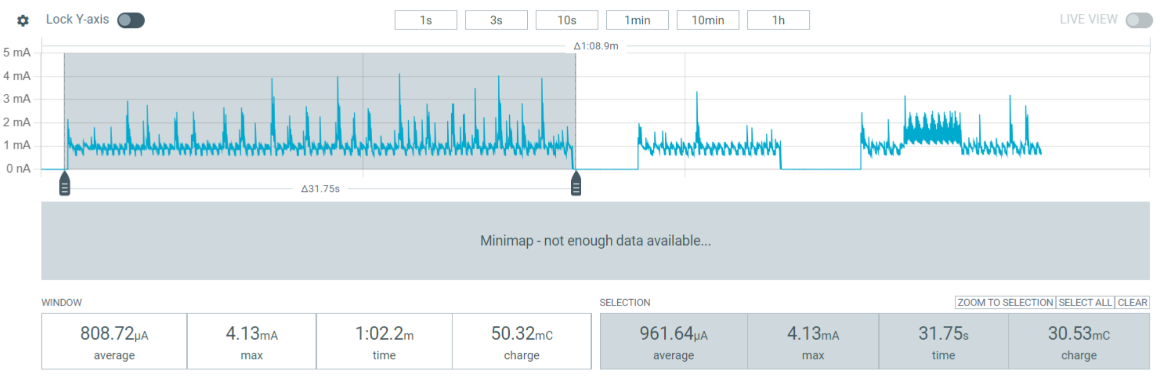
615.84µA	9.50mA	9.996s	6.16mC
average	max	time	charge

SELECTION

81.34nA	3.30µA	400.5ms	32.58nC
average	max	time	charge

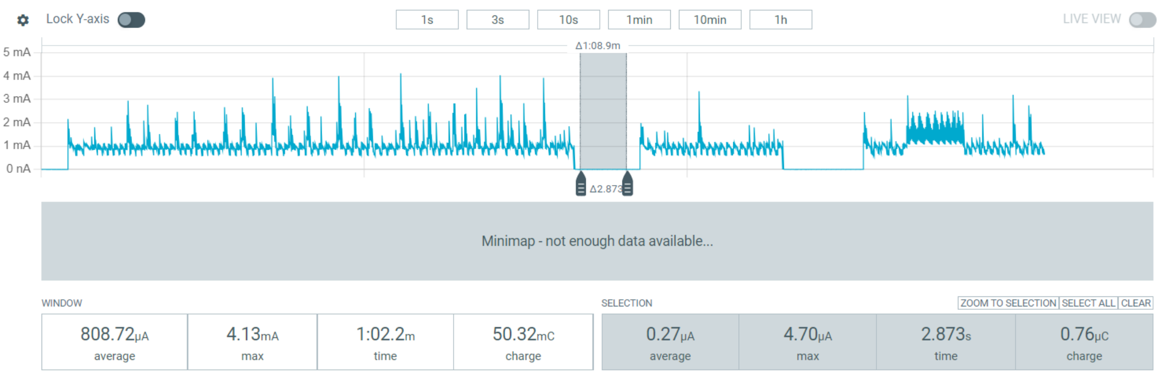
Figure 7: Power measurement — Blink v3 bis (sleep/System OFF)

- `Beat Hero`:



WINDOW				SELECTION			
808.72µA average	4.13mA max	1:02.2m time	50.32mC charge	961.64µA average	4.13mA max	31.75s time	30.53mC charge

Figure 8: Power measurement — Beat Hero during gameplay



WINDOW				SELECTION			
808.72µA average	4.13mA max	1:02.2m time	50.32mC charge	0.27µA average	4.70µA max	2.873s time	0.76µC charge

Figure 9: Power measurement — Beat Hero with the CPU sleeping

To evaluate the system's energy behavior, current consumption measurements were performed on the nRF52840 board in two scenarios: active gameplay and CPU in sleep mode.
Consumption during gameplay:
The measurement corresponding to a complete session (duration: 31.75 s) shows:

- Average current: 961.64 µA
- Peak current: 4.13 mA
- Total charge: 30.53 mC

The profile exhibits a baseline close to 1 mA with periodic peaks up to 4 mA, corresponding to interrupt servicing, scheduler execution, event handling, and peripheral activity. This behavior is consistent with an event-driven system in which the CPU wakes only when there is work to perform.
Consumption in sleep mode (WFI):
With the CPU in low-power mode and no pending events (duration: 2.873 s):

- Average current: 0.27 µA
- Peak current: 4.70 µA

- Total charge: 0.76 µC

Current consumption falls in the sub-microamp range, a reduction of more than three orders of magnitude compared to active execution.
Comparison:

System state	Average current
Gameplay running	∼962 µA
CPU sleeping	∼0.27 µA

Table 9: Power consumption comparison across system states

Idle consumption is approximately 3500 times lower than during active gameplay, confirming that the event-driven design meets the energy-efficiency objectives: busy-waiting is eliminated, the CPU remains awake only when necessary, and time spent in low-power mode is maximized without loss of responsiveness.

5. Conclusions

5.1. Degree of Objective Fulfillment

Objective	Status
Layered architecture	✓ Met
LPC/nRF portability	✓ Met
Event-driven model	✓ Met
64-bit time management	✓ Met
Low-power operation	✓ Met
Watchdog	✓ Met
Beat Hero game	✓ Met
Debouncing	✓ Met

Table 10: Objective fulfillment status

5.2. Conclusions and Lessons Learned

This project enabled us to meet all stated objectives and to acquire key competencies in embedded systems development. Across eight working sessions, we learned how to design and implement a modular five-layer software architecture (HAL, drivers, runtime, services, and application) that ensures portability across different ARM platforms, understanding the importance of hardware abstraction and separation of concerns.

We gained hands-on experience with event-driven programming by implementing an event management system based on a circular FIFO queue and priority-based subscriptions, which taught us how to handle concurrency through critical sections and how to prevent race conditions. Developing the software alarm system improved our understanding of time management in embedded systems, including extending 32-bit counters to 64-bit through software techniques. In addition, we learned energy-optimization techniques through low-power modes (WFI, System OFF) and implemented robustness mechanisms such as a hardware watchdog to recover from lockups.

The successful implementation of the Beat Hero and Bit Counter Strike games demonstrates the feasibility of the complete system and allowed us to apply all these concepts in an integrated practical project, developing skills in embedded debugging, the use of tools such as Keil µVision, power measurement with Nordic PPK2, and understanding platform differences at the hardware level.

5.3. Self-Assessment Notes

Name	**NIP**	**Grade**	**Comments**
Fernando	897113	8	I learned a lot about embedded systems, and it was a pleasure to work for the first time with my colleague Guillermo.
Guillermo	896594	8	Carrying out a project like this, where the workload has been considerably higher than what we are used to, has been a major challenge. However, I believe I learned a lot in terms of organization and coordination, since although it was the first time working with my colleague Fernando, we managed to meet the objective as a team.

Table 11: Authors' self-assessment

References

- NXP Semiconductors. *UM10275: LPC2104/05/06 User Manual* (Rev. 02). NXP B.V., April 8, 2009. Available at: `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, July 4, 2024. [Online]. Available at: `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, December 3, 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, August 3, 2011.
- Arm Limited. *100166_0001_00_en: ARM Cortex-M4 Processor Technical Reference Manual* (Revision r0p1). Arm Limited, February 23, 2015.
- Arm Limited. *DDI 0403E.e: Arm v7-M Architecture Reference Manual.* Arm Limited, February 15, 2021.
- ARM Limited. *ARM7TDMI Technical Reference Manual.* ARM Limited, 1995.
- Keil. *µVision User's Guide.* ARM Keil, 2024.
- Teaching staff of the course "Hardware Project". *Course material and laboratory guides.* University of Zaragoza, 2025.

Appendices

Appendix A: Statistics

FIFO Queue Statistics

The rt_fifo.c module maintains event statistics that are publicly accessible via rt_FIFO_estadisticas():

- estadisticas[EVENT_TYPES]: counter of enqueued events per type
- total: historical total of enqueued events

Usage:

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

Event Manager Latency Metrics

The event manager rt_GE.c computes latency metrics internally for performance analysis:

- tiempo_max_sin_tratar: maximum latency between enqueue and processing
- media_tiempo_sin_tratar: accumulated average latency
- eventos_tratados: total number of processed events
- tiempo_sin_tratar: total latency accumulator

Note: These variables are internal to the module and are not exposed through a public API. They are accessible only via the debugger for diagnostics.

Beat Hero Game Statistics

The Beat Hero game (juego_beat.c) maintains player performance statistics:

- puntuacion: total accumulated score
- puntuacion_por_compas: average score per bar
- puntuacion_por_nivel[4]: cumulative score per level
- tiempo_reaccion_total: sum of all reaction times
- tiempo_reaccion_medio_ms: average reaction time
- tiempo_reaccion_mas_rapido_ms: best recorded reaction time

These statistics are updated in real time during gameplay and are used to evaluate player performance.

Appendix B: Beat Hero Game (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>

// ===========================
//    CONSTANTS
// ===========================
#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

// ------------------- Internal state ------------------------
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;

    //----- Statistics -------
    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;
    uint32_t tiempo_reaccion_mas_rapido_ms;
} juego_t;

static juego_t j;

// --------------- Helpers -------------------

static void juego_apagar_todos(void){
    for (uint8_t i = 1; i <= LEDS_NUMBER; ++i) drv_led_establecer((LED_id_t)i, LED_OFF);
}

static void juego_encender_todos(void){
    for (uint8_t i = 1; i <= LEDS_NUMBER; ++i) drv_led_establecer((LED_id_t)i, LED_ON);
}

static void juego_reprogramar(uint32_t retardo_ms){
    svc_alarma_activar(svc_alarma_codificar(0, retardo_ms, 0), ev_JUEGO_TICK, 0);
}

static void juego_resetear_inactividad(void) {
    svc_alarma_activar(svc_alarma_codificar(0, JUEGO_TIMEOUT_INACTIVIDAD_MS, 0), ev_INACTIVIDAD, 0);
}

// --------------- Logic  -------------------

static uint8_t generar_compas(void){
    static uint8_t ultimo_generado = 0;
    uint32_t r;
    uint8_t resultado = 0;

    do {
        r = drv_random_generar_rango(4); // Generates 0, 1, 2, 3

        switch (j.nivel) {
            case 1:
                // Level 1: ONLY one LED ON (1 or 2).
                // No empty bars (0) and no double bars (3) are allowed.
                if (r == 0) resultado = 1;
                else if (r == 3) resultado = 2;
                else resultado = (uint8_t)r;
                break;

            case 2:
                // Level 2: empty bars (0) are allowed.
                // Double bars (3) are NOT allowed.
                if (r == 3) resultado = 0;
                else resultado = (uint8_t)r;
                break;

            default:
                // Levels 3 and 4: allow two LEDs (3).
                resultado = (uint8_t)r;
                break;
        }

        // In level 1, enforce alternation for visual clarity.
        if (j.nivel == 1 && ultimo_generado != 0 && resultado == ultimo_generado) {
            // If repeated in level 1, force the opposite pattern.
            resultado = (resultado == 1) ? 2 : 1;
        }

    } while (resultado == ultimo_generado && j.nivel > 1); // Avoid exact
        repetition at higher levels.

    ultimo_generado = resultado;
    return resultado;
}

static void avanzar_compas(void){
    j.compas[0] = j.compas[1];
    j.compas[1] = j.compas[2];
    j.compas[2] = generar_compas();
}

static void mostrar_compases(void){
    juego_apagar_todos();
    // TOP (LEDs 1 and 2).
    if (j.compas[2] & 0x01) drv_led_establecer(1, LED_ON);
    if (j.compas[2] & 0x02) drv_led_establecer(2, LED_ON);

    // BOTTOM (LEDs 3 and 4).
    if (j.compas[1] & 0x01) drv_led_establecer(3, LED_ON);
    if (j.compas[1] & 0x02) drv_led_establecer(4, LED_ON);
}

static void procesar_pulsacion(uint32_t boton_id) {
    uint32_t ahora = drv_tiempo_actual_ms();
    if (!j.primera_pulsacion) {
        j.primera_pulsacion = true;
        j.timestamp_primera_pulsacion = ahora;
        j.patron_recibido = 0;
    }
    if (boton_id == 1) j.patron_recibido |= 0x01;
    if (boton_id == 2) j.patron_recibido |= 0x02;
}

static void evaluar_ronda(void){
    if (!j.primera_pulsacion) { // If the player did not press anything...
        if (j.compas[0] != 0) j.puntuacion--;
        return;
    }

    uint32_t tiempo = j.timestamp_primera_pulsacion - j.timestamp_inicio;

    // Outside the window (>40%): -1 point.
    if (tiempo > j.ventana_ms) {
        j.puntuacion -= 1;
        return;
    }

    // Incorrect pattern: -1 point.
    uint8_t esperado = j.compas[0];
    if (j.patron_recibido != esperado) {
        j.puntuacion -= 1;
        return;
    }

    // Compute accuracy.
    uint32_t porcentaje = (tiempo * 100) / j.t_compas_ms;

    // Point assignment:
```

```c
    if (porcentaje <= JUEGO_PRECISION_10) {
        j.puntuacion += 2; // +2 if within 10%
    }
    else if (porcentaje <= JUEGO_PRECISION_20) {
        j.puntuacion += 1; // +1 if within 20%
    }
    // 0 points if between 20% and 40% (no change).

    // Update statistics
    j.puntuacion_por_nivel[j.nivel-1] += j.puntuacion;
    if (j.compases_jugados > 0) j.puntuacion_por_compas = j.puntuacion / j.compases_jugados;
    j.tiempo_reaccion_total += tiempo;
    if (j.compases_jugados > 0) j.tiempo_reaccion_medio_ms = j.tiempo_reaccion_total / j.compases_jugados;
    if (tiempo < j.tiempo_reaccion_mas_rapido_ms) j.tiempo_reaccion_mas_rapido_ms = tiempo;

}

static void actualizar_nivel_y_bpm(void){
    // If all bars for the current level have been completed, advance to the next level (if any remain).
    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;   // Increase difficulty.

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

// ------------------- Public interface -----------------------

void juego_beat_iniciar(void){
    juego_apagar_todos();
    j.nivel = 1;
    j.bpm = JUEGO_BPM_INICIAL;
    j.puntuacion = 0;
    j.puntuacion_por_compas = 0;
    for (uint8_t i=0; i<JUEGO_NIVEL_MAX; i++) j.puntuacion_por_nivel[i] = 0;
    j.tiempo_reaccion_total = 0;
    j.tiempo_reaccion_medio_ms = 0;
    // The reaction time cannot be larger than the window; otherwise a point would be lost.
    j.tiempo_reaccion_mas_rapido_ms = j.ventana_ms;
    // Starts at -1 because the first "pixel" must drop to the bottom before the game begins.
    j.compases_jugados = -1;
    j.t_compas_ms = 60000/ j.bpm;
    j.patron_recibido = 0;
    j.primera_pulsacion = false;
    j.timestamp_primera_pulsacion = 0;

    j.compas[0] = 0;
    j.compas[1] = 0;
    j.compas[2] = generar_compas();

    juego_resetear_inactividad();

    j.paso_animacion = 0;
    j.estado = INICIO;
    juego_encender_todos();
    juego_reprogramar(ANIM_PARPADEO_MS);
}

void juego_beat_actualizar(EVENTO_T evento, uint32_t auxData){

    // If the inactivity period expires, enter sleep.
    if (evento == ev_INACTIVIDAD && j.estado != DORMIR) {
        juego_apagar_todos();
        j.estado = DORMIR;
        juego_reprogramar(1);
        return;
    }
    // If a long press is detected, start a new session.
    if (evento == ev_PULSACION_LARGA && j.estado != DORMIR) {
        juego_beat_iniciar();
        return;
    }

    switch (j.estado) {
        case INICIO:
            if (evento == ev_JUEGO_TICK) {
                // Play start-of-game animation.
                j.paso_animacion++;
                switch(j.paso_animacion) {
                    case 1: juego_apagar_todos();   juego_reprogramar(
                        ANIM_PARPADEO_MS); break;
                    case 2: juego_encender_todos(); juego_reprogramar(
                        ANIM_PARPADEO_MS); break;
                    case 3: juego_apagar_todos();   juego_reprogramar(200);
                        break;
                    default:
                        // Once the animation ends, transition to
                            MOSTRAR_SECUENCIA.
                        j.estado = MOSTRAR_SECUENCIA;
                        juego_reprogramar(1);
                        break;
                }
            }
            break;

        case MOSTRAR_SECUENCIA:
            if (evento == ev_JUEGO_TICK) {
                // Show the LED sequence to the player and then wait for input.
                mostrar_compases();
                j.timestamp_inicio = drv_tiempo_actual_ms();
                j.patron_recibido = 0;
                j.primera_pulsacion = false;
                juego_reprogramar(j.t_compas_ms);
                j.estado = ESPERAR_ENTRADA;
            }
            break;

        case ESPERAR_ENTRADA:
            if (evento == ev_BOTON_PULSAR) {
                // Reset inactivity timer on any press.
                juego_resetear_inactividad();
                // If button 3 or 4 is pressed, the player is requesting exit (
                    or restart),
                // so transition to VERIFICANDO_SALIDA.
                if (auxData >= 3) { // Exit
                    juego_apagar_todos();
                    j.estado = VERIFICANDO_SALIDA;
                    // >3 seconds so that there is time to trigger the restart long press.
                    juego_reprogramar(3100);
                    return;
                }
                // Otherwise, record the input pattern for later evaluation.
                if (auxData == 1 || auxData == 2) procesar_pulsacion(auxData);

            } else if (evento == ev_JUEGO_TICK) {
                // When the bar period expires, compute score and statistics.
                j.compases_jugados++;
                evaluar_ronda();
                avanzar_compas();
                // Update level if needed.
                actualizar_nivel_y_bpm();

                // End the game if max bars reached or score too low.
                // (Comment out "|| j.puntuacion <= JUEGO_PUNTOS_MIN" to avoid losing by score and
                //  observe inactivity-based sleep instead.)
                if (j.compases_jugados >= JUEGO_COMPASES_MAX || j.puntuacion <= JUEGO_PUNTOS_MIN) {
                    j.estado = FIN_PARTIDA;

                    if (j.compases_jugados >= JUEGO_COMPASES_MAX) {
                        j.anim_exito = true;  // Win.
                    } else {
                        j.anim_exito = false; // Lose.
                    }
                    // Prepare end-of-game animation.
                    j.paso_animacion = 0;
                    juego_reprogramar(1);
                } else {
                    // Otherwise proceed to the next bar/round.
                    j.estado = MOSTRAR_SECUENCIA;
                    juego_reprogramar(1);
                }
            }
            break;

        case VERIFICANDO_SALIDA:
            // while(1); // uncomment to verify watchdog

            // If a long-press event arrives: RESTART.
            if (evento == ev_PULSACION_LARGA) {
                juego_beat_iniciar();
            }
            // If the 3100 ms timeout expires and no long press occurred: end the game.
            else if (evento == ev_JUEGO_TICK) {
                j.estado = FIN_PARTIDA;
                j.paso_animacion = 0;
                juego_reprogramar(1);
            }
            break;

        case FIN_PARTIDA:
            if (evento == ev_JUEGO_TICK) {
                // Show animation.
                j.paso_animacion++;
                if (j.paso_animacion < 8) {
                    if (j.paso_animacion % 2 != 0) {
                        // Select animation depending on outcome.
                        if (j.anim_exito) {
                            juego_encender_todos(); // VICTORY: all LEDs on.
                        } else {
                            // DEFEAT: cross using LEDs 1 and 4.
                            juego_apagar_todos();
                            drv_led_establecer(1, LED_ON);
                            drv_led_establecer(4, LED_ON);
                        }
                    }
                    else juego_apagar_todos();

                    juego_reprogramar(ANIM_FIN_MS);
                } else {
                    // When the animation ends, transition to sleep.
                    j.estado = DORMIR;
                    juego_reprogramar(1);
                }
            }
            break;

        case DORMIR:
            juego_apagar_todos();
            drv_consumo_dormir();
            break;
    }
}
```

Appendix C: Event Manager (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"


// Data structure to store subscriptions
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(){

    // Inactivity alarm after 20 seconds with no activity.
    svc_alarma_activar(svc_alarma_codificar(0,20*1000,0), ev_INACTIVIDAD, 0);
    EVENTO_T id_ev;
    uint32_t aux;
    Tiempo_us_t ts;
    while(1){

        drv_WDT_alimentar(); // Feed the watchdog

        // If there are events in the FIFO...
        if(rt_FIFO_extraer(&id_ev, &aux, &ts) != (uint8_t)-1){
            // Update statistics
            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;

            // Walk the subscription list and invoke callbacks
            if ((uint8_t)id_ev < EVENT_TYPES) {
                for (uint8_t i = 0; i < ge_tabla[(uint8_t)id_ev].num_suscritos;
                    i++) {
                    if (ge_tabla[(uint8_t)id_ev].lista_callbacks[i] != NULL) {
                        ge_tabla[(uint8_t)id_ev].lista_callbacks[i](id_ev, aux);
                    }
                }
            }
        } else{
            // Otherwise, wait
            drv_consumo_esperar();
        }
    }

}

// Helper
static void preparar_despertar_inactividad(void){
#if BUTTONS_NUMBER > 0
    for (uint8_t i = 1; i <= BUTTONS_NUMBER; ++i) {
        hal_ext_int_habilitar(i);   // SENSE=LOW on ALL buttons
    }
#endif
}


static uint32_t contador_alarma = 0;

// Re-arm
void rt_GE_actualizar(EVENTO_T ID_EVENTO, uint32_t auxData){

    switch((uint8_t)ID_EVENTO){
        case ev_INACTIVIDAD:
            preparar_despertar_inactividad();
            drv_consumo_dormir();
            break;

        case ev_BOTON_PULSAR:
            contador_alarma++;
            svc_alarma_activar(
                svc_alarma_codificar(0, 20000, 0),
                ev_INACTIVIDAD,
                contador_alarma
            );
            break;

        default:
            break;
    }
}
```

Appendix D: FIFO Queue (rt_fifo.c)

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

// Queue parameters.
#ifndef RT_FIFO_CAPACITY
#define RT_FIFO_CAPACITY 32
#endif


// Internal state.
typedef struct {
    EVENTO eventos_encolados[RT_FIFO_CAPACITY];
    volatile uint32_t head;                // next free position to write
    volatile uint32_t tail;                // next position with data to read (
        main)
    uint32_t estadisticas[EVENT_TYPES];    // per-event-type counters
    uint32_t total;                       // historical total enqueued
    uint32_t mon_overflow;                // monitor to assert on overflow (0 =>
        none)
    volatile uint32_t contador;            // current number of enqueued elements
        : 0..RT_FIFO_CAPACITY
} 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) {

    // Enter critical section
    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 -> infinite loop for diagnostics */ }
    }

    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++;

    // Statistics
    if (ID_evento < EVENT_TYPES) cola.estadisticas[ID_evento]++;
    cola.total++;

    // Exit critical section
    rt_sc_salir(sc);
}


uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t *auxData, Tiempo_us_t *TS)
     {

    // Enter critical section
    rt_sc_estado_t sc = rt_sc_entrar();

    uint32_t t = cola.tail;
    if (cola.contador == 0){
        rt_sc_salir(sc);
        return (uint8_t)-1;
    }

    if (ID_evento) *ID_evento = cola.eventos_encolados[t].ID_EVENTO;
    if (auxData) *auxData = cola.eventos_encolados[t].auxData;
    if (TS) *TS = cola.eventos_encolados[t].TS;

    cola.tail = (t + 1) % RT_FIFO_CAPACITY;
    cola.contador--;

    // Return number of events still pending after extraction.
    uint32_t pendientes = cola.contador;
    if (pendientes > 254) pendientes = 254;

    // Exit critical section
    rt_sc_salir(sc);

    return (uint8_t)pendientes;
}


uint32_t rt_FIFO_estadisticas(EVENTO_T ID_evento) {
    if (ID_evento == ev_VOID) return cola.total;
    if ((uint32_t)ID_evento < EVENT_TYPES) return cola.estadisticas[(uint32_t)
        ID_evento];
    return 0;
}
```

Appendix E: Random (LPC) (hal_random_lpc.c)

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

/*
 * We need access to a time counter to "simulate" entropy.
 * It is assumed that hal_tiempo has initialized Timer 1.
 * Otherwise, a fixed value will be returned and the system will rely on manual
     seeding in main.
 */

void hal_random_iniciar(void) {
    // The LPC2105 provides no specific hardware RNG that needs initialization.
}

uint8_t hal_random_generar_u8(void) {

    // Use the current Timer 1 value (T1TC) as a source of "noise".
    uint32_t entropia_simulada = T1TC;

    // Take the low byte.
    return (uint8_t)(entropia_simulada & 0xFF);
}
```

Appendix F: Random (nRF) (hal_random_nrf.c)

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

void hal_random_iniciar(void) {
    /*
     * Configure the random number generator.
     * Enable bias correction.
     */
    NRF_RNG->CONFIG = RNG_CONFIG_DERCEN_Enabled;

    // Start the task.
    NRF_RNG->TASKS_START = 1;
}

uint8_t hal_random_generar_u8(void) {
    // Clear the previous event for safety.
    NRF_RNG->EVENTS_VALRDY = 0;

    // Wait until a value is ready. Latency is nondeterministic but fast for 1
        byte.
    while (NRF_RNG->EVENTS_VALRDY == 0);

    // Clear the event for the next read.
    NRF_RNG->EVENTS_VALRDY = 0;

    // Read from the VALUE register.
    return (uint8_t)(NRF_RNG->VALUE);
}
```

Appendix G: Critical Sections (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();

    // For this lab we do not use the previous state value, so we return 0.
    return 0;
}

void rt_sc_salir(rt_sc_estado_t estado_prev){
    (void)estado_prev;   // Unused in this simplified version.
    __enable_irq();
}
```

Appendix H: Critical Sections (nRF) (rt_sc_nrf.c)

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

/*
 * - PRIMASK = 1  => IRQs disabled
 * - PRIMASK = 0  => IRQs enabled
 */

rt_sc_estado_t rt_sc_entrar(void)
{
    // Save the previous state.
    uint32_t primask = __get_PRIMASK();
    // Disable interrupts.
    __disable_irq();
    return primask;
}

void rt_sc_salir(rt_sc_estado_t estado_prev)
{
    // Restore the exact previous state.
    __set_PRIMASK(estado_prev);
}
```

Appendix I: Button Driver (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     // Press debounce time
#define RETARDO_SOLTAR_MS 50     // Release debounce time
#define RETARDO_ENCUESTA_MS 20   // Periodic polling interval (check if still
    pressed)
#define PULSACION_LARGA 3000     // 3 seconds as long-press threshold

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 when the button was pressed
    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
};

// Upper-layer callback.
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();

    // Initial state: waiting for presses.
    for (uint8_t i = 0; i < BUTTONS_NUMBER; ++i) {
        botones[i].estado = e_esperando;
    }

    // Enable SENSE (LOW) on all buttons.
    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);
    }
}
```

```c


// Called from the HAL ISR upon press detection.
void drv_botones_pulsar_RSI(uint8_t id_boton){

    if (id_boton == 0 || id_boton > BUTTONS_NUMBER) return;

    // Only process if we are waiting (prevents creating multiple alarms).
    if (botones[id_boton - 1].estado != e_esperando) {
        return;  // This button is already being processed; ignore.
    }

    // Button pressed: store timestamp.
    botones[id_boton - 1].inicio_pulsacion = drv_tiempo_actual_ms();

    // Reset long-press flag.
    botones[id_boton - 1].larga_enviada = false;


    // NOTE: For LPC with level-triggered interrupts, the ISR already disables
        the interrupt.
    // For NRF with edge-triggered interrupts, we must disable it here.
    // Since this function runs in ISR context, LPC already has it disabled.
    hal_ext_int_deshabilitar(id_boton);

    svc_alarma_activar(svc_alarma_codificar(0, RETARDO_REBOTE_MS, 0),
        ev_BOTON_RETARDO, id_boton);

    botones[id_boton - 1].estado = e_rebotes;

}

void drv_botones_actualizar(EVENTO_T evento, uint32_t auxiliar){
    uint32_t ahora; // used to measure how long the button has been pressed
    uint8_t id_boton = (uint8_t)auxiliar;

    // Out-of-range button? Return...
    if (id_boton == 0 || id_boton > BUTTONS_NUMBER) {
        return;
    }

    switch (evento) {

        // Internal event: timeout for debounce/release alarms.
        case ev_BOTON_RETARDO:
            switch (botones[id_boton - 1].estado){

                // We just detected the ISR and are in debounce.
                case e_rebotes:
                    if (hal_ext_int_obtener_estado(id_boton)) {
                        // Still pressed after delay: valid press.
                        botones[id_boton - 1].estado = e_muestreo;

                        if (drv_botones_callback) {
                            drv_botones_callback(ev_BOTON_PULSAR, id_boton);
                        }

                        // Schedule alarm for periodic checking.
                        svc_alarma_activar(svc_alarma_codificar(0,
                            RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton)
                            ;
                    } else {
                        // Noise/bounce: return to waiting and re-enable.
                        botones[id_boton - 1].estado = e_esperando;
                        hal_ext_int_habilitar(id_boton);
                    }
                    break;

                // Waiting for the button to be released.
                case e_muestreo:
                    ahora = drv_tiempo_actual_ms();

                    // Check whether it is still physically pressed.
                    if(hal_ext_int_obtener_estado(id_boton)) {

                        // Compute press duration.
                        uint32_t duracion = ahora - botones[id_boton - 1].inicio_pulsacion;

                        // If pressed for >= 3 seconds AND not yet notified AND from button 3 or 4 (id_boton >= 3).
                        if (duracion >= PULSACION_LARGA && !botones[id_boton - 1].larga_enviada && id_boton >= 3) {

                            // Emit the event to the game.
                            if (drv_botones_callback) {
                                drv_botones_callback(ev_PULSACION_LARGA, id_boton);
                            }

                            // Mark as sent to avoid repeated notifications.
                            botones[id_boton - 1].larga_enviada = true;
                        }

                        // Keep polling periodically while pressed.
                        svc_alarma_activar(
                            svc_alarma_codificar(0, RETARDO_ENCUESTA_MS, 0),
                            ev_BOTON_RETARDO,
                            id_boton
                        );

                    } else {
                        // No longer pressed: transition to release state.
                        botones[id_boton - 1].estado = e_salida;

                        // Release debounce delay.
                        svc_alarma_activar(svc_alarma_codificar(0, RETARDO_SOLTAR_MS, 0), ev_BOTON_RETARDO, id_boton);
                    }
                    break;

                // After release, wait for stable idle before re-arming SENSE.
                case e_salida:
                    if (!hal_ext_int_obtener_estado(id_boton)) {
                        // Ready for a new press.
                        botones[id_boton - 1].estado = e_esperando;
                        hal_ext_int_habilitar(id_boton);
                    } else {
                        // If still pressed, return to sampling.
                        botones[id_boton - 1].estado = e_muestreo;
                        svc_alarma_activar(svc_alarma_codificar(0, RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton);
                    }
                    break;

```

```
                case e_esperando:
                default:
                    break;
            }
            break;

        case ev_BOTON_PULSAR:
            break;

        default:
            break;
    }
}
```

Appendix J: Alarm Service (svc_alarmas.c)

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

// Internal structure
typedef struct{
    bool activa;
    svc_alarma_flags_t flags;       // Activation flags encoding.
    uint32_t restante_ms;           // Remaining countdown in ms.
    EVENTO_T id_evento;             // Event to enqueue upon expiration.
    uint32_t aux;                   // auxData carried with the event.
} alarma_t;

typedef struct{
    alarma_t alarmas[svc_ALARMAS_MAX];                       // All alarms in the system.
    MONITOR_id_t monitor_overflow;                           // Monitor asserted when no alarm slots remain.
    void (*cb_encolar)(uint32_t id_ev, uint32_t aux);        // Enqueue function
    EVENTO_T ev_tick;                                        // Tick event that triggers updates.
} sistema;

static sistema s;

// Auxiliary (internal) functions.
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;

        // For button debounce, also distinguish by aux (button id).
        if (ev == ev_BOTON_RETARDO) {
            if (s.alarmas[i].aux == aux) return i;
        } else {
            // For other events, keep the original semantics.
            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) {
        s.cb_encolar((uint32_t)a->id_evento, a->aux);
    }
}

// Public operations
void svc_alarma_iniciar(uint32_t monitor_overflow, void (*cb_encolar)(uint32_t
    id_ev, uint32_t aux), EVENTO_T ev_a_notificar){

    for (int i = 0; i < (int)svc_ALARMAS_MAX; ++i) {
        s.alarmas[i].activa = false;
        s.alarmas[i].flags = 0;
        s.alarmas[i].restante_ms = 0;
        s.alarmas[i].id_evento = ev_VOID;
        s.alarmas[i].aux = 0;
    }

    s.monitor_overflow = monitor_overflow;
    s.cb_encolar = cb_encolar;
    s.ev_tick = ev_a_notificar;
}

void svc_alarma_activar(svc_alarma_flags_t flags, EVENTO_T id_evento, uint32_t
    auxData){
    rt_sc_estado_t sc = rt_sc_entrar();
    // Check whether the alarm for the event already exists and is active.
    int id_alarma = buscar_por_evento(id_evento, auxData);

    // If retardo_ms == 0 => CANCEL the alarm (if it exists).
    uint32_t retardo = periodo_ms(flags);
    if (retardo == 0) {
        if (id_alarma >= 0) {
            s.alarmas[id_alarma].activa = false;
        }
        return;
    }

    // If we are not re-arming, find the next free alarm slot.
    if (id_alarma < 0) id_alarma = buscar_libre();

    // If none are free, assert the monitor.
    if (id_alarma < 0) {
        if (s.monitor_overflow) drv_monitor_marcar(s.monitor_overflow);
        while(true){ /* No alarm slots available */ }
    }

    // If re-arming or creating, fill in the new data.
    s.alarmas[id_alarma].activa = true;
    s.alarmas[id_alarma].flags = flags;
    s.alarmas[id_alarma].restante_ms = retardo;
    s.alarmas[id_alarma].id_evento = id_evento;
    s.alarmas[id_alarma].aux = auxData;
    rt_sc_salir(sc);
}

void svc_alarma_actualizar(EVENTO_T id_evento, uint32_t auxData){

    if (id_evento != s.ev_tick) return;

    // Treat auxData as ms elapsed since the last tick (if 0, use at least 1).
    uint32_t transcurrido = (auxData == 0) ? 1 : auxData;

    rt_sc_estado_t sc = rt_sc_entrar();

    for(int i = 0; i < (int)svc_ALARMAS_MAX; ++i){

        if(!s.alarmas[i].activa) continue;

        if(transcurrido >= s.alarmas[i].restante_ms){
            // EXPIRES
            disparar(&s.alarmas[i]);

            if (es_periodica(s.alarmas[i].flags)) {
                // Periodic re-arming.
                uint32_t periodo = periodo_ms(s.alarmas[i].flags);
                uint32_t overshoot = transcurrido - s.alarmas[i].restante_ms;

                if (periodo == 0) periodo = 1;
                // If overshoot is a multiple of the period, fire once and keep a full period remaining.
                s.alarmas[i].restante_ms = periodo - (overshoot % periodo);
                if (s.alarmas[i].restante_ms == 0) s.alarmas[i].restante_ms = periodo;
            } else {
                // One-shot: deactivate.
                s.alarmas[i].activa = false;
            }
        }else{
            // Not yet expired: decrement.
            s.alarmas[i].restante_ms -= transcurrido;
        }
    }

    rt_sc_salir(sc);
}
```